Modélisation de performance et simulation d'applications OpenMP

Idriss Daoudi

université de BORDEAUX

Inria

THÈSE PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Monsieur Idriss DAOUDI**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ: INFORMATIQUE

Modélisation de performance et simulation d'applications OpenMP

Date de soutenance: 21 septembre 2021

Membres du jury:

Emmanuel JEANNOT	Directeur de Recherche	INRIA Bordeaux	Président de Jury
Samuel THIBAULT	Professeur	Université de Bordeaux	Directeur de thèse
Thierry GAUTIER	Chargé de Recherche	INRIA Grenoble	Directeur de thèse
Gaël THOMAS	Professeur	Telecom SudParis	Rapporteur
Camille COTI	Maître de conférences	Université de Paris 13	Examinatrice
Fabrice RASTELLO	Directeur de Recherche	INRIA Grenoble	Examinateur
Isabelle D'AST	Ingénieur Logiciel HPC	CERFACS	Invitée
Luka STANISIC	Ingénieur de Recherche	Huawei Technologies	Invité

Après avis de:

Henri CASANOVA	Professeur	Université de Hawaï	Rapporteur
Gaël THOMAS	Professeur	Télécom SudParis	Rapporteur

Modélisation de performance et simulation d'applications OpenMP

Résumé : Anticiper le comportement des applications, étudier et concevoir des algorithmes sont quelques-uns des objectifs les plus importants pour les études de performances et de correction des simulations et des applications liées au calcul intensif. De nombreux frameworks ont été conçus pour simuler de grandes infrastructures informatiques distribuées et les applications qui y sont exécutées. Au niveau des noeuds, certains outils ont également été proposés pour simuler des applications parallèles basées sur des tâches. Cependant, une capacité critique manquante à ces travaux est la capacité à prendre en compte les effets d'accès non uniforme à la mémoire (NUMA, *Non-Uniform Memory Access*), même si pratiquement toutes les plateformes HPC (*High Performance Computing*, i.e. calcul haute performance) présentent aujourd'hui de tels effets. Nous modélisons différentes architectures à mémoire partagée en effectuant nos propres mesures pour en obtenir les caractéristiques. Nous présentons donc dans cette thèse un nouveau simulateur d'applications parallèles à base de tâches dépendantes, qui permet d'expérimenter plusieurs modèles de localité des données. Celui-ci s'appuie sur l'enregistrement d'une trace de l'exécution séquentielle de l'application cible, en utilisant l'interface standard de trace pour OpenMP, OMPT (*OpenMP Trace*). Nous introduisons également trois modèles de performances dont deux sont sensibles à la localité : un premier modèle qui ne prend en compte que les temps d'exécution des tâches, un modèle léger qui utilise des informations de topologie pour pondérer les transferts de données, et enfin un modèle plus complexe qui prend en compte le stockage de données dans le LLC (*Last Level Cache*, i.e. cache de dernier niveau, en général le L3). Nous validons nos modèles sur des cas tests d'algèbre linéaire dense et montrons qu'en moyenne, notre simulateur prédit de manière reproductible et rapide le temps d'exécution avec une erreur relative réduite et permet l'expérimentation et l'étude de diverses heuristiques d'ordonnancement.

Mots-clés : systèmes à mémoire partagée, OpenMP, tâches, simulation, modélisation des performances

---

Unité de recherche :
Laboratoire Bordelais de Recherche en Informatique (LaBRI),
UMR 5800, Université de Bordeaux, 33400 Talence, France.

Performance Modelling and Simulation of OpenMP Applications

Abstract: Anticipating the behavior of applications, studying, and designing algorithms are some of the most important purposes for the performance and correction studies about simulations and applications relating to intensive computing. Many frameworks were designed to simulate large distributed computing infrastructures and the applications running on them. At the node level, some frameworks have also been proposed to simulate task-based parallel applications. However, one missing critical capability from these works is the ability to take Non-Uniform Memory Access (NUMA) effects into account, even though virtually every HPC (High Performance Computing) platform nowadays exhibits such effects. We model different shared-memory architectures by performing our own measures in order to obtain their characteristics. We thus present in this PhD a new simulator for dependency-based task-parallel applications, that enables experimenting with multiple data locality models. It is based on collecting a trace of the sequential execution of the targeted application using the standard OpenMP tracing interface, OMPT (OpenMP Trace). We also introduce three models, two of them being locality-aware performance models: a first model that only takes into account tasks execution time, a lightweight model that uses topology information to weight data transfers, and eventually a more complex model that takes into account data storage in the LLC (Last Level Cache, generally L3). We validate both models on dense linear algebra test cases and show that, on average, our simulator reproducibly and quickly predicts execution time with a small relative error and allows the experimentation and studying of various scheduling heuristics.

Keywords: shared-memory systems, OpenMP, tasks, simulation, performance modeling

---

Research unit:
Laboratoire Bordelais de Recherche en Informatique (LaBRI),
UMR 5800, Université de Bordeaux, 33400 Talence, France.

Remerciements

Je souhaite remercier, et en premier lieu, Dieu, pour toutes les bénédictions qu'il m'a offertes.

Je voudrais ensuite exprimer tous mes remerciements et toute ma gratitude à mes directeurs de thèse, M. Samuel Thibault et M. Thierry Gautier. Ce travail n'aurait jamais pu être possible sans leur implication, leur inverstissement et surtout leur confiance infaillible en moi. En plus de son aspect scientifique, une thèse a aussi un aspect humain et je suis ravi de les avoir cotoyé durant ces trois années. Merci pour votre soutien et vos conseils durant l'élaboration de ce travail.
Je remercie également les rapporteurs de thèse, M. Henri Casanova et M. Gael Thomas pour avoir accepté de lire minutieusement ce manuscrit. Merci pour vos rapports et vos commentaires très encourageants. Je tiens aussi à remercier tous les membres du jury, plus particulièrement M. Emmanuel Jeannot qui a accepté de présider le jury de thèse. Merci d'avoir accepté de faire partie de ce travail.
J'adresse aussi mes remerciements à tous les gens qui ont participé de près ou de loin à ce travail. Qu'ils soient membres de l'Inria comme M. Brice Goglin, M. Francois Rué, M. Julien Lelaurain, ou même extérieurs comme Dounya, Philippe, Antoine, Stéphane et Abdesslam, merci d'avoir contribué au bon déroulement de cette thèse. Votre présence et vos conseils m'ont été d'une aide précieuse tout au long de ce périple.

Je souhaite remercier ma mère. Tes sacrifices n'ont pas été vains. Enfin, tout ce travail -et tout mon cursus- est dédié à mon père.

Table des matières

Chapitre 1

Introduction

Sommaire

1.1 Entre performance et complexité

Le calcul haute performance (HPC) est l'utilisation de puissances de calcul agrégées pour offrir des performances supérieures à celle possibles avec un ordinateur de bureau standard. Employé dans tous les domaines, il est utilisé pour résoudre des problèmes à grande échelle qui seraient autrement **trop longs ou même impossibles à calculer**, et devient de plus en plus important à mesure que les ensembles de données s'agrandissent et que les paramètres des expérimentations se multiplient.

Afin de répondre aux besoins croissants des scientifiques en termes de **vitesse de calcul**, les fabricants de puces ont développé des processeurs complexes caractérisés par une fréquence de plus en plus élevée, une microachitecture de plus en plus puissante (capable d'exécution spéculative et dans le désordre) et des caches plus grands. Ce développement a atteint ses limites lorsque les problèmes de dissipation de puissance et les rendements décroissants sont devenus conséquents, forçant les fabricants à les abandonner.

La communauté informatique a donc été témoin d'un évènement marquant au début du $21^{ième}$ siècle avec l'introduction des processeurs multicœurs pour maintenir les performances : les cœurs multiples fonctionnent avec une fréquence plus faible, se partagent des caches plus grands, ont une hiérarchie mémoire à plusieurs niveaux à l'intérieur même de la puce, et permettent des performances parallèles globales plus élevées et une meilleure efficacité énergétique que leurs ancêtres à cœur unique [1]. Le constructeur Intel a par exemple abandonné le processeur très complexe *Pentium 4* au profit d'une ancienne famille de processeurs nommée *Pentium M* fonctionnant avec une consommation d'énergie moyenne très faible, une production de chaleur

[1] `https://techterms.com/definition/multi-core`

	Flux de données	
Flux d'instructions	SISD	SIMD
	MISD	**MIMD**

TABLE 1.1 : La taxonomie de Flynn

et une vitesse d'horloge bien inférieures à celles de la version *Pentium 4*, mais avec des performances similaires (un *Pentium M* à 1.6GHz peut typiquement atteindre, voire dépasser, les performances d'un *Pentium 4* à 2.4GHz[1]), et qui sera la base des générations multicœurs suivantes (Intel *Core*).

De manière générale et selon plusieurs critères, il existe différentes manières de classer les processeurs. La classification la plus répandue est appelée la taxonomie de Flynn [2]. Le tableau 1.1 résume cette classification :

- SISD (*Single Instruction, Single Data* ; unique flux d'instructions, unique flux de données) : représente les machines séquentielles traditionnelles. En d'autres termes, une unité de traitement exécute les instructions d'un seul flux de manière séquentielle ;

- SIMD (*Single Instruction, Multiple Data* ; unique flux d'instructions, multiples flux de données) : dans ce type d'architectures, toutes les unités de traitement exécutent la même instruction en même temps et se synchronisent. Cependant, les différentes unités de traitement fonctionnent sur des données différentes. Les processeurs vectoriels appartiennent à la classe des architectures SIMD.

- MISD (*Multiple Instructions, Single Data* ; multiples flux d'instructions, unique flux de données) : représente des architectures dans lesquelles un seul flux de données est introduit dans plusieurs unités de traitement. Chacune opère sur les données de manière indépendante avec différents flux d'instructions. Cependant, même si toutes les unités fonctionnent sur le même flux de données, chaque unité fonctionne sur une donnée distincte ;

- **MIMD** (*Multiple Instruction, Multiple Data* ; multiples flux d'instructions, multiples flux de données) : représente des machines multiprocesseurs où chaque processeur exécute son propre code indépendamment et de manière asynchrone des autres. Chaque processeur fonctionne sur son propre flux de données. Les processeurs communiquent des données entre eux. La plupart des supercalculateurs, machines parallèles et multicœurs actuelles suivent ce design.

La classification de Flynn souffre cependant de certaines limitations. La classe MIMD par exemple comprend une grande variété d'architectures. Pour cette raison, Johnson [3] a proposé une classification plus poussée de ces machines, qui est basée sur leur organisation mémoire (partagée ou distribuée) et le mécanisme utilisé pour les communications / synchronisations (variables partagées ou passage de messages). Le tableau 1.2 résume cette classification :

	Communications / synchronisations	
Organisation de la mémoire	GMSV	GMMP
	DMSV	DMMP

TABLE 1.2 : La classification de Johnson pour les machines MIMD

- GMSV (*Global Memory, Shared Variables* ; mémoire partagée, variables partagées) : représente les multiprocesseurs traditionnels à mémoire partagée, également connus sous le nom de systèmes multiprocesseurs symétriques (SMP) [4]. Tous les cœurs partagent la mémoire et les entrées/sorties (IO) de manière égale et peuvent accéder au même emplacement mémoire à la même vitesse, fournissant ainsi un accès uniforme à la mémoire (**UMA**, *Uniform Memory Access*, figure 1.1 à gauche) ;

- GMMP (*Global Memory, Message Passing* ; mémoire partagée, passage de message) : représente des machines implémentant un espace d'adressage global et où les communications sont réalisées au moyen de passages de messages au lieu d'utiliser des variables partagées ;

- DMMP (*Distributed Memory, Message Passing* ; mémoire distribuée, passage de message) : cette architecture implémente un modèle où la mémoire est physiquement distribuée sans aucune possibilité d'accéder aux données distantes de manière transparente. La communication de données est réalisée explicitement par le passage d'un message à travers un réseau de communication ;

- DMSV (*Distributed Memory, Shared Variables* ; mémoire distribuée, variables partagées) : représente les machines où la mémoire est physiquement distribuée entre les noeuds (ou groupe de cœurs), mais tous ces cœurs en parallèle ont la même mémoire partagée permettant un accès aux données locales (à l'intérieur des noeuds) et à distance (dans les mémoires partagées des autres noeuds). La mémoire est par conséquent plus proche de certains cœurs qui peuvent alors accéder à certains emplacements mémoire plus rapidement que d'autres, fournissant un accès non uniforme à la mémoire : on parle alors de systèmes **NUMA** (*Non-Uniform Memory Access* [5], figure 1.1 à droite), et qui sont l'objet de ce travail.

Pour exploiter donc pleinement les machines NUMA et afin de faire des calculs toujours plus rapides, les scientifiques doivent découper les applications de résolution de leurs problèmes respectifs en morceaux. Un ordonnanceur s'occupe alors de les soumettre à l'exécution sur les différents cœurs en parallèle, tout en respectant les contraintes de dépendances entre les résultats intermédiaires. La recherche pour développer ces ordonnanceurs est donc importante : il faut étudier et tester diverses stratégies d'ordonnancement pour déterminer laquelle offre les meilleures performances pour un type d'applications spécifique, en d'autres termes, savoir dans quel ordre effectuer les tâches, où placer les données, et où placer ces tâches. Seulement, ces recherches se heurtent à un sérieux problème de reproductibilité : les systèmes récents étants de plus en plus complexes, il est difficile d'obtenir toujours les mêmes performances pour les mêmes paramètres afin de comparer les résultats, que se soit à

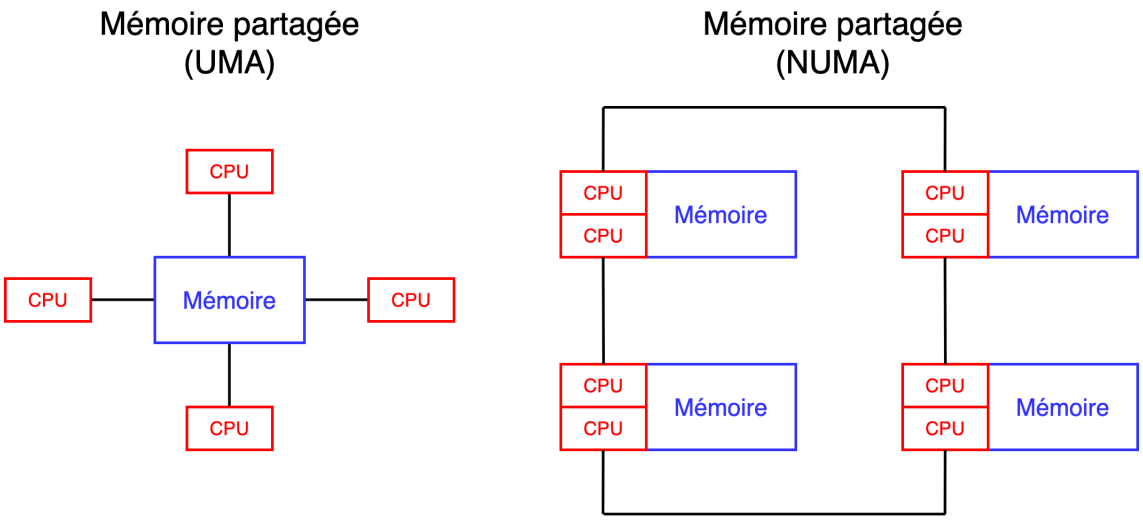


FIGURE 1.1 : Aperçu des architecture à mémoire partagée UMA et NUMA

cause de la mise à jour du BIOS qui a impacté la fréquence des cœurs, de la chaleur de la pièce, des mises à jour logicielles qui changent les durées d'exécution des tâches (et donc potentiellement les valeurs des compromis à trouver) ou encore d'autres variables. Pour palier ce problème de reproductibilité, la simulation de l'exécution d'applications nécessitant l'ordonnancement peut être envisagée.

1.2 La simulation au service de la science

La simulation est un élément clé dans l'étude de plusieurs domaines. Diverses disciplines scientifiques utilisent la simulation pour explorer et mieux comprendre les phénomènes observés [6]. Que se soit en physique, en chimie ou en médecine, la simulation est aussi utilisée dans le domaine de l'informatique afin de modéliser le comportement des architectures, des applications, etc... La simulation peut donc nous permettre de comprendre les éléments et phénomènes qui interviennent dans l'exécution d'une application, d'autant plus qu'elle a déjà montré tout son intérêt dans la résolution des problèmes de reproductibilité sur d'autres systèmes.

En effet, dans le cadre du développement d'applications à base de tâches par exemple, la simulation a permis de comprendre si elles ont été correctement conçues, si elles sont exécutées efficacement par l'ordonnanceur, et de tester leurs limites et leur sensibilité au matériel et aux réseaux sur les plateformes basées sur GPU [7]. Cela a ouvert la porte à une recherche sur de telles plateformes qui soit à la fois réaliste et reproductible [8]. La simulation a permis donc un prototypage rapide, avant la mise en œuvre réelle pour des systèmes réels. Cependant, l'approche utilisée s'est révélée être simpliste et fournissait des résultats peu précis pour les architectures NUMA. La complexité de ces dernières, avec leur hiérarchie de mémoire, freine l'expérimentation sur de telles plateformes en plus des autres contraintes techniques (vieillissement du matériel, évolution du système...), et les travaux précédents [7] ont par ailleurs montré que simuler et prédire avec précision les performances des plateformes actuelles nécessite fortement de prendre en compte les effets NUMA.

Le travail proposé ici est alors décomposé en deux axes majeurs :

- la **simulation d'une architecture** en prenant en compte la hiérarchie mémoire ;
- la **simulation d'une application** à base de tâches en prenant en compte les effets NUMA.

Atteindre un niveau comparable de qualité de simulation pour une application sur une architecture à mémoire partagée (NUMA) permettra de même que dans le cas GPU de concevoir, prototyper et éventuellement mettre en œuvre rapidement le bon ordonnanceur pour le bon matériel et/ou type d'application, grâce à une méthodologie robuste et reproductible basée sur des simulations fiables.

1.3 Organisation du document

Face à la croissance de la complexité des architectures modernes, la simulation est une piste sérieuse pour prédire les performances des applications et s'impose même comme une nécessité pour palier aux problèmes de la reproductibilité. Avec un modèle d'accès non uniforme à la mémoire (NUMA), la localité des données et leurs mouvements sont des composants clé pour comprendre le comportement et les performances des applications étudiées. Mais alors des questions se posent : comment fonctionne la mémoire dans les processeurs récents ? Comment modélise-t-on une architecture NUMA ? Comment peut-on simuler une application OpenMP à base de tâches ? Comment modélise-t-on les performances ? Pouvons-nous développer un outil qui permettra l'étude des applications dans un environnement reproductible ? Ce manuscrit traite ces questions de manière détaillée, et s'articule comme suit :

- le Chapitre 2 décrit le contexte général de la thèse en détaillant la structure mémoire des architectures modernes ainsi que les différents paradigmes de la programmation parallèle, afin de mieux comprendre les enjeux et la motivation de ce travail ;
- le Chapitre 3 présente l'expression d'une architecture NUMA à l'aide du framework SimGrid et les défis rencontrés qui nous ont poussé à développer notre propre benchmark pour affiner la modélisation ;
- le Chapitre 4 détaille la conception de notre simulateur pour la prédiction des performances d'applications OpenMP d'algèbre linéaire dense à base de tâches ;
- Le Chapitre 5 expose plusieurs contributions pour modéliser les performances de ces applications en prenant en considération les effets NUMA ;
- le Chapitre 6 présente les résultats et des exemples palpables des avantages offerts par notre approche.

Chapitre 2

Contexte et motivation

Sommaire

Dans ce chapitre, on présentera le cadre général de ce travail : en explicitant d'abord les principes de conception des architectures NUMA et de leurs systèmes de gestion de la mémoire, on discutera ensuite de la programmation à base de tâches pour avoir une idée générale des motivations et défis relevés par cette thèse, en finissant par présenter l'état de l'art.

2.1 Les architectures NUMA

L'utilisation du design NUMA pour les machines physiques à mémoire partagée est l'une des tendances clé de la conception matérielle. Le design NUMA est utilisé par de nombreux constructeurs d'architectures multicœurs, et il est prévu que les modèles multicœurs émergeants adoptent une conception NUMA également.

L'objectif principal de ce design est de fournir une scalabilité des performances pour les machines multicœurs avec une grande mémoire principale : la mémoire est partitionnée en plusieurs régions NUMA, chacune étant associée à un groupe de plusieurs cœurs via un contrôleur mémoire, comme observé sur la figure 2.1 : les cœurs (en bleu clair) sont inter-connectés par des liens (en rouge) offrants différentes routes pour accéder à la mémoire partagée via 2 contrôleurs mémoire (en gris), à la

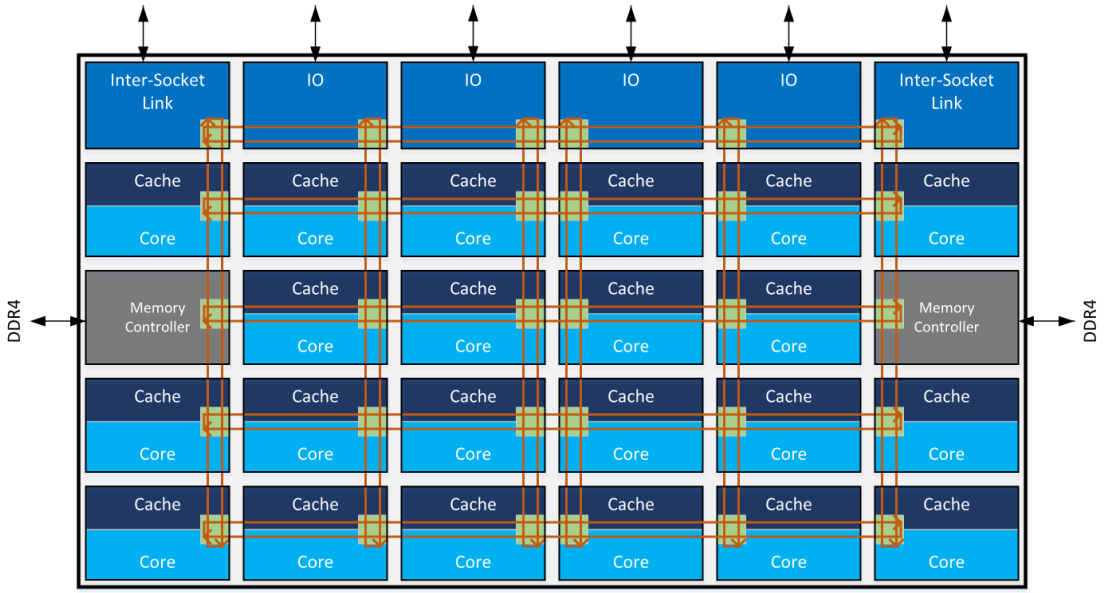


FIGURE 2.1 : Représentation conceptuelle d'un socket dans l'architecture Intel Skylake [9]

mémoire principale via les points d'entrée/sortie (*IO* en bleu foncé) ou à la mémoire des autres sockets (*Inter-Socket Link* en bleu foncé).

L'accès à la mémoire (c'est-à-dire au contrôleur mémoire) dans la région locale est très rapide, alors que l'accès non local (à distance) doit passer par un réseau de liens sur puce pour accéder à un banc mémoire différent et peut être beaucoup plus lent : si le placement des données et les calculs font en sorte que *plusieurs* transits de données passent par les mêmes liens, ces flux se ralentissent entre eux et les calculs sont par conséquent aussi ralentis, puisqu'ils sont en attente de données. De plus, cette asymétrie de performances devient plus importante lorsque le nombre de cœurs dans une même région augmente, ce qui introduit plus d'effets de contention (affectant la bande passante des liens traversés) et de concurrence (affectant la latence des liens traversés). Pour maximiser l'utilisation d'une machine NUMA, un développeur doit donc minimiser le nombre d'accès mémoire à distance et s'assurer que la charge soit équilibrée entre les contrôleurs mémoires. Pour faire cela, il doit soigneusement choisir dans quel nœud NUMA il place ses données et sur quels cœurs il place ses calculs. C'est donc un travail complexe, d'autant plus que les architectures modernes sont variées et amènent chacunes leurs propres défis. L'un des objectifs du travail en ordonnancement est donc d'automatiser ces prises de décisions, d'où notre objectif d'offrir un environnement pour mener à bien ces recherches.

2.2 La hiérarchie mémoire

Dans les processeurs modernes, les unités arithmétiques et logiques (ALU) effectuent des opérations sur des registres. La quantité totale de données pouvant être stockées dans les registres est très faible (moins d'1KB), de sorte que les processeurs les chargent souvent puis les vident. Par ailleurs, l'accès à la mémoire principale

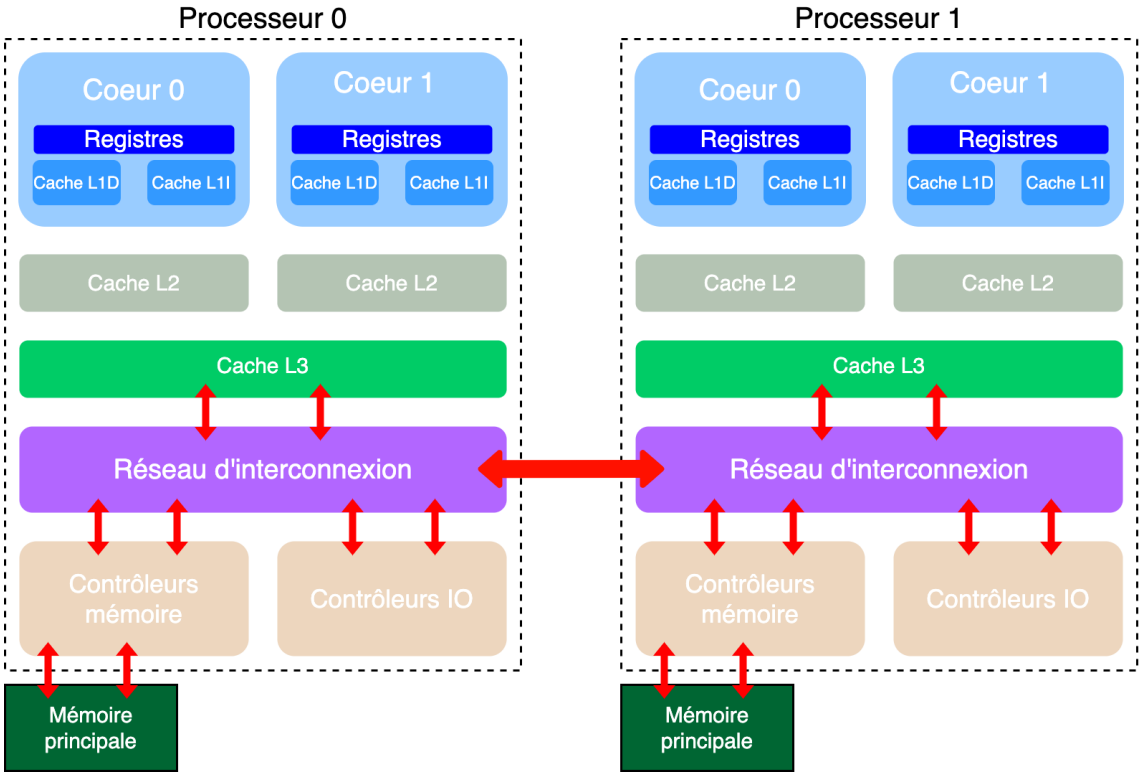


FIGURE 2.2 : Hiérarchie mémoire dans les architectures NUMA

(RAM) est lent : entre 100 [1] et 2000 cycles sur les architectures récentes (pour comparer, les ALU sont capables d'effectuer jusqu'à 64 opérations sur des registres dans un seul cycle[2]).

Ainsi donc, pour minimiser les accès à la mémoire principale, les architectures modernes incluent des caches auxquels les cœurs accèdent avec une latence beaucoup plus faible. Comme explicité dans la figure 2.2, les caches sont généralement organisés en 3 niveaux et permettent le stockage des données avec une granularité en lignes de cache, qui correspond généralement à 64B : les transferts entre niveaux de cache ou entre un cache et la mémoire principale se fait donc par morceaux de 64B.

Un cœur travaille donc d'abord sur les données présentes dans les registres. Si elles ne s'y trouvent pas, il faut alors les récupérer depuis le premier niveau de cache (L1) qui est une petite mémoire (entre 32 et 64KB) divisée en deux (L1D pour les données, L1I pour les instructions) à laquelle on accède avec une faible latence. Si les données requises par le processeur ne sont pas présentes dans le cache L1, le processeur les cherche dans les niveaux de cache supérieurs : d'abord dans le cache L2, qui, comme exposé sur la figure 2.3, est plus gros que le cache L1 (entre 256KB et 12MB, souvent partagés entre 2 cœurs) et est accessible à une latence plus élevée. Enfin, le cache L3 (aussi appelé LLC pour *Last Level Cache*) est encore plus gros (entre 5MB et plusieurs centaines de MB) et est généralement partagé entre tous les cœurs du banc NUMA, tout en étant accessible à une latence encore plus élevée.

Si les données requises par un cœur ne sont présentes dans aucun cache, il enverra

[1] https://developers.redhat.com/blog/2016/03/01/reducing-memory-access-times-with-caches
[2] https://en.wikichip.org/wiki/flops

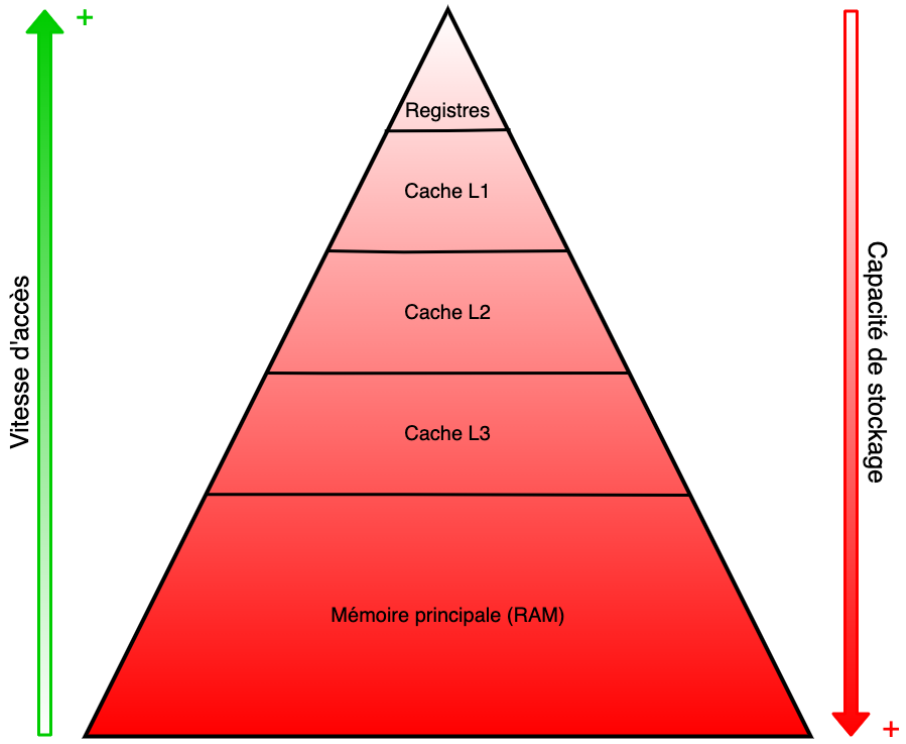


FIGURE 2.3 : Vitesse d'accès et capacité de stockage des composants mémoire dans les architectures NUMA

une requête pour les récupérer depuis la mémoire principale (RAM) ou depuis les caches des autres bancs NUMA, impactant significativement les performances. Donc, lorsque les données sont répliquées dans le cache, ces données y restent et peuvent être dans plusieurs caches en même temps. Par conséquent, lorsque les cœurs modifient ces données, les architectures actuelles utilisent des protocoles pour assurer la cohérence entre caches : cette cohérence garantit que tout changement dans les données d'un cache soit reflété par un changement dans tous les autres caches qui peuvent avoir une copie du même emplacement des données. Il garantit aussi que tout chargement ou stockage de données dans un registre de processeur, s'il est acquis à partir du cache local, sera correct, même si un autre processeur utilise les mêmes données. Le réseau d'interconnexion qui assure la cohérence du cache peut utiliser plusieurs techniques. L'une des premières est le protocole MESI [10] (*Modified*, *Exclusive*, *Shared*, *Invalid*), parfois appelé *snooping cache*, dans lequel un bus partagé est utilisé pour permettre à toute écriture d'un processeur dans la mémoire d'être détectée par tous les autres processeurs et vérifiée pour voir si le même emplacement mémoire est mis en cache localement. Si c'est le cas, une indication est enregistrée et le cache est soit mis à jour, soit au moins invalidé, de sorte qu'aucune erreur ne se produise.

Un cache peut donc être vu comme une liste contenant des données utilisées par les cœurs. Ainsi, lorsque des données récupérées depuis la RAM ou depuis un cache distant doivent être insérées dans un cache déjà plein, il faut libérer de l'espace dans ce cache pour pouvoir y placer les données nouvellement requises. Le protocole LRU [11] (*Least Recently Used*) permet alors de placer ces données en tête de liste, et d'évincer les données les moins récemment utilisées par les cœurs (qui se retrouvent

donc en queue de liste) afin de libérer de l'espace de stockage. On voit donc que la complexité de la gestion de la mémoire dans les architectures NUMA implique que le choix du placement des données est l'un des enjeux majeurs pour obtenir les meilleurs performances : leur localité, c'est-à-dire leur proximité avec les cœurs qui les utilisent, impactent significativement les performances.

2.3 Le paradigme de programmation à base de tâches

2.3.1 La programmation parallèle

De manière générale, il existe une variété de modèles de programmation pour prendre en compte le parallélisme. Nous distinguons principalement trois modèles de programmation largement utilisés dans le domaine de la programmation parallèle :

- le modèle de programmation par passage de messages (*message passing programming model*) pour les architectures distribuées ;
- **le modèle de programmation en mémoire partagée** (*shared memory programming model*) et qui est la cible de ce travail ;
- le modèle de programmation hybride (*hybrid programming model*) qui combine les deux modèles précédents.

2.3.2 La programmation en mémoire partagée : OpenMP

L'introduction des architectures à mémoire partagée a nécessité le développement d'outils et de nouvelles façons de programmer afin de tirer parti de toute la puissance disponible : cela consiste en l'utilisation de plusieurs ressources (en l'occurrence des processeurs) pour résoudre un problème, qui est décomposé en une série d'étapes plus petites exécutées en même temps par plusieurs unités de calcul qui communiquent via la mémoire partagée, en lisant et écrivant des données de manière asynchrone. Du point de vue du programmeur, il n'y a donc pas de notion de communication **explicite** des données (à la différence du modèle de programmation par passage de messages). Grâce à ce principe, l'effort de programmation est grandement simplifié. Les protocoles de cohérence de cache jouent un rôle important dans la synchronisation de toutes les copies de données à différents emplacements dans la mémoire. Cependant, ces protocoles seuls ne sont pas en mesure d'empêcher l'accès simultané aux données partagées. Par conséquent, pour assurer la cohérence, des mécanismes de synchronisation explicites tels que des verrous et des sémaphores sont nécessaires pour contrôler les accès concurrents.

Une des normes réputées de programmation parallèle sur les architectures à mémoire partagée est **OpenMP** (*Open Multi-Processing*), qui a été proposée en 1997 par SGI et KAI et est destinée à servir de méthode portable pour écrire du code sur différentes architectures à mémoire partagée. OpenMP fournit des abstractions de programmation parallèle principalement sous la forme de directives de compilateur. Les programmeurs peuvent donc paralléliser un programme séquentiel existant écrit en C, C++ ou Fortran en insérant des directives OpenMP à des emplacements se prêtant

à la parallélisation. Le compilateur transforme alors le programme conformément aux directives présentes au début des blocs de code qui sont exécutés en parallèle par un système d'exécution spécifique à l'architecture : ces directives permettent la réalisation rapide de la programmation basée sur des threads de bas niveau avec la parallélisation des boucles *for* (1997), la **programmation basée sur les tâches** (2008 puis 2013 avec les dépendances), la vectorisation (2013) ou encore l'exécution de code sur les accélérateurs (2013 puis 2015). Par ailleurs, OpenMP fournit également des fonctions système d'exécution et des variables d'environnement permettant aux programmeurs de contrôler et de suivre le processus de parallélisation.

2.3.3 La programmation à base de tâches

Dans le cas de la programmation à base de tâches, les programmeurs expriment le parallélisme de manière flexible en utilisant des abstractions appelées "tâches". A l'inverse des codes classiques, les programmes à base de tâches contiennent des structures dynamiques telles que des boucles *while* et des appels de fonction récursifs qui ne conviennent pas à la parallélisation à l'aide des constructions de parallélisme régulières et rigides.

Du point de vue du modèle de programmation, une tâche est un bloc de travail : c'est une section de code avec les données nécessaires à son exécution. La programmation à base de tâches consiste donc à créer dynamiquement des tâches à accomplir par les cœurs et qui sont gérés par un ordonnanceur de tâches dont le rôle est d'attribuer les tâches aux unités de calcul qui peuvent les accomplir. Introduite dans le standard OpenMP à partir de la version 3.0 [12] (en 2008 sans dépendances entre les tâches), ce type de programmation repose donc sur une vue abstraite des tâches et des données qui permet non seulement une instanciation efficace de diverses données et opérations d'un programme, mais permet également de concevoir et mettre en œuvre des algorithmes pour l'ordonnancement des tâches. La version 4.0 a quant à elle ajouté la capacité à décrire des dépendances entre tâches en fonction des dépendances de données.

Cependant, les runtimes basés sur les tâches existent depuis longtemps. Ils ont été popularisés en 1998 par Cilk [13] et par la première preuve théorique de la garantie de performance de l'algorithme de vol de travail. Le modèle Cilk des tâches indépendantes sur des machines à mémoire partagée a été étendu dans plusieurs directions : la capacité à définir une synchronisation point à point entre les tâches dans les modèles de tâches avec dépendances (comme Athapascan [14]), à s'exécuter sur des architectures hétérogènes avec des accélérateurs (comme les travaux de Hermann et al. [15], StarPU [16], StarSs [17], XKaapi [18]) ou des systèmes de mémoire distribuée (comme StarPU [19], StarSs [20], X10 [21], Kaapi [22], Athapascan [14]).

L'ordonnancement est donc un enjeu majeur de la programmation à base de tâches puisque les informations sur les tâches et leurs données peuvent être utilisées par les runtimes pour hiérarchiser l'exécution de certaines tâches par rapport à d'autres, choisir la bonne ressource pour exécuter des tâches, ou encore optimiser la planification et l'exécution des tâches afin d'obtenir de meilleures performances. C'est donc sur les questions de localité des données dans les caches (ou domaines NUMA) que l'ordonnancement doit apporter des améliorations.

```
     for (k = 0; k < A.nt; k++) {
        tempkm = k == A.nt-1 ? A.n-k*A.nb : A.nb;
        ldak = BLKLDD(A, k);
        double *dA = A(k, k);

#pragma omp task depend(inout:dA[0:A.mb*A.mb])
        LAPACKE_dpotrf_work(LAPACK_COL_MAJOR, lapack_const(PlasmaUpper),
                            tempkm, dA, ldak);

        for (m = k+1; m < A.nt; m++) {
           tempmm = m == A.nt-1 ? A.n-m*A.nb : A.nb;
           double *dA = A(k, k);
           double *dB = A(k, m);

#pragma omp task depend(in:dA[0:A.mb*A.mb]) depend(inout:dB[0:A.mb*A.mb])
           cblas_dtrsm(CblasColMajor,
              (CBLAS_SIDE)PlasmaLeft, (CBLAS_UPLO)PlasmaUpper,
              (CBLAS_TRANSPOSE)PlasmaTrans, (CBLAS_DIAG)PlasmaNonUnit,
              A.mb, tempmm, zone, dA, ldak, dB, ldak);
        }

        for (m = k+1; m < A.nt; m++) {
           tempmm = m == A.nt-1 ? A.n-m*A.nb : A.nb;
           ldam = BLKLDD(A, m);
           double *dA = A(k, m);
           double *dB = A(m, m);

#pragma omp task depend(in:dA[0:A.mb*A.mb]) depend(inout:dB[0:A.mb*A.mb])
           cblas_dsyrk(CblasColMajor,
                 (CBLAS_UPLO)PlasmaUpper, (CBLAS_TRANSPOSE)PlasmaTrans,
                 tempmm, A.mb, (-1.0), dA, ldak, (1.0), dB, ldam);

           for (n = k+1; n < m; n++) {
              double *dA = A(k , n);
              double *dB = A(k , m);
              double *dC = A(n , m);

#pragma omp task depend(in:dA[0:A.mb*A.mb], dB[0:A.mb*A.mb]) \
                 depend(inout:dC[0:A.mb*A.mb])
              cblas_dgemm(CblasColMajor, (CBLAS_TRANSPOSE)PlasmaTrans,
              (CBLAS_TRANSPOSE)PlasmaNoTrans, A.mb, tempmm, A.mb,
              mzone, dA, ldak, dB, ldak, zone, dC, A.mb);
           }

        }
     }
```

FIGURE 2.4 : Algorithme de factorisation Cholesky de la suite Kastors en langage C, les dépendances entre les tâches sont déduites des dépendances de données exprimées avec la clause *depend*

2.4 Motivations

On voit bien que la recherche en ordonnancement est un élément clé dans le paradigme de programmation par tâches pour exécuter efficacement une application sur des ressources données, à cause notamment des aspects NUMA et de cache. Ainsi donc, pour pouvoir comparer les implémentations et quantifier précisément les améliorations apportées par ces recherches, on a besoin de faire des expériences reproductibles.

Cependant, bien qu'il puisse sembler évident que l'exécution d'expérimentations avec des entrées identiques aboutirait à des sorties identiques, ce n'est souvent pas vrai. Les résultats calculés peuvent varier entre les exécutions de la même analyse, surtout sur les processeurs modernes. Plusieurs facteurs entrent en jeu, comme la manière dont les processeurs individuels coopèrent dans un système multicœurs (ou distribué). Par ailleurs, les caractéristiques NUMA des architectures modernes discutées dans les sections 2.1 et 2.2 compromettent la capacité d'atteindre la reproductibilité. En plus de la hiérarchie mémoire qui peut induire des phénomènes de contention et/ou de concurrence (résultants sur des fluctuations de latence et de bande passante), les améliorations apportées au fonctionnement même des cœurs induisent une variabilité importante (les instructions de multiplication addition fusionnées (*FMA*) par exemple ne peuvent pas préserver l'ordre des opérations). De plus, l'environnement dans lequel les expérimentations sont menées est fragile et éphémère à l'échelle de quelques mois jusqu'à plusieurs années, car les compilateurs, les bibliothèques et les systèmes d'exploitation sont continuellement mis à jour, de sorte que revisiter une étude vieille de 10 ans nécessiterait le recours à un musée de super-ordinateurs. Par exemple, une mise à jour de la bibliothèque MKL peut optimiser certaines fonctions, ce qui peut potentiellement changer les résultats. Par ailleurs, ces résultats peuvent aussi être affectés par les paramètres de la machine, comme pour les effets du *frequency governor* discutés plutard dans ce manuscrit. Parfois même, des changements de performances peuvent être inexpliqués : il nous est arrivé de mesurer 1200 GFlop/s pour une expérimentation donnée. Le relancement (le lendemain même) a résulté sur une performance d'environ 850 GFlop/s seulement alors que les paramètres de la machine et les versions des logiciels utilisés étaient identiques. Il est cependant normal que ces variations arrivent, et ce n'est pas un défaut de méthodologie : ce sont les piles logicielles qui sont devenues très/trop complexes. Il y a encore des détails de configuration de la machine qui nous échappent probablement car encore peu connus.

D'un autre côté, les expérimentations requièrent parfois la réservation de la machine cible pour une longue période, ce qui peut être prohibé, ou même, la plateforme peut juste ne pas être disponible. De plus, l'utilisation d'un CPU différent, même si les modèles sont identiques, ne garantit pas l'obtention de résultats de performance cohérents : c'est même extrêmement difficile d'aboutir sur les mêmes valeurs, et ce pour plusieurs raisons sur lesquelles un chercheur n'a pas toujours le contrôle, comme la position du CPU dans le *rack* et sa proximité avec les disques durs, ou encore la dissipation thermique des nœuds voisins.

Une solution donc à ces problèmes est l'utilisation de la simulation, pour -dans notre cas- offrir un environnement **reproductible** pour l'étude des stratégies d'ordonnancement des applications à base de tâches. Cette approche permettrait aux

chercheurs non seulement de faire abstraction des préoccupations concernant la reproductibilité, mais aussi d'expérimenter leurs travaux sur un grand nombre d'architectures et de modèles pour s'assurer que les heuristiques développées sont valables pour toutes les plateformes et non pas spécifiques à une machine en particulier. Les travaux de Vérité et al. (non encore publiés) présentent par exemple des courbes de performances allant jusqu'à 30 GPUs, ce qui n'existe pas du tout en pratique, mais qui a permis de tirer des conclusions sur le comportement de son ordonnanceur. Les expérimentations peuvent par ailleurs être beaucoup **plus rapides** grâce à la simulation, puisque les instructions arithmétiques peuvent être remplacées directement par le temps qu'elles prennent, réduisant sensiblement le temps de simulation par rapport au temps d'exécution total réel de l'application.

De surcroît et puisqu'elle peut être déterministe, la simulation permettrait aussi de débugguer plus facilement pour **toujours** trouver l'origine du problème avec des *breakpoints*, afficher des résultats intermédiaires, étudier voire même faire des statistiques sur le comportement du programme, et relancer la simulation autant de fois que nécessaire jusqu'à comprendre l'origine du bug.

2.5 État de l'art

2.5.1 Les propriétés d'un simulateur

Différents travaux ont été effectués pour répondre aux besoins en simulation exprimés par les chercheurs. La nature des outils développés se divise en deux catégories :

- les simulateurs *cycle-accurate* fournissent une description de l'état de l'exécution à chaque cycle de l'horloge du CPU (exécution au cycle près). Ils sont donc en conséquence souvent coûteux en termes de temps de simulation ;
- les simulateurs *non cycle-accurate* se contentent à l'inverse de ne considérer que le temps pris par un bloc de code, nécessitant donc une modélisation. Ils sont donc moins coûteux en termes de temps de simulation ;

Ces outils peuvent simuler différents paradigmes de programmation, comme les simulateurs pour les systèmes distribués (MPI), les simulateurs des boucles *for* ou encore les simulateurs d'applications à base de tâches (avec ou sans dépendances de données) pour les systèmes à mémoire partagée. Il peuvent aussi prendre en compte d'autres éléments comme les communications réseaux, la consommation d'énergie ou les effets NUMA avec à chaque fois la possibilité d'être *cycle-accurate*, et donc simuler chaque paquet de données, chaque quantité d'énergie consommée par seconde ou chaque ligne de cache transférée.

2.5.2 Les simulateurs existants

De nombreux simulateurs ont été conçus pour prédire les performances dans divers contextes afin d'analyser le comportement des applications. Plusieurs simulateurs ont été développés pour étudier les performances des applications MPI (calcul distribué) sur des plateformes simulées, tels que BigSim [23], xSim [24], Dimemas [25],

Outil	Paradigmes simulés	Propriétés
BigSim	Charm++, MPI	*extreme scale*
xSim	MPI	*extreme scale*
DIMEMAS	MPI, OMpSs	*cloud*
MERPSYS	MPI	*large scale*, énergie
CloudSim	MPI	*cloud*
Aversa et al.	MPI, OpenMP	architectures SMP
Rico et al.	Threads	architectures à mémoire partagée
Stanisic et al.	Tâches	GPU

TABLE 2.1 : Exemple de quelques outils existants et de leurs propriétés

MERPSYS [26] pour les simulations de performances et de consommation d'énergie, ou encore SimGrid [27].

Tandis que certains outils de simulation sont orientés vers la simulation *cloud* comme CloudSim [28] ou GreenCloud [29], d'autres études sont orientées vers les simulations sur des architectures spécifiques, comme les travaux d'Aversa et al. [30] pour les applications hybrides MPI/OpenMP sur SMP, les simulations d'applications basées sur les tâches sur processeurs multicœurs [31, 32, 33, 34, 35] ou encore la simulation des graphes de tâches sur GPU [7]. Mais comme pour Simany [36], aucun modèle de mémoire particulier n'est implémenté dans ces études. Pour ce dernier par exemple, le temps virtuel est calculé à partir d'informations de synchronisation fournies statiquement par l'utilisateur, mais n'utilise aucun modèle de mémoire ou de cache : les effets de cache sont reproduits par des annotations de synchronisation dans les programmes en supposant toujours que les données demandées ne sont pas dans le cache au début d'un bloc d'exécution, et sont ensuite dans le cache pour le reste. De plus, le module d'interface réseau simule les accès aux données par un *payload* de taille fixe dont le coût de traitement est également fixe.

SimGrid [27], le framework dans lequel s'inscrit cette thèse, est un simulateur reconnu ciblant les architectures à mémoire distribuée. Il est *non cycle-accurate*, et est à mi-chemin entre émulation et simulation : par exemple, un transfert de données est modélisé par un flux (d'un nombre d'octets défini) qui progresse à une certaine vitesse sur les liens parcourus (en fonction des paramètres de ces liens) et n'est donc pas fixe : son coût dépend de la bande passante disponible sur les liens traversés. On ne modélise donc pas chaque octet (ou paquet) transféré. De même pour modéliser les calculs, on n'exécute pas d'instructions arithmétiques : on calcule uniquement le temps que prend l'exécution d'un certain nombre d'opérations flottantes en fonction de la puissance de l'unité de calcul responsable de l'exécution. On fait donc tourner la partie contrôle du vrai code de l'application sans effectuer réellement les calculs et les transferts à l'octet près : ce sont les briques de base de l'exécution qu'on modélise afin de pouvoir simuler à bas coût. Par ailleurs, SimGrid est un outil versatile : même s'il est à l'origine destiné à la modélisation des groupes de machines, son développement s'attache à ne pas modéliser un type d'architectures en particulier, mais de modéliser un système distribué de manière générale. Dans ce travail, on va donc "détourner" son usage pour modéliser ce qui se passe à l'intérieur d'un nœud NUMA.

De nombreux efforts ont été faits pour étudier les performances des applications

basées sur des tâches, que ce soit avec la modélisation des accès NUMA sur de grands nœuds de calcul [37, 38], ou avec des accélérateurs [7]. Certaines études ont une approche similaire à notre travail, que ce soit dans un sens technique, comme l'utilisation des composants de SimGrid pour la simulation de boucles parallèles avec diverses techniques d'ordonnancement des boucles dynamiques [39], ou dans un sens de modélisation, comme simNUMA [40] sur des machines multicœurs (atteignant environ 30% d'erreur de précision pour la simulation d'une factorisation LU) ou HLSMN [41] (qui ne tient pas compte des dépendances entre tâches). Cependant, à notre connaissance, aucun simulateur disponible actuellement ne permet de prédire les performances d'applications basées sur des tâches avec des dépendances de données sur les architectures à mémoire partagée tout en prenant en compte à la fois les effets NUMA et la localité des données.

Ce manuscrit est la continuité des travaux effectués par Philippe Virouleau et présentés dans sa thèse [42], qui porte sur l'amélioration de l'exploitation des architectures NUMA à travers les supports exécutifs. Effectivement, ces travaux avaient présenté le développement d'un simulateur préliminaire avec l'objectif de tester de nouvelles heuristiques d'ordonnancement et de comparer les performances réelles des supports exécutifs par rapport à ce qu'il serait **théoriquement possible d'atteindre**. Cependant, ce prototype ne repose pas sur une description de la topologie de la machine, et les modèles de localité développés implémentent des scénarios spécifiques (meilleur cas, pire cas), ce qui a abouti à des résultats de simulation encourageants mais peu précis pour un grand nombre de cœurs. Ces travaux seront rediscutés dans la section 6.4.

Chapitre 3

Modélisation d'une architecture NUMA

Sommaire

Afin de simuler des applications à base de tâches sur des architectures à mémoire partagée, une modélisation fine et précise des machines cibles doit être réalisée. L'expression de ces plateformes est faite avec l'outil SimGrid [27], qui permet de représenter leurs éléments architecturaux à l'aide d'abstractions déjà implémentées. Donc, dans ce chapitre, nous discuterons de la modélisation d'une machine NUMA à l'aide de ces éléments, mais aussi de la manière avec laquelle nous avons mesuré les valeurs des bandes passantes des liens entre les divers éléments qui composent les machines à mémoire partagée, nécessaire pour établir des modèles de performances tels que présentés dans les chapitres suivants.

3.1 La modélisation de plateformes avec SimGrid

Afin de modéliser une machine réelle, SimGrid [27] fournit les outils qui permettent une description détaillée de chaque élément d'un système distribué, présentés dans la figure 3.1 :

- les ***hosts*** représentent des entités de calcul (comme les cœurs d'un CPU) ;
- les ***links*** représentent les liens qui inter-connectent les composants d'une machine ;

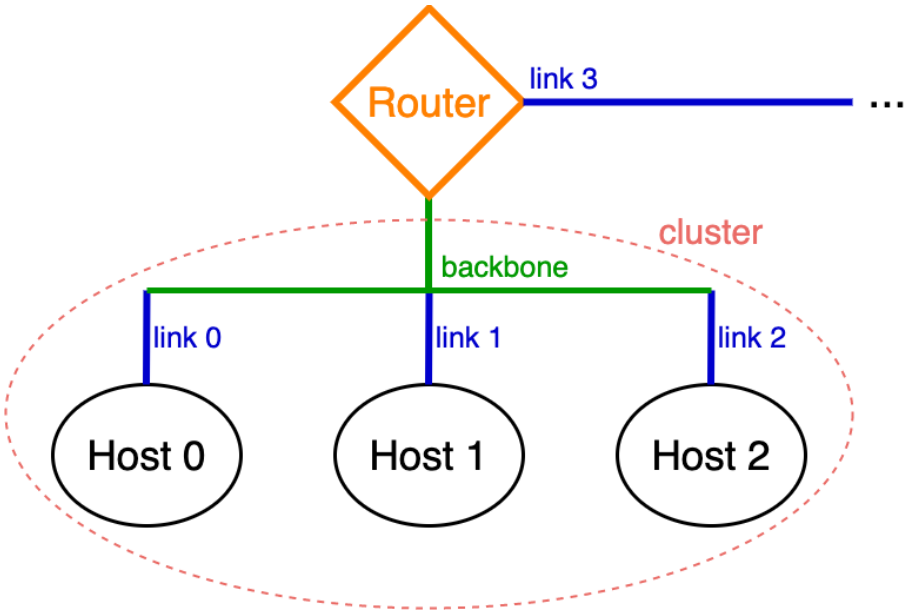


FIGURE 3.1 : Les composants SimGrid qui représentent les éléments constituant un système distribué

- les ***backbones*** sont eux-mêmes des liens mais utilisés pour représenter un commutateur ;
- un ***cluster*** est donc un groupe de *hosts*, *links* et/ou *backbones* ;
- les ***routeurs*** permettent la connexion des *clusters* au monde extérieur ;

De plus, SimGrid permet aussi de définir le routage de la plateforme, c'est à dire le chemin emprunté par les communications entre deux *hosts*, soit explicitement pour imposer un comportement identique à celui d'un système en particulier, ou en utilisant des politiques de routage déjà implémentées telles que l'algorithme Floyd-Warshall pour calculer les chemins.

Il est essentiel de noter que ces éléments de base sont paramétrables : on peut donc définir une puissance théorique pour les *hosts* (en termes de GFlops), mais aussi la latence (en secondes) et la bande passante (en GBps) des *links*/*backbones*, ce qui permet de modéliser la machine au plus près de la réalité.

Par ailleurs, SimGrid permet d'exprimer les différents types de liens qui peuvent interconnecter une plateforme [43], notamment les liens ayant :

- une bande passante globale (appelée *shared*) ;
- une bande passante unilatérale, soit une bande passante dans un seul sens (appelée *splitduplex*) ;
- une bande passante par flux (appelée $fatpipe$).

Dans la réalité, un lien peut avoir plusieurs de ces limitations de bande passante à la fois. Pour modéliser cela avec SimGrid et comme présenté sur la figure 3.2, il suffit de mettre bout à bout un lien de chaque type pour pouvoir paramétrer les différentes caractéristiques de bande passante pour les flux qui les traversent.

FIGURE 3.2 : Modélisation d'un lien ayant différentes limitations de bande passante entre deux composants avec SimGrid

SimGrid inclut aussi des modèles qui décrivent comment la plateforme simulée réagit aux actions de l'application, comme par exemple pour le calcul du temps que prend une communication donnée sur la plateforme simulée. Dans notre cas, on choisit le modèle LV08 [44] (défini par défaut) qui prend en compte les variations de bande passante pendant l'exécution de l'application. Ce dernier est paramétré afin de reproduire le comportement d'une carte mère qui parvient à utiliser très efficacement son réseau d'interconnexion avec les valeurs suivantes :

- *TCP-gamma* = -1
- *bandwidth-factor* = 1
- *latency-factor* = 0
- *weight-S* = 0

Ces paramètres ont pour effet de désactiver les effets de slow-start, d'utilisation très imparfaite de la bande passante, et de changement de comportement en fonction de la taille du message [45], permettant ainsi une modélisation rapide et fiable [46], tout en étant au plus près de la réalité.

Les plateformes simulées sont décrites au format XML, ce qui présente certains inconvénients puisque c'est un langage différent de ceux utilisés traditionnellement pour la calcul haute performances, et qui n'est pas parfois assez expressif pour certains systèmes (en particulier les grandes plateformes présentant des motifs répétitifs). Cependant, ce format spécifique garantit que la modélisation de l'architecture ne soit pas mélangée avec l'application testée puisque dans un fichier séparé, car cela facilite la campagne d'expérimentations par la suite (les mélanger est considéré comme une mauvaise pratique), et permet en plus de générer ces plateformes de façon automatisée (avec un script Python par exemple).

3.2 Modélisation d'une architecture NUMA

On a présenté dans la section précédente SimGrid, qui fournit des outils permettant la modélisation d'une architecture distribuée. Puisque ce travail est orienté vers des applications exécutées sur des architectures à mémoire partagée, nous allons voir comment on adapte ces éléments pour représenter une machine NUMA après avoir présenté les architectures étudiées.

3.2.1 Les architectures étudiées

Les expérimentations présentées dans ce travail ont été effectuées sur PlaFRIM [1], la Plateforme Fédérative pour la Recherche en Informatique et Mathématiques, qui est une plateforme de calcul couvrant des activités de recherche variées telles que la modélisation et la mise au point de codes dans tous les domaines (CFD, électromagnétisme, sismique, HPC, santé, optimisation). Elle fournit aussi les briques logicielles nécessaires au développement de solveurs, runtimes... avec le but de mettre à disposition des moyens de calcul dotés d'architectures nouvelles.

Ainsi donc, deux architectures fournies par PlaFRIM sont étudiées dans cette thèse :

- Intel Xeon Gold 6240 [2] : une machine dual socket de 36 cœurs (18 cœurs par socket), de micro-architecture Cascade-Lake, chaque socket ne comprenant qu'un seul banc NUMA dont les cœurs se partagent 24.75MB de cache L3 ;

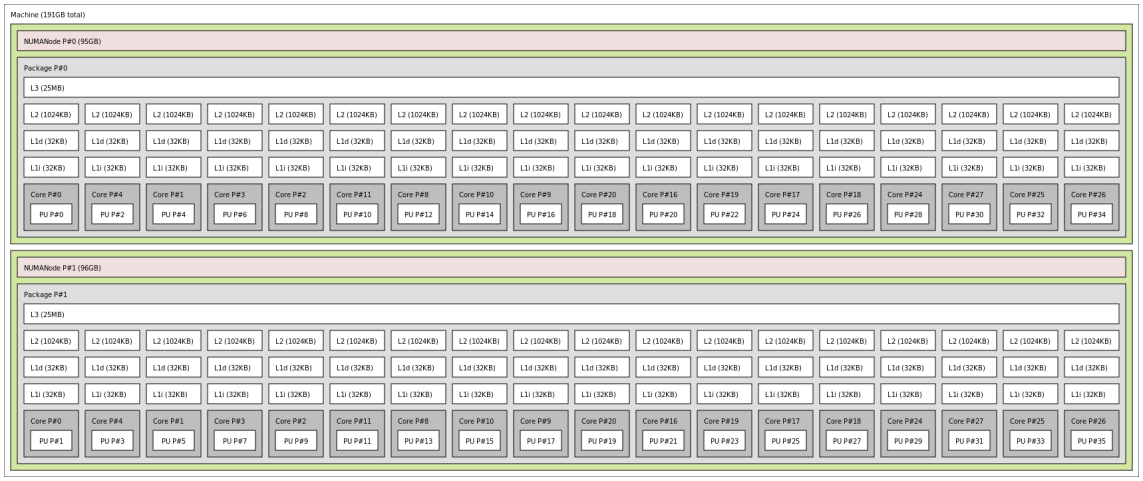


FIGURE 3.3 : Représentation de la machine Intel avec HWLoc

- AMD EPYC 7452 [3] : une machine dual socket de 64 cœurs (32 cœurs par socket), de micro-architecture Zen2, chaque socket contenant 8 bancs NUMA, et chaque banc NUMA comprenant un groupe CCX (4 cœurs qui se partagent un cache L3 de 16MB). Une paire de bancs NUMA constitue une die, et une paire de CCX constitue un CCD.

[1] https://www.plafrim.fr/
[2] https://en.wikichip.org/wiki/intel/xeon_gold/6240
[3] https://en.wikichip.org/wiki/amd/epyc/7452

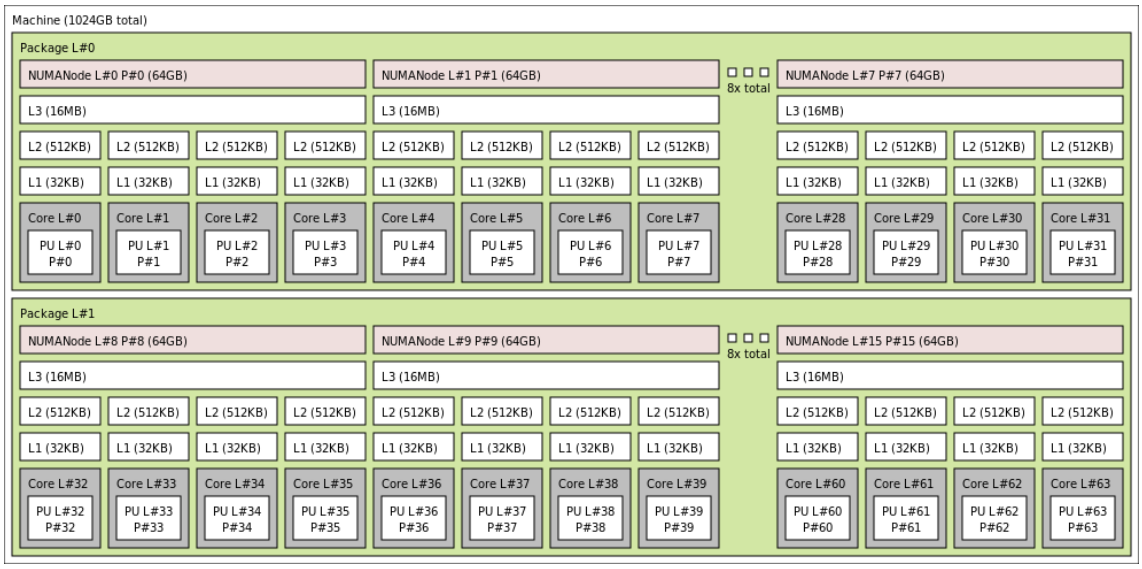


FIGURE 3.4 : Représentation de la machine AMD avec HWLoc

3.2.2 Modélisation d'une architecture NUMA avec SimGrid

Dans ce travail, nous voyons une architecture NUMA comme une machine distribuée : plusieurs unités de calcul appelés "cœurs" ainsi qu'une mémoire cache sont inter-connectés, formant un nœud NUMA. Selon la machine, un ou plusieurs nœuds NUMA également inter-connectés forment une socket qui peut être couplée à une ou plusieurs autres sockets, chacune ayant son propre contrôleur mémoire qui permet l'accès à la mémoire principale (RAM). Les sockets sont à leur tour inter-connectées avec des liens UPI (pour Intel) ou Infinity Fabric (AMD).

Modélisation de l'architecture Intel

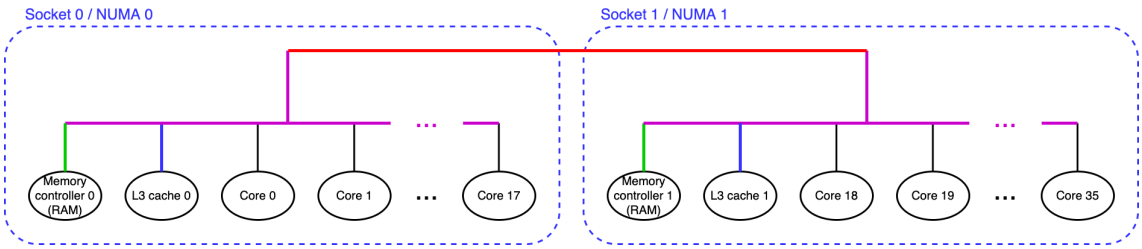


FIGURE 3.5 : Représentation des composants de la machine Intel

Comme explicité dans les figures 3.5 et 3.6, notre représentation d'une architecture NUMA peut être traduite par les abstractions fournies par SimGrid. La figure 3.5 montre les différents composants de l'architecture Intel présentée dans la section 3.2.1, tandis que la figure 3.6 montre comment ces mêmes éléments sont exprimés au sens SimGrid : chacun d'entre eux est représenté par un *host* (*host* 0, *host* 1, *host* 2... pour les cœurs ; $C0$, $C1$ pour les caches et $M0$, $M1$ pour les mémoires principales). Les différentes couleurs des liens entre les composants de l'architecture signifient des valeurs de bande passante différentes. On notera que la présence de routeurs

permet uniquement de regrouper les domaines NUMA mais n'influe aucunement sur la modélisation de la machine, puisqu'ils permettent une continuation des liens intra-nœud.

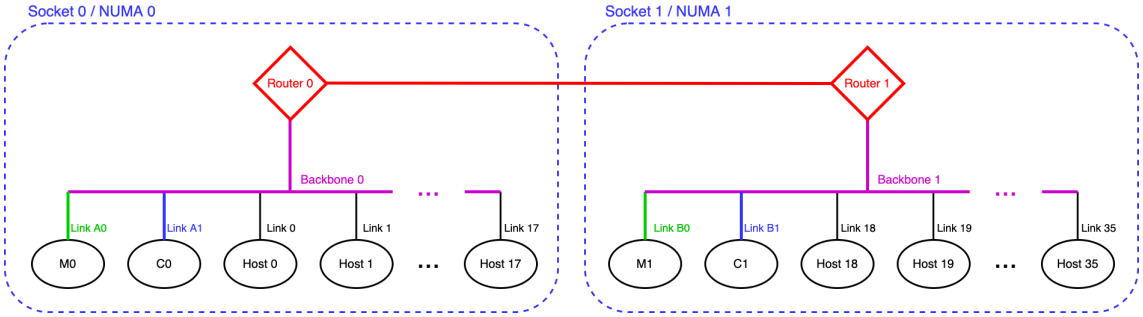


FIGURE 3.6 : Modélisation des composants de l'architecture Intel avec SimGrid

Modélisation de l'architecture AMD

Tandis qu'un seul niveau de hiérarchie est nécessaire pour modéliser la machine Intel, l'architecture AMD est beaucoup plus complexe : comme précisé dans la section 3.2.1, basée sur la micro-architecture Zen2 et présentée sur la figure 3.7, elle comprend 2 sockets composées chacune de 4 dies connectées via un réseau Infinity Fabric. Chaque die contient alors 2 nœuds NUMA, qui à leur tour contiennent 4 cœurs se partageant le même cache L3.

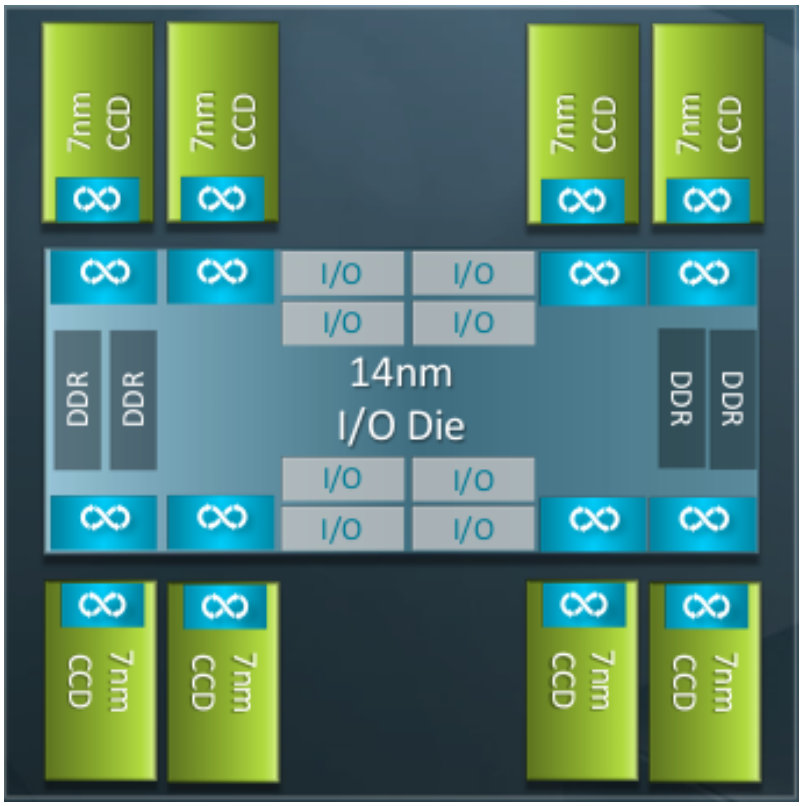


FIGURE 3.7 : Vue de haut niveau des 2 sockets d'un processeur AMD 7002 Series de micro-architecture Zen2 [47]

La figure 3.8 montre notre modélisation de l'architecture AMD, qui est beaucoup plus complexe que la plateforme Intel :

- le réseau Infinity Fabric entre les dies est modélisé par un réseau de liens (en rouge) suivant la topologie AMD ;
- chaque die est ensuite modélisée comme indiqué en haut à gauche de la figure 3.8 : un *backbone* (en violet) représente l'interconnexion Infinity Fabric intégrée entre le réseau die-to-die, la RAM et deux CCX ;
- chaque CCX embarque son propre *backbone* (en vert) qui a une vitesse supérieure à celle de l'interconnexion intégrée.

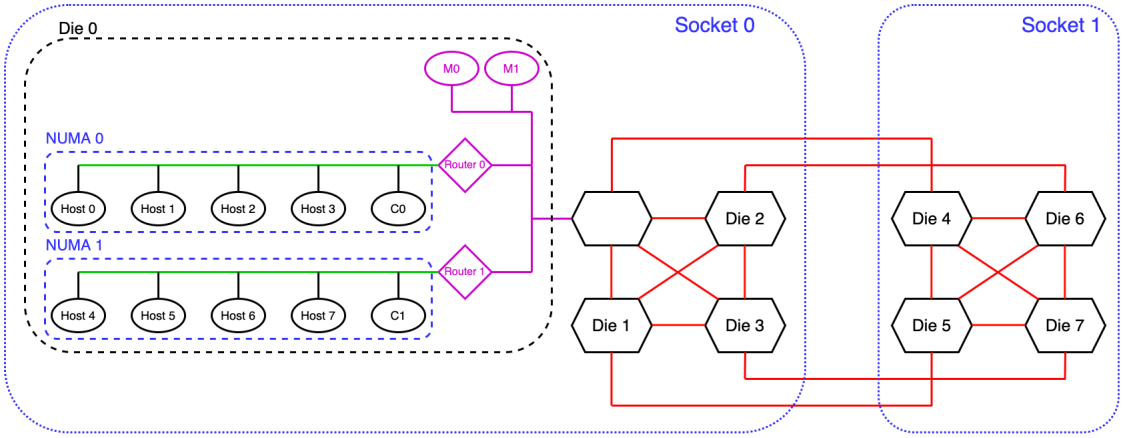


FIGURE 3.8 : Modélisation des composants de l'architecture AMD avec SimGrid : les $Host$ 0, $Host$ 1, $Host$ 2... représentent des cœurs, les éléments $C0$, $C1$ représentent des caches et les éléments $M0$ et $M1$ représentent la mémoire principale (RAM)

On voit donc qu'on se retrouve avec une modélisation très proche de l'architecture Zen2, et ce niveau de détail est nécessaire pour bien prendre en compte que les cœurs d'un CCX ont un accès haut débit au cache L3 correspondant, un accès plus lent aux caches L3 des autres dies du même socket, et un accès encore plus lent aux caches L3 des dies du second socket. Cette modélisation permet aussi de bien prendre en compte les conflits de bande passante sur les liens partagés par les différentes unités de calcul (*hosts*). Il est important de noter que l'on peut être tenté de ne modéliser qu'un seul niveau de liens de communication pour l'architecture AMD. Cependant, les résultats que nous montrerons dans le Chapitre 6 prouvent qu'un respect de la hiérarchie architecturale de la machine est essentiel pour obtenir une bonne précision.

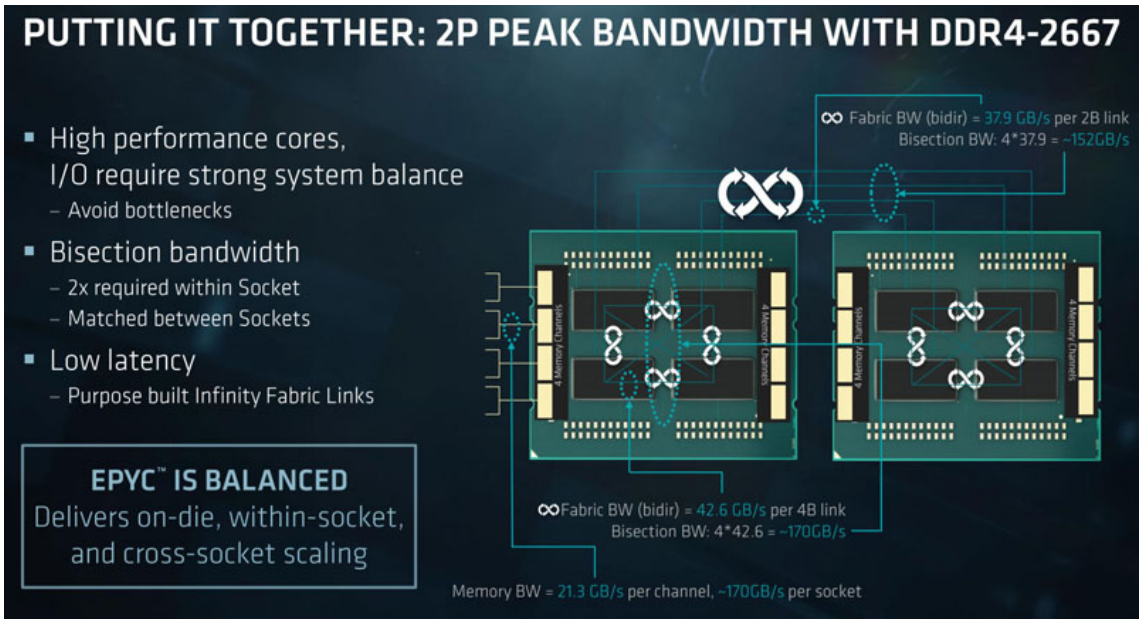


FIGURE 3.9 : *Flyer* marketing d'AMD présentant les bandes passantes des liens *Infinity Fabric* sur les architectures EPYC [48]

Pour compléter notre modèle et comme explicité précédemment, nous devons fournir les valeurs de bande passante aux composants SimGrid de notre plateforme (*links, backbones*) afin de modéliser une vraie exécution. Tandis que certaines valeurs sont directement fournies par la documentation du constructeur, comme sur la figure 3.9 qui est une présentation "marketing" d'AMD pour célébrer les débits des liens *Infinity Fabric* intégrés dans les processeurs EPYC récents, elles sont parfois beaucoup trop optimistes et pas représentatives de la bande passante atteignable dans une exécution réelle, alors que d'autres ne sont pas disponibles du tout : on a choisi de développer notre propre benchmark pour mesurer les bandes passantes manquantes et confirmer (ou infirmer) les données fournies par les fabricants.

3.3 sOMP-BWB : un benchmark pour mesurer les bandes passantes dans des architectures NUMA

L'obtention des bandes passantes des liens qui inter-connectent les éléments de la plateforme simulée nécessite du benchmarking, afin de pouvoir modéliser au plus près de la réalité les interactions entre ses différents composants.

3.3.1 Principe

Bien entendu, nous avons d'abord commencé par considérer les outils existants comme STREAM [49] ou encore Intel *Memory Latency Checker* (Intel MLC [4]), qui fournit la bande passante des contrôleurs mémoire et des liens inter-sockets.

[4] `https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html`

Les paramètres d'Intel MLC par exemple permettent de réduire la taille du tampon afin de conserver les données dans le cache L3, mesurant ainsi la bande passante disponible entre les cœurs et le cache L3 partagé, c'est-à-dire la bande passante de l'interconnexion intra-CCX (en vert dans la figure 3.8). Avec un tampon plus grand, on mesure la bande passante disponible entre les cœurs et la RAM, mais comme le goulot d'étranglement est au niveau de la RAM, c'est la vitesse de la mémoire principale que l'on mesure, et non la vitesse du réseau violet.

Cet outil ne permet donc pas de mesurer la bande passante de tous les liens de la topologie (le *backbone* en violet, les interconnexions entre dies en rouge...), ce qui nécessite le développement de notre propre outil appelé sOMP-BWB.

Ainsi donc, notre outil utilise le principe de lecteurs/écrivains similaire au benchmark likwid [50], et qui permet de mesurer la bande passante atteinte par les transferts entre caches L3. Plus précisément, on utilise pThreads pour créer un écrivain (ou lecteur) par thread, qui sont épinglés aux CPUs voulus avec HWLoc (thread binding) [51]. L'écrivain commence alors à écrire des données ; lorsque celui-ci a terminé, un mécanisme de sémaphores permet de prévenir le lecteur qui commencera à rapatrier les données écrites vers son cache local (si celui-ci est différent de celui de l'écrivain). Une fois que le lecteur a terminé, l'écrivain est prévenu et se remet à écrire des données afin d'invalider celles que le lecteur a lu, lecteur qui se retrouvera du coup obligé de retransférer les données de nouveau.

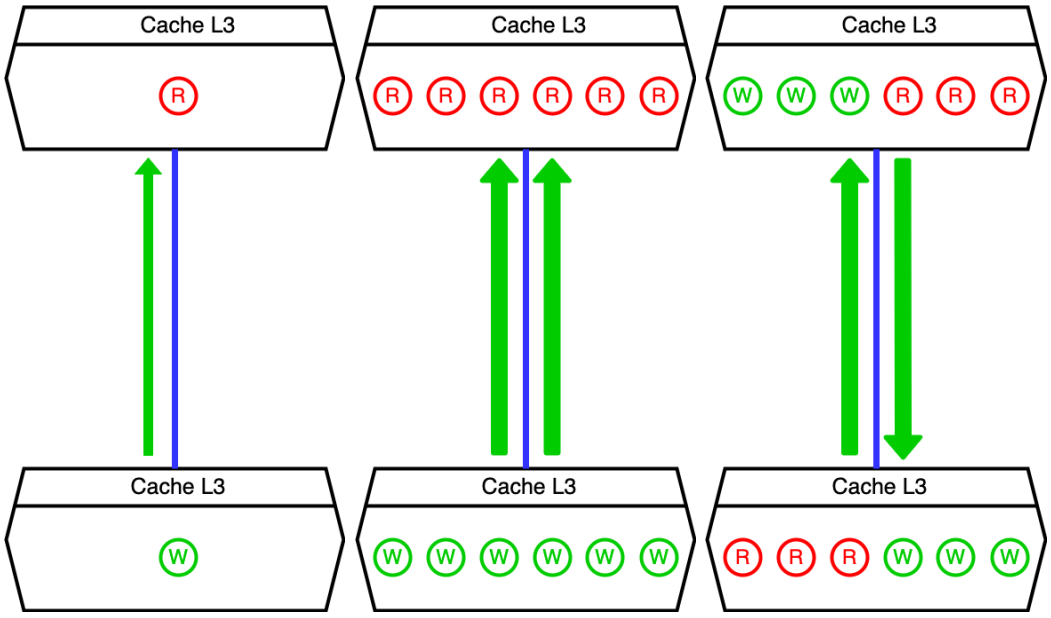
```
for (int j = 0; j < ITER; j++){
   canRead.wait();
   pthread_barrier_wait (&barrier);

   // Mesure du temps
   auto begin = std::chrono::steady_clock::now();
   for (int i = 0; i < ARRAY_SIZE; i++){
      tabR[i] = tabW[i];
   }
   __sync_synchronize();
   auto end = std::chrono::steady_clock::now();

   canWrite.notify();
}
```

FIGURE 3.10 : Extrait du code source de sOMP-BWB qui montre la mesure du temps du lecteur : *tabR* est le vecteur vide du lecteur qu'il doit remplir en lisant les éléments du vecteur *tabW* rempli par l'écrivain

On **mesure** alors **le temps de lecture** comme explicité dans la figure 3.10, qui est le temps nécessaire pour transférer les données du cache distant vers le cache local. Ce processus est répété au moins 1000 fois ($ITER$ dans 3.10) pour atténuer les bruits pouvant venir de l'OS ou de l'architecture. En définissant au préalable la quantité de données à écrire/lire par chaque paire ($ARRAY\_SIZE$ dans 3.10), on peut donc déduire la bande passante des liens traversés en GBps.

Par ailleurs, notre benchmark permet aussi de mesurer directement les caractéristiques des liens de la machine discutés dans la section 3.1 et présentés dans la figure 3.11 :

- la bande passante par flux (*fatpipe*) 3.11a : en mesurant la bande passante atteinte par une seule paire lecteur-écrivain.
- la bande passante unilatérale du lien (*splitduplex*) 3.11b : en utilisant de plus en plus de paires lecteurs-écrivains, avec le groupe de lecteurs proche d'un cache L3 différent du cache L3 à proximité du groupe d'écrivains, les bandes passantes agrégées fournissent alors la bande passante unilatérale ;
- la bande passante globale du lien (*shared*) 3.11c : en utilisant de plus en plus de paires lecteurs-écrivains, avec la moitié du groupe de lecteurs-écrivains proche d'un cache L3 et l'autre moitié proche d'un autre cache L3, on obtient alors la bande passante partagée puisque les flux prennent des sens inverses sur les mêmes liens ;



(a) Fatpipe (b) Splitduplex (c) Shared

FIGURE 3.11 : Représentation des caractéristiques de partage de bande passante des liens, les "R" en rouge représentent les lecteurs, et les "W" en vert représentent les écrivains

3.3.2 Quelques exemples de mesures

Mesures *fatpipe* (figure 3.11a)

La figure 3.12 montre les résultats des mesures de la bande passante sur la machine AMD en fonction de la quantité de données utilisée, avec une seule paire lecteur-écrivain : on place toujours le lecteur sur le CPU0 (donc sur la die 0) et on fait varier l'emplacement de l'écrivain sur tous les cœurs de chaque socket. A gauche,

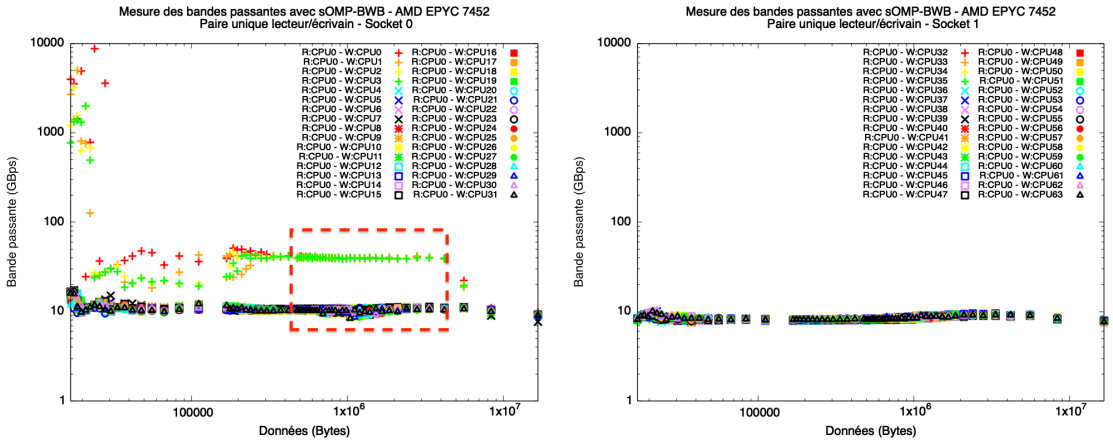


FIGURE 3.12 : Mesure de bande passante avec une seule paire lecteur-écrivain sur la machine AMD avec l'écrivain placé successivement sur les cœurs du socket 0 (à gauche) et du socket 1 (à droite), tandis que le lecteur reste toujours sur le CPU0

on voit que différentes bandes passantes sont observées sur le socket 0 en fonction de la localité de l'écrivain : les valeurs sont plus élevées (en moyenne 40GBps) lorsque l'écrivain est placé sur les CPU0 à CPU3, qui appartiennent à la même die que le lecteur (die 0 en l'occurrence, le lien partagé est en vert dans la figure 3.8), tandis que toutes les autres configurations affichent une même bande passante différente (environ 10GBps) : la baisse à partir du CPU4 traduit donc la limitation des liens inter-die (en violet dans la figure 3.8). Cependant, lorsque l'écrivain est placé sur les CPU32 à CPU63, donc dans le socket 1 (figure 3.12 à droite), on observe une légère baisse de bande passante comparé au cas précédent (en moyenne 8GBps) : la vitesse de lecture est donc limitée par les liens inter-socket (en rouge dans la figure 3.8).

Par ailleurs, dans le cas où les paires lecteur-écrivain sont sur la die 0 (figure 3.12 à gauche), on observe que la quantité de données utilisée influe aussi sur la bande passante : si les données tiennent dans le cache L1 (ou L2), le lecteur n'aura besoin d'accéder qu'au cache L1 (ou L2) adjacent (ou partagé dans le cas L2), ce qui explique les bandes passantes très élevées pour des petites quantités de données. Le cadre en rouge indique quant à lui le plateau des valeurs de bandes passantes correspondantes aux tailles des données qui tiennent dans le cache L3, et ce sont ces valeurs là qu'on considèrera pour notre modélisation de la plateforme.

Les mêmes observations peuvent être constatées sur les mesures pour la machine Intel figure 3.13 : on a plus de bande passante lorsque le CPU0 accède aux données de ses propres caches L1 et L2 (jusqu'à 30GBps). Pour les autres CPU du même socket où on doit passer par le cache L3 partagé, on voit qu'on atteint une bande passante inférieure d'environ 12GBps, représentée par le réseau violet de la figure 3.6. Pour les cœurs du second socket, les flux traversent les liens inter-socket représentés par le réseau rouge de la figure 3.6, avec une bande passante encore plus réduite (environ 7GBps).

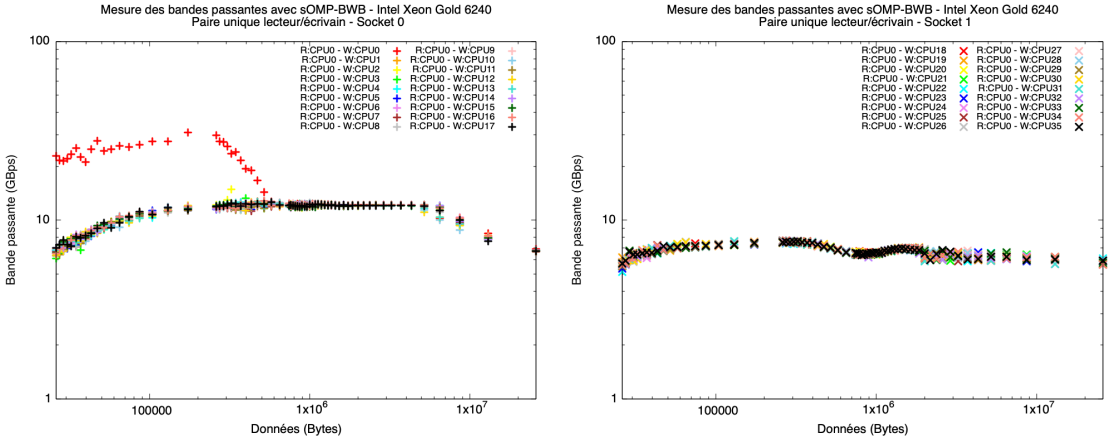


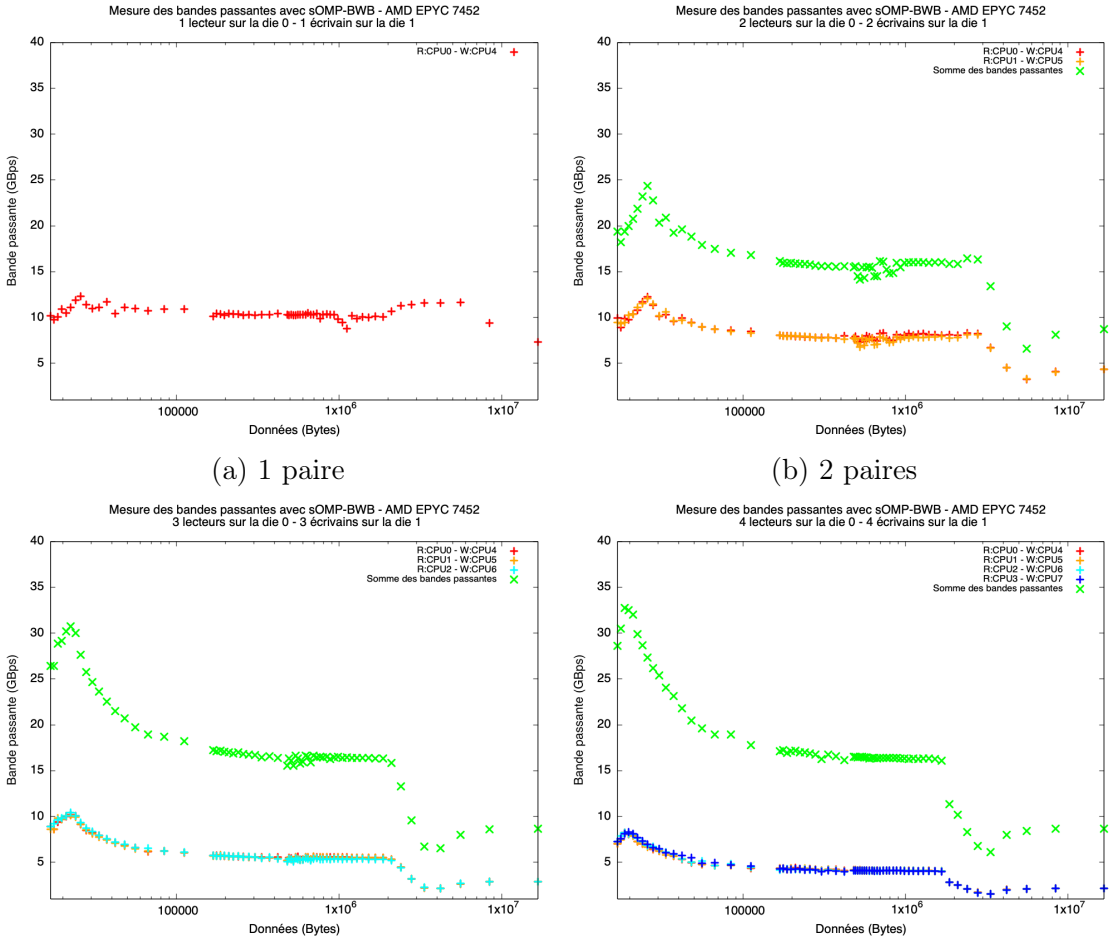
FIGURE 3.13 : Mesure de bande passante avec une seule paire lecteur-écrivain sur la machine Intel avec l'écrivain placé successivement sur les cœurs du socket 0 (à gauche) et du socket 1 (à droite), tandis que le lecteur reste toujours sur le CPU0

Mesures *splitduplex* (figure 3.11b)

Comme explicité dans la section précédente, ces résultats de bande passante atteinte par une paire unique lecteur-écrivain nous permet d'obtenir la valeur de bande passante par flux des liens traversés ($fatpipe$). Cependant, on peut tester d'autres scénarios pour observer les caractéristiques des liens de la plateforme. Par exemple, pour obtenir la bande passante unilatérale du lien entre les dies 0 et 1 pour l'architecture AMD, on peut placer un lecteur sur la die 0 et un écrivain sur la die 1. Ensuite, en augmentant le nombre de lecteurs et d'écrivains dans chaque die (en utilisant plusieurs cœurs sur chaque die), on peut calculer la somme des bandes passantes : c'est la bande passante unilatérale du lien entre les dies 0 et 1. La figure 3.14 présente ce cas : on voit alors qu'en rajoutant des paires lecteur-écrivain, on arrive à un maximum (atteint lorsqu'on teste avec 3 paires, figure 3.14c) de 16GBps de bande passante cumulée, ce qui nous fournit donc la bande passante *splitduplex* du lien entre les dies 0 et 1. De plus, dans le cas à 2 paires 3.14b, on voit bien que les paires obtiennent les mêmes débits (en moyenne 7.5GBps chacun). Lorsqu'on plafonne dans les 2 cas suivants, les paires voient leur débit limité (environ 5GBps pour chacunes des 3 paires 3.14c puis 4GBps pour chacunes des 4 3.14d), alors que la somme des bandes passantes atteinte reste la même (16GBps). Ceci prouve que le partage des bandes passantes se passe de façon optimale, ce qui est rassurant quant au fait de modéliser ça simplement avec les liens SimGrid.

Mesures *shared* (figure 3.11c)

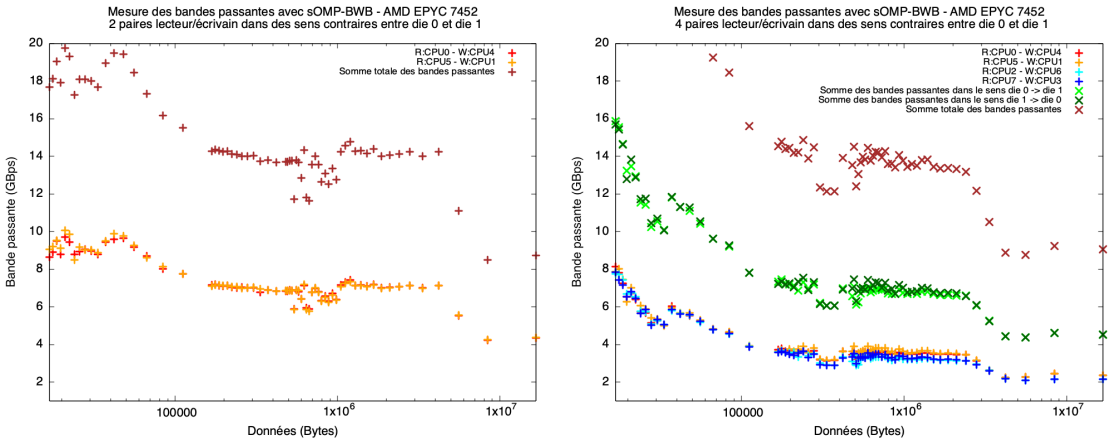
Alors qu'on a placé tous les lecteurs (et tous les écrivains) sur la même die dans les tests précédents 3.11a 3.11b, on peut cette fois-ci croiser les paires, comme présenté dans la figure 3.11c : en épinglant 2 paires (figure 3.15a) lecteur-écrivain dans des sens différents (un lecteur sur la die 0 et son écrivain sur la die 1 pour la première paire, et un lecteur sur la die 1 et son écrivain sur la die 0 pour la deuxième, à multiplier par 2 dans le cas 4 paires) en même temps, on peut observer la bande passante globale du même lien entre les dies 0 et 1, puisque les flux de données prennent cette



(a) 1 paire (b) 2 paires

(c) 3 paires (d) 4 paires

FIGURE 3.14 : Mesure de bande passante avec plusieurs paires lecteur-écrivain dans le même sens entre les dies 0 et 1



(a) 2 paires (b) 4 paires

FIGURE 3.15 : Mesure de bande passante avec plusieurs paires lecteur-écrivain dans le sens contraire entre les dies 0 et 1

fois-ci des sens contraires. On voit alors qu'une bande passante d'environ 7GBps est toujours atteignable, que se soit pour des flux uniques ou pour leur somme dans chaque direction. La somme totale reste quant à elle identique dans le cas à 4 paires (figure 3.15b) atteignant environ 14GBps : c'est cette valeur de bande passante qu'on indiquera comme bande passante *shared* dans notre plateforme simulée. La documentation AMD 3.9 quant à elle évoque une bande passante de 42.6GBps pour une DDR à 2.667GHz, et puisqu'on est en mode *powersave* du *frequency governor* (1.5GHz), cette valeur correspond à 23.9GBps : on est loin des chiffres annoncés par le marketing d'AMD.

Par ailleurs, on remarque que la bande passante *splitduplex* est supérieure à la bande passante *shared* dans nos mesures. En effet, lorsqu'un transfert de données est effectué d'un cache à un autre, il y a aussi des messages qui sont envoyés dans le sens contraire pour les requêtes d'envoi qui correspondent au protocole du cache : c'est ce qui explique la différence des résultats entre les deux expérimentations.

Commutativité

L'outil sOMP-BWB permet donc d'obtenir toutes les valeurs de bande passante dans une architecture à mémoire partagée. En réitérant les tests effectués pour chaque die, on peut alors s'assurer de la "commutativité" des bandes passantes des liens entre dies, permettant ainsi de fournir les valeurs exactes pour chaque caractéristique de lien des machines à la plateforme modélisée avec SimGrid, comme présenté dans la section 3.2.2.

3.3.3 Discussion

La modélisation des architectures NUMA que nous avons présenté se repose donc sur la modélisation précise de la topologie réelle, et sur des mesures fines des différentes valeurs et caractéristiques des liens qui composent la machine. Cependant, on ne considère que le comportement des LLC en ne prenant en compte que les

valeurs de bande passante où la taille des données échangée tient dans le cache L3. sOMP-BWB pourrait être utilisé pour étudier le comportement de la contention entre les caches L2 des différents cœurs qui partagent un même cache L3, mais il est important de rappeler qu'on ne considérera dans ces travaux que le niveau L3 dans la hiérarchie des caches : on fait ici un compromis entre la précision des simulations et leur coût. On pourrait prendre en considération les caches L2, mais cela impactera notre temps de simulation sans pour autant résulter sur une grande amélioration de la précision, puisque pour mieux profiter des performances de la machines, une application aura tendance à choisir des tailles de blocs à traiter qui tiennent dans le cache L3. De plus, l'obtention des valeurs de bande passante réellement disponible entre les caches L2 est beaucoup plus complexe : dans les mesures présentées dans la section 3.3.2, pour les réseaux rouges et violets de la figure 3.8, on avait plusieurs cœurs à disposition pour atteindre facilement la saturation de bande passante. Si on veut observer les comportements entre caches L2 à l'intérieur du réseau vert par exemple, on n'aurait que 4 cœurs pour faire des paires lecteur-écrivain : il y a alors un grand risque de ne pas atteindre ce maximum aussi facilement.

Par ailleurs, notre modélisation des architectures NUMA devra être différente dans certains cas comme pour les machines à mémoire hétérogène : effectivement, la présence d'un GPU ajoute de la contention au niveau de la RAM (puisqu'il aura aussi besoin des données qui s'y trouvent) impactant ainsi les mesures de bande passante. Il faudrait alors étendre le principe de mesure et étendre la modélisation pour simuler correctement la politique de partage de bande passante entre le transfert RAM/GPU et le transfert RAM/CPU, et faire de même dans le cas de la simulation des cartes réseau. A ce sujet, on peut notamment mentionner les travaux en cours de publication de Philippe Swartvagher qui, dans le cadre de sa thèse, a observé des effets d'interactions non triviaux, avec possiblement une priorisation des transferts RAM/réseau par rapport aux transferts RAM/CPU.

De plus, certains types d'architecture peuvent nécessiter une autre manière de modéliser les interconnexions entre composants. Pour des machines de type Intel Xeon Phi (bien que décédées), une grille d'interconnexion au lieu d'un réseau pourrait être nécessaire pour simuler l'architecture et prendre en compte les différents scénarios de routage (qui peut être dynamique et effectué par le matériel) pouvant intervenir lors de l'exécution d'une application sur ce type de plateformes.

Enfin, dans l'état actuel de nos travaux, on combine des informations venant des constructeurs avec nos mesures concrètes via sOMP-BWB. L'automatisation de toutes les prises de mesures pourrait être envisagée pour ne plus s'appuyer sur la documentation.

Chapitre 4

sOMP : un simulateur pour applications à base de tâches

Sommaire

Dans ce chapitre, on verra comment on utilise une trace d'exécution d'une application pour la simuler. En effet, on peut collecter les données nécessaires pour reconstruire le graphe de tâche d'une application et les utiliser pour prédire son comportement et son temps d'exécution. Pour faire cela, on a donc développé un simulateur à l'aide du framework SimGrid.

4.1 La simulation d'applications avec SimGrid

En plus d'offrir la possibilité de modéliser une plateforme comme présenté dans la section 3.1, SimGrid [27] fournit aussi une interface nommée S4U qui permet de décrire et simuler des algorithmes abstraits dans plusieurs domaines (HPC, Cloud, P2P...). La simulation est typiquement composée de plusieurs acteurs (*actors*) qui exécutent des fonctions définies par l'utilisateur. Pour faire le lien avec les plateformes modélisées avec SimGrid, chaque acteur simule un cœur (*host*) de la plateforme modélisée. Ensemble, les acteurs utilisent explicitement l'interface S4U pour exprimer leurs calculs, leur utilisation du disque, leurs communications ainsi que d'autres activités (*activities*) qui ont donc lieu sur les ressources décrites dans les plateformes simulées.

Par ailleurs, les acteurs peuvent aussi interagir entre eux via des mécanismes de synchronisation comme des *mutex*, des *semaphores*, des barrières (*barrier*) ou encore des conditions variables (*CondVar*). SimGrid s'occupe alors de calculer le temps nécessaire à la réalisation de chaque activité et orchestre les acteurs en conséquence, en attendant la fin de leur activités, en prenant en compte le temps nécessaire à la réalisation de leurs communications qui dépendent des paramètres de la plateforme simulée, et en incluant dans le calcul du temps les synchronisations temporelles induites par les *mutex*.

Dans ce travail, nous allons donc utiliser les possibilités offertes par SimGrid pour concevoir et développer un simulateur pour applications à base de tâches. Cependant, avant de simuler lesdites applications, on doit d'abord commencer par collecter des données sur leur exécution : on a donc besoin de générer leurs *traces*.

4.2 Collecte des traces avec TiKKi

Une partie importante de ce travail repose sur les données récupérées après une exécution **séquentielle** de l'application cible, qui forment ce qu'on appelle une trace. Donc pour obtenir la trace d'une application à base de tâche, on utilise un outil déjà existant nommé TiKKi.

Le runtime libKOMP [52] d'OpenMP embarque des outils de trace basés sur les travaux de de Kergommeaux et al [53] et repris dans Kaapi [22]. L'un de ces outils est TiKKi, qui a été développé par Philippe Virouleau alors ingénieur dans l'équipe AVALON de l'INRIA Grenoble, et qui repose sur OMPT [54], l'API OpenMP pour les outils de performances intégrés dans le standard OpenMP depuis sa révision 5.0 [55]. OMPT permet en effet de développer des outils d'instrumentation orientés vers des méthodologies basées sur les traces.

TiKKi permet la capture de tous les événements nécessaires à la construction du graphe de tâches de l'application et les enregistre dans un fichier. La structure de la trace d'exécution est une séquence de régions parallèles, où les événements de chaque tâche sont enregistrés. Ces évènements sont enrichis avec des informations sur les performances de chaque tâche comme :

- le nom, l'ordre de soumission et la numérotation logique du CPU sur lequel la tâche est exécutée ;
- les dates de création, de début et de fin de chaque tâche ;
- les dépendances entre tâches ;
- les types d'accès mémoire effectués (R, RW...) ;
- la localité [52] et la quantité de données utilisées (avec PAPI [56], une interface pour l'utilisation des compteurs de performances matériels que l'on trouve dans la plupart des microprocesseurs).

Aussi donc, lorsque TiKKi produit une trace, il génère une séquence de fichiers (un par région parallèle). Chacun de ces fichiers peut ensuite être simulé sous la forme d'un graphe de tâches distinct. TiKKi peut générer plusieurs formes de sortie

```
Name: dgemm
JobId: 4307
StartTime: 1617551316135.580078
EndTime: 1617551316149.144043
MemoryNode: 0
Handles: 0x7fa055400000 0x7fa055200000 0x7fa016200000
Modes: RW R R
DependsOn: 4031 4074 4053
```

FIGURE 4.1 : Extrait d'une trace TiKKi pour une tâche de type *gemm*

des traces d'exécution : graphe de tâches sous forme de fichier .dot, diagramme de Gantt sous forme d'un fichier .csv pour un traitement avec R, ou encore le format de fichier spécifique pour les simulations effectuées par notre simulateur au format .rec (GNU Recutils).

Au final, un chercheur en ordonnancement lancera donc son application une seule fois en séquentiel sur la machine cible, afin d'en collecter la trace qu'il pourra convertir au format rec. L'application reste donc inchangée et pourra avec le simulateur changer le nombre de cœurs (pour une simulation d'exécution parallèle), l'ordonnanceur, le placement mémoire, etc.

4.3 Application sur des algorithmes d'algèbre linéaire : la suite Kastors

Ce travail de simulation est orienté vers les applications d'algèbre linéaire à base de tâches. Pour en tester les performances, on va donc utiliser la suite Kastors [57] qui a été conçue pour évaluer l'implémentation du paradigme des tâches dépendantes d'OpenMP, introduit dans le cadre des spécifications OpenMP 4.0.

Le sous-ensemble Plasma [58] de la suite Kastors fournit 3 algorithmes de factorisation matricielle dense en double précision. Les matrices étant découpées en blocs, l'application peut alors être représentée sous la forme d'un graphe de tâches comme présenté sur la figure 4.2, chaque cœur travaillant donc sur quelques blocs de la matrice (tâches) dont on choisit la taille de manière appropriée, afin de profiter au mieux de la taille du cache L3 et sans en déborder. Ces algorithmes sont :

- la factorisation de **Cholesky** consiste, pour une matrice symétrique définie positive A, à déterminer une matrice triangulaire inférieure L telle que $A = L.L^T$. L'algorithme est composé de 4 types de tâches :
  - **gemm** ($\theta(n^3)$),
  - **trsm** ($\theta(n^2)$),
  - **syrk** ($\theta(n^2)$),
  - **potrf** ($\theta(n)$).

Cholesky est donc majoritairement composé de tâches *gemm* qui sont très efficaces et impliquent 3 blocs de matrice. Cet algorithme présente également une quantité non négligeable de réutilisation des données entre les tâches ;

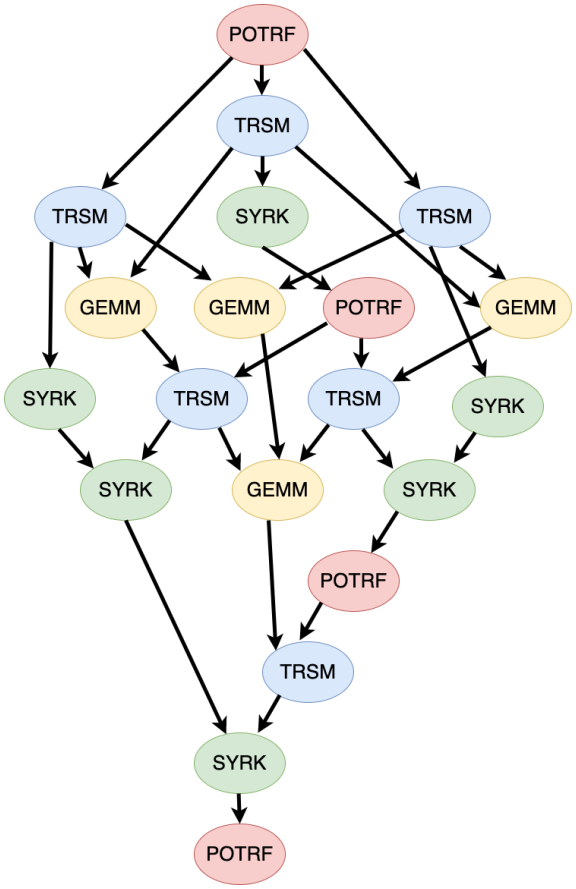


FIGURE 4.2 : Graphe de tâche d'une factorisation Cholesky à 4x4 blocs

- la décomposition **QR** d'une matrice A est une décomposition de la forme $A = Q.R$ où Q est une matrice orthogonale et R est une matrice triangulaire supérieure. L'algorithme est composé de 4 types de tâches :
  - **tsmqr** ($\theta(n^3)$),
  - **ormqr** ($\theta(n^2)$),
  - **tsqrt** ($\theta(n^2)$),
  - **geqrt** ($\theta(n)$).

  Cet algorithme est donc majoritairement composé de tâches *tsmqr*, qui sont nettement moins performantes que les tâches *gemm*, et impliquent 4 blocs de matrice et 1 bloc de type *scratch* (c'est-à-dire un bloc présent en permanence dans le cache local). Il faut noter aussi que l'algorithme QR présente moins de réutilisation des données entre les tâches, ce qui a donc tendance à générer plus d'évictions des caches ;

- la décomposition **LU avec pivot** d'une matrice A est une décomposition de la forme $A = L.U$ où L est une matrice triangulaire inférieure et U est une matrice triangulaire supérieure. L'algorithme est composé de 4 types de tâches :
  - **gemm** ($\theta(n^3)$),
  - **laswp** ($\theta(n^2)$),
  - **swptr** ($\theta(n^2)$),
  - **getrf** ($\theta(n)$).

  Cet algorithme est donc aussi majoritairement composé de tâches *gemm* et présente aussi moins de réutilisation des données entre les tâches. Par ailleurs, le pivotage amène un comportement beaucoup moins régulier que les tâches *gemm* que l'on discutera dans le prochain chapitre.

4.4 Le simulateur sOMP

sOMP [59] est le simulateur que nous avons développé pour les applications parallèles à base de tâches dépendantes qui repose sur SimGrid, afin de permettre la prédiction de leur performances, mais aussi offrir un outil fiable aux chercheurs en ordonnancement de tâches pour concevoir et tester leurs implémentations.

4.4.1 Vue générale

sOMP utilise les traces d'exécution des applications cibles obtenues à l'aide de TiKKi, cependant, les deux sont indépendants : d'autres outils peuvent être utilisés pour générer des traces (comme StarPU). Par ailleurs et comme mentionné précédemment, on choisit de n'utiliser que des traces d'exécutions séquentielles afin d'éviter les bruits et interférences engendrées par l'utilisation d'un grand nombre de cœurs et qui peuvent donc capturer les effets de cache et de contention que l'on veut justement simuler et non rejouer.

Comme explicité dans la figure 4.3, le simulateur sOMP, qui pourrait à terme être un composant SimGrid à part entière, prend en entrée la trace séquentielle de l'application cible, la modélisation de la plateforme sur laquelle l'utilisateur souhaite la simuler, un ordonnanceur de tâches et des modèles de performances qui offrent différentes stratégies d'amélioration de la précision des simulations, avec comme objectif la prédiction des performances de l'application sur n'importe quel nombre de cœurs.

4.4.2 Conception

Le principe du simulateur est comme suit : on commence d'abord par récupérer la plateforme simulée et ses caractéristiques (nombre de cœurs, hiérarchie...). Le simulateur crée alors autant d'acteurs que de cœurs dans la plateforme : chaque acteur sera l'entité responsable des fonctions à exécuter sur chaque cœur simulé (calculs et communications). Ensuite, on récupère les données de chaque tâche qui sont contenues dans la trace de l'application obtenue avec TiKKi (section 4.2). Les

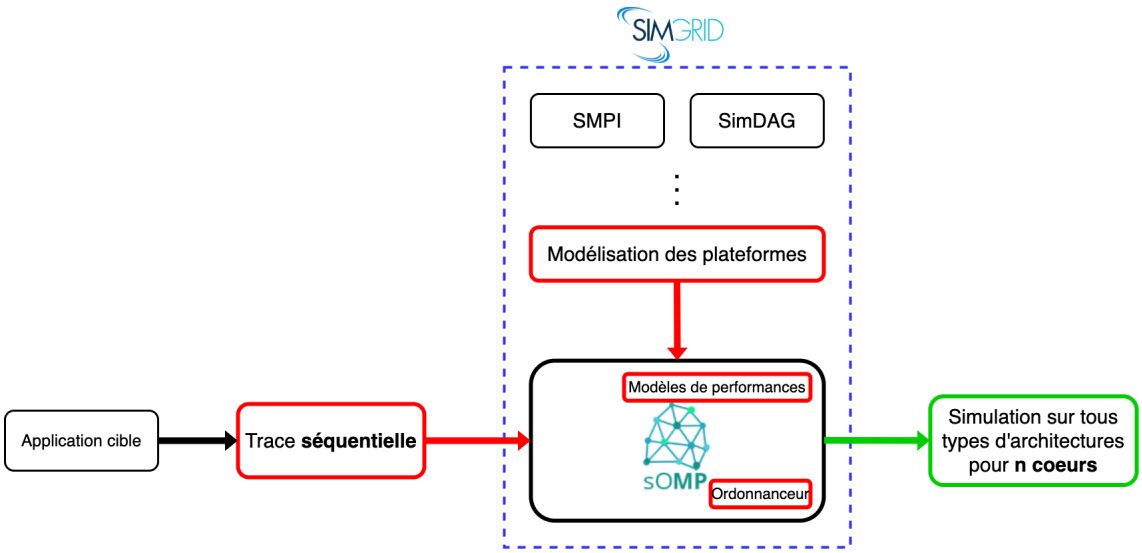


FIGURE 4.3 : Vue générale du projet : en rouge les entrées du simulateur, en vert le résultat souhaité

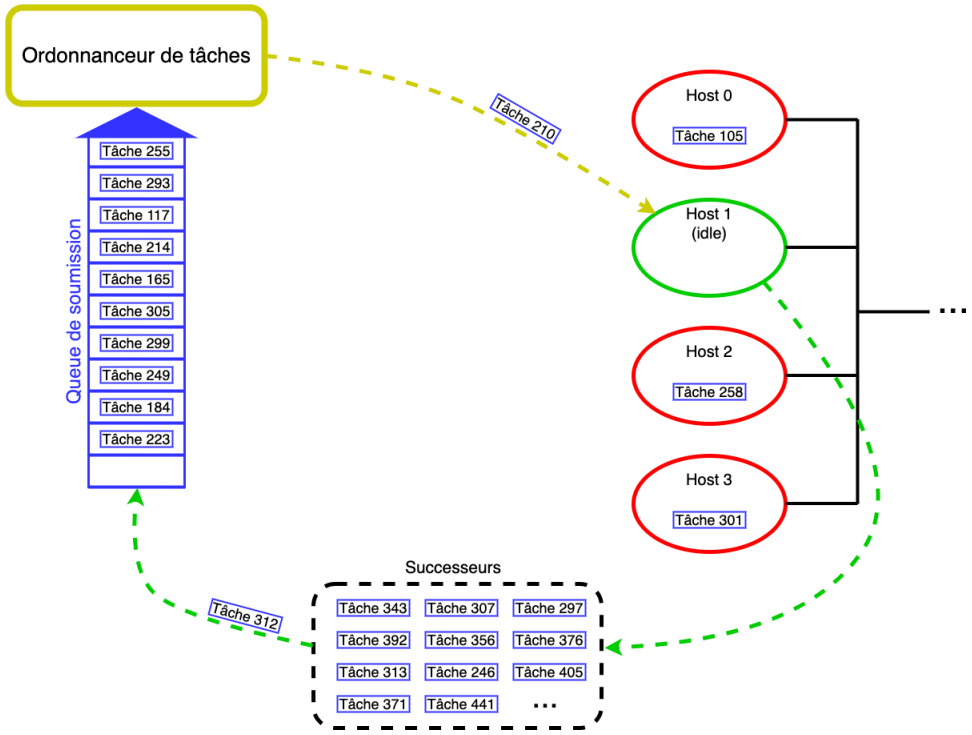


FIGURE 4.4 : Fonctionnement interne du simulateur sOMP : ici, l'ordonnanceur soumet la tâche 210 au Host 1 qui a justement terminé sa tâche précédente et a activé son successeur, la tâche 312, qui prendra du coup une place dans la queue de soumission pour être exécutée à son tour

tâches sont ensuite placées dans une queue de soumission et manipulées par un ordonnanceur de tâches présenté dans la sous-section suivante.

Ainsi donc et comme présenté dans la figure 4.4, sur chaque cœur simulé, un acteur SimGrid (appelé *worker*) reçoit les tâches une par une pour la simulation. Le *worker* simule d'abord les accès mémoire de la tâche : pour chaque accès aux données, il déclenche un message d'une taille correspondante (en octets) à la quantité de données utilisées par la tâche. Ensuite, le *worker* attend la fin du transfert de tous les messages (représentants des accès mémoire), incrémentant ainsi l'horloge interne de SimGrid puisque les communications tiennent compte des paramètres des liens traversés par les messages (bande passante, latence) et de la contention induite par les accès concurrents. Enfin, le *worker* simule l'exécution de la tâche en avançant encore une fois l'horloge interne de SimGrid d'un temps égal au temps réel d'exécution de la tâche, obtenu à partir de la trace. Une fois l'exécution d'une tâche terminée, le *worker* est alors chargé d'activer la soumission de ses successeurs à l'ordonnanceur, si à leur tour toutes leurs dépendances ont été satisfaites.

4.4.3 L'ordonnanceur de tâches implémenté

Pour gérer la soumission des tâches dans sOMP, nous avons implémenté un ordonnanceur (*scheduler*) simple similaire à un runtime OpenMP typique. Comme présenté dans la sous-section précédente, les tâches sont placées dans une queue de soumission de type *FIFO* (*First In First Out*) : notre ordonnanceur s'occupe alors de placer chaque tâche sur une ressource (représentée par un acteur) suivant deux conditions :

- les dépendances de la tâche doivent être satisfaites ;
- la disponibilité de la ressource de calcul.

Comme explicité plus haut, l'objectif du simulateur sOMP est de permettre -entre autres- l'étude de différentes politiques de placement des tâches : nous avons donc conçu notre outil de façon à permettre l'implémentation d'autres ordonnanceurs très facilement.

4.4.4 Discussion

sOMP est donc un simulateur pour applications à base de tâches utilisant des traces d'exécutions séquentielles comme celles fournies par TiKKi. Ce projet est *open-source* (`https://gitlab.inria.fr/idaoudi/omps/-/wikis/home`) et fonctionnel à l'attention des chercheurs en ordonnancement, avec plusieurs améliorations prévues (et discutées dans la section 7.2) comme l'ajout d'une collection d'ordonnanceurs connus pour en tester les effets sur les applications de l'utilisateur. Par ailleurs, différents modèles de performances peuvent être implémentés afin de prendre en considération finement les accès aux données et leurs mouvements, qui est le sujet du chapitre suivant.

Chapitre 5

Modèles de performances

Sommaire

Après avoir discuté de l'implémentation de sOMP, notre simulateur pour applications à base de tâches avec dépendances, on présente dans ce chapitre les modèles de performances qui permettront de simuler les accès mémoire de l'application tracée. On commence donc par présenter un modèle naïf qui ne prend en considération que le temps d'exécution des tâches, qui sera ensuite amélioré par les modèles COMM et COMM+CACHE offrant différents niveaux de modélisation des effets NUMA.

5.1 La simulation du temps d'exécution

5.1.1 Le modèle "TASK"

Au moment de l'exécution (*runtime*), une tâche exécute principalement **des instructions arithmétiques entrelacées avec des accès mémoire** (généralement des instructions de type load/store). $T_C(t_i)$ étant le temps d'exécution des calculs de la tâche fourni par la trace séquentielle collectée avec TiKKi et $Temps(t_i)$ le temps total de la tâche, on considérera dans ce premier modèle "naïf" que :

$$Temps(t_i) = T_C(t_i) \tag{5.1}$$

On ne simulera donc que le temps de calcul des tâches sans aucune considération pour les accès mémoire et la localité des données. Un tel modèle n'est bien adapté que pour les applications compute-bound, c'est-à-dire les applications dont les tâches font peu d'accès à la mémoire, mais représente aussi pour nous un point de repère : c'est un modèle témoin.

5.1.2 Discussion

Bien entendu, une telle modélisation de l'exécution des tâches d'une application parallèle est assez grossière, puisqu'on sait bien que les durées des tâches s'allongent lorsque l'application est exécutée sur plusieurs cœurs en raison de la localité des données [60] : le fait d'utiliser de plus en plus de cœurs implique de plus en plus d'échanges de données entre les caches, qui se retrouvent bridés par la bande passante des liens traversés (les liens inter-sockets sont les plus pénalisants car ils ont la plus faible bande passante de l'architecture) et sont potentiellement sollicités par de nombreux transferts si l'ordonnanceur ne place pas les tâches de manière appropriée.

Comme explicité précédemment, ce modèle servira surtout de point de repère pour se comparer aux autres modèles où les accès aux données seront pris en compte. Il est nécessaire de séparer dans la simulation de l'exécution des tâches, la partie calcul de la partie accès mémoire, afin de pouvoir rejouer la première (ce qu'on fait dans le modèle TASK) et simuler la seconde. La simulation des accès NUMA permettra en effet à sOMP de prendre en compte les effets de localité des données pour être plus fidèle vis-à-vis du fonctionnement de l'architecture, comme présenté dans le Chapitre 3. C'est donc pourquoi on ne rejoue que le coût des calculs en utilisant les temps des tâches issus d'une exécution séquentielle (cela ne rejoue en effet que le temps sans les effets de localité puisque tout se passe sur un seul cœur), la simulation des effets NUMA s'y ajoute ensuite.

Cependant, il est important de noter que d'autres effets entrent en jeu lors de l'exécution d'une tâche : les effets thermiques jouent un rôle important et obligent le *frequecy governor* (qui contrôle la façon dont le processeur augmente et diminue sa fréquence de calcul en réponse aux demandes de l'utilisateur) à ajuster la fréquence en fonction de la température de la machine, qui n'est pas uniquement liée à sa capacité à dissiper la chaleur, mais aussi à d'autres paramètres comme la température de la pièce, la chaleur libérée par d'autres machines dans le rack, etc... Ainsi donc et pour palier ce problème, on choisit de définir la politique du *governor* au mode *powersave*, ce qui maintiendra la fréquence de tous les cœurs utilisés à la valeur minimale, permettant d'écarter d'éventuels effets de fluctuation de la fréquence sur les temps d'exécution des tâches qui sont en dehors du cadre de cette thèse.

Les figures 5.1 et 5.2 montrent les différents temps d'exécution d'un noyau de type *gemm* selon la politique du *governor* choisie. Dans le graphe 5.1 où le *governor* par défaut (*ondemand*) a été utilisé, on voit bien que les temps d'exécution s'étendent entre 6ms et 13ms pour une exécution à 32 cœurs et entre 6ms et 22ms pour une exécution à 64 cœurs, soit une différence de 16ms entre les temps d'exécution d'un même type de tâches. Cependant, dans le graphe 5.2 où le *governor* a été fixé en mode *powersave*, les temps sont certes plus longs (au moins 13ms) mais présentent une variabilité plus limitée (maximum de 22s à 64 cœurs, soit une différence de 9ms). Les autres types de tâches affichent aussi des résultats similaires aussi observés

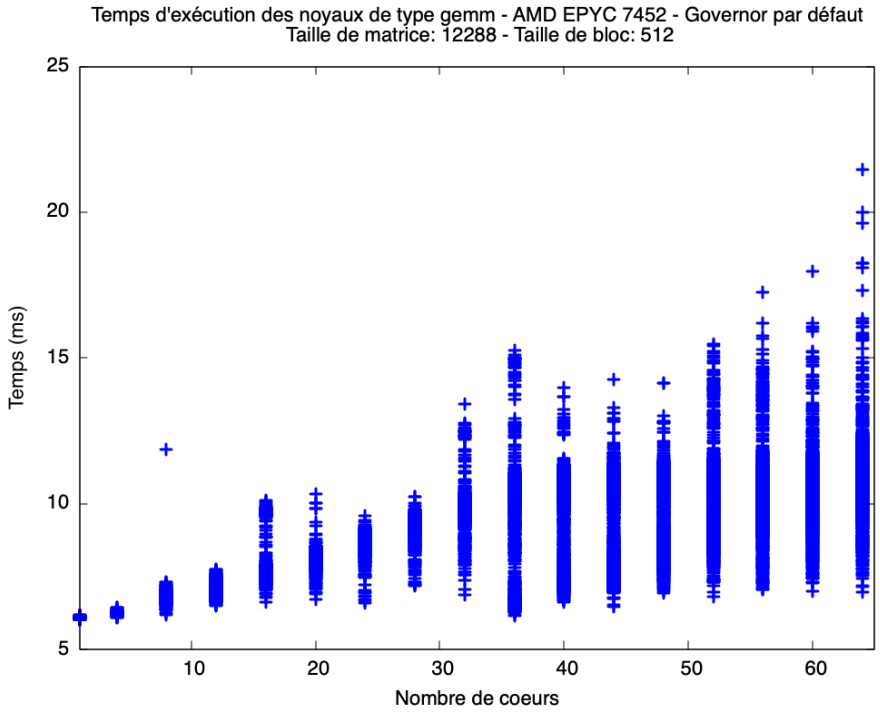


FIGURE 5.1 : Temps d'exécution des noyaux *gemm* de l'algorithme Cholesky en fonction du nombre de cœurs avec le *governor* par défaut

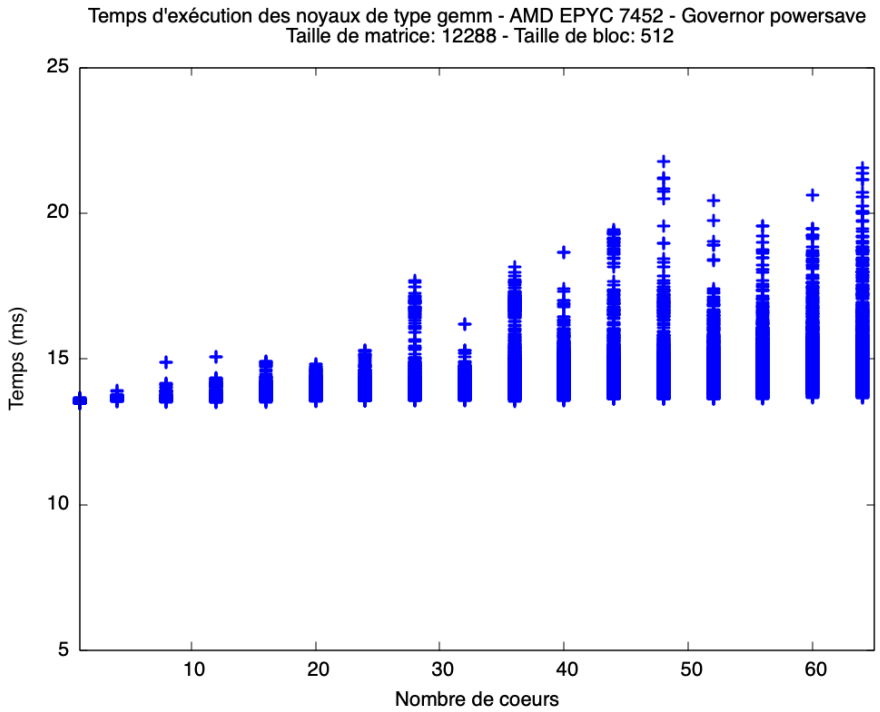


FIGURE 5.2 : Temps d'exécution des noyaux *gemm* de l'algorithme Cholesky en fonction du nombre de cœurs avec le *governor* en mode *powersave*

et présentés dans la thèse de Philippe Virouleau [42], ce qui montre donc l'impact de la fréquence sur les temps de calculs. La considération des effets du *governor* pouvant nous emmener loin, on choisit alors dans ce travail de n'utiliser que le mode *powersave*.

5.2 Impact de la localité des données sur la simulation

5.2.1 Le modèle COMM

Le modèle COMM est une incrémentation du modèle précédent (5.1.1) avec comme objectif de savoir si la prise en charge des transferts mémoire de manière naïve peut améliorer la précision des simulations. Le modèle TASK est donc étendu en modélisant les accès aux données induits par les dépendances de données entre les tâches : on traduit un accès par un transfert (ou communication) SimGrid (voir la section 4.1). Aussi, comme présenté dans la section 4.2, les traces obtenues avec TiKKi permettent non seulement de connaître les types d'opérations mémoire effectuées par chaque tâche de l'application, mais aussi de connaître les données qui ont été manipulées, représentées par un *handle*.

Dans une exécution réelle d'une application, les tâches d'allocation qui s'exécutent d'abord sur les cœurs placent les données dans les mémoires des bancs NUMA correspondants (*first-touch*), et sont suivies ensuite par les tâches de calcul (*gemm*, $potrf$, *geqrt*...) qui accèdent et manipulent ces données. Les emplacements sur lesquels ont été exécutées les tâches d'allocation permettent donc de déduire la localité des données de l'application auxquelles accèdent les tâches dans les mémoires principales (RAM) des nœuds NUMA, accès qui sont modélisés par sOMP avec des communications au sens SimGrid.

Cependant et comme mentionné au début de la section 5.1.1, il est important de noter que dans l'exécution réelle, **calculs et accès mémoire se chevauchent** (*overlapping*) : dans la conception du modèle, il faut donc prendre un considération un ratio de chevauchement (*overlap ratio*) qui sera discuté dans la sous-section suivante.

5.2.2 Design d'implémentation

Grâce aux propriétés super-scalaires des architectures modernes, les tâches d'algèbre linéaire dense sont un entrelacement de lectures, de calculs et d'écritures. Une tâche s'exécutant donc sur un cœur d'un CPU effectue des instructions arithmétiques entrelacées avec des instructions mémoire (load/store). Par ailleurs, en fonction de la qualité de l'implémentation et du comportement des cœurs, la latence des instructions mémoire peut être plus ou moins recouverte d'instructions arithmétiques. En d'autres termes, sur toute la durée de la tâche, le temps d'exécution des instructions mémoire est plus ou moins superposé avec le temps d'exécution des calculs, formalisé par l'équation suivante :

$$max(T_C(t_i), T_M(t_i)) \leq Temps(t_i) \leq T_C(t_i) + T_M(t_i) \tag{5.2}$$

avec $T_M(t_i)$ le temps d'exécution des accès mémoire de la tâche t_i.

Dans notre cas d'application d'algèbre linéaire dense, les tâches sont composées d'un seul appel BLAS. Par exemple pour les tâches de type *gemm*, la multiplication matrice-matrice est généralement soigneusement conçue pour obtenir un chevauchement suffisant. Cependant, ceci est beaucoup moins le cas pour les tâches de la factorisation QR. Par conséquent, il est nécessaire de définir un ratio de chevauchement, c'est-à-dire un facteur représentant la quantité de temps d'accès aux données recouvert par du temps de calcul.

Dans ce travail, on définit ce ratio par l'expérimentation : d'autres pistes seront discutées dans le chapitre 7. En expérimentant avec différentes valeurs pour trouver celles qui réduisent l'erreur de précision de la simulation, on trouve par exemple sur l'architecture AMD que le temps de calcul est couvert à 60% dans le cas de la factorisation Cholesky, 4% dans la cas QR et 10% pour LU, ce qui confirme bien la remarque effectuée précédemment sur la qualité d'implémentation des tâches Cholesky (principalement les tâches *gemm* qui sont dominantes). Cela signifie donc que si le temps des accès mémoire est inférieur à 60% du temps des calculs arithmétiques, il est considéré comme totalement recouvert par ces derniers, sinon, on rajoute le surplus au temps de calcul.

Une manière de reconfirmer ces valeurs est de simuler ces applications pour différentes tailles du problème tout en gardant le même ratio, ce qu'on verra dans le Chapitre 6.

Les instructions mémoire qui sont prises en compte sont celles qui lisent et/ou écrivent les opérandes d'entrée et de sortie de la tâche (autrement dit des blocs de matrice) ou le tampon *scratch* : elles sont donc regroupées par opérande pour simplifier la simulation. On ignore les accès aux variables scalaires ou vectorielles locales qui tiennent dans la cache L1 et qui sont donc déjà prises en compte dans $T_C(t_i)$. Aussi, le regroupement par opérande permet une compatibilité avec le modèle de programmation SimGrid qui est orienté vers les plateformes à mémoire distribuée : les accès mémoire de la tâche sont modélisés par des transferts de données par opérande de tâche, c'est-à-dire comme des communications SimGrid entre le cœur et le contrôleur mémoire (une communication par opérande).

Toutefois, étant donné que les applications d'algèbre linéaire accèdent généralement au contenu des opérandes de manière entrelacée, ces communications sont effectuées simultanément par mode d'accès : toutes les opérations mémoire de type lecture (*read*) sont simultanées, et toutes les opérations mémoire de type écriture (*write*) sont également simultanées. Cependant, les communications des différents modes d'accès sont effectuées de manière séquentielle, car généralement, une tâche lit d'abord ses données, effectue ensuite les calculs, et enfin réécrit le résultat dans la mémoire. $T_M(t_i)$ peut donc s'écrire :

$$T_M(t_i) = \max_{j=1}^{n} T_{CommR}(a_{i,j}) + \max_{j=1}^{n} T_{CommW}(a_{i,j}) \tag{5.3}$$

où n est le nombre d'accès mémoire, $a_{i,j}$ est la j-ième opérande de la tâche t_i, et $T_{CommR}(a_{i,j})$ (resp. $T_{CommW}(a_{i,j})$) le temps de lecture (resp. écriture) de $a_{i,j}$ en fonction de son emplacement NUMA et du CPU qui exécute t_i.

La figure 5.3 permet de visualiser ceci : la tâche doit effectuer deux lectures (*Read1* et *Read2*) qui sont donc effectuées en parallèle, avec la première étant plus

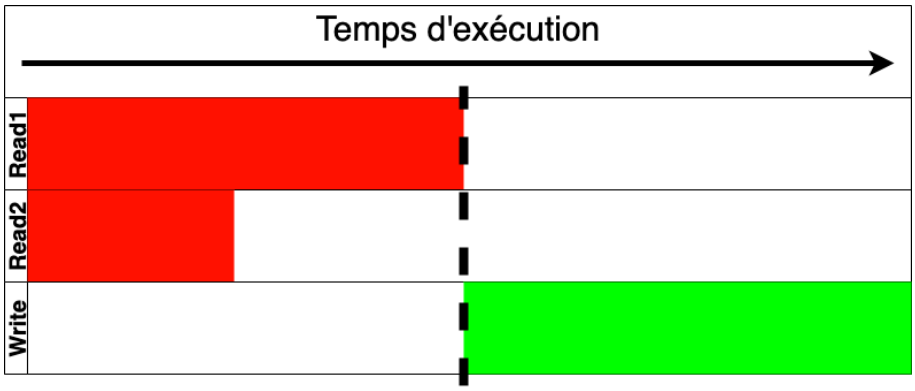


FIGURE 5.3 : Exemple d'une tâche effectuant 2 lecteurs et une écriture sur un bloc de matrice

lente car ses données proviennent d'un nœud NUMA distant. Une fois les lectures terminées (délimité en pointillés), la tâche peut alors entamer la phase d'écriture.

Par conséquent, l'ensemble des tâches s'exécutant en même temps sur les différents cœurs induit un ensemble correspondant de communications qui progressent de façon concurrente sur la plateforme : SimGrid peut alors déterminer, à chaque pas de temps de la simulation, le partage de bande passante entre les communications et ainsi prendre en compte la contention sur les liens simulés [61].

5.2.3 Discussion

Le modèle COMM permet donc d'améliorer les simulations en prenant en compte la localité des données : les différents flux de données entre les composants de l'architecture modélisés par des communications impactent le temps d'exécution de l'application et dépendent des paramètres des liens traversés ; on crée alors des effets de contention et de concurrence qui influencent les temps simulés. Cependant, il est essentiel de noter que lors de l'exécution réelle de l'application, la localité des données n'est pas statique : des effets de cache entrent en jeu en plus des effets NUMA.

En effet, lorsqu'une tâche s'exécute sur un cœur donné, les blocs de la matrice utilisés restent dans le cache L3 (en supposant que la taille du bloc est supérieure à la taille des caches L1 et L2) auquel le cœur a accès. En conséquence, si une tâche ayant accès au même cache L3 a besoin de ces mêmes blocs, le cœur qui exécute cette tâche les récupère depuis ce cache L3 au lieu de la mémoire principale (RAM), ce qui est moins coûteux : le transfert est plus rapide et économise de la bande passante d'interconnexion.

On peut donc dire que le modèle COMM est pessimiste, car il ne considère que les temps d'accès avec des données qui ne bougent pas, entraînant un surcoût en termes de temps de communication. Pour remédier à cela, on voit donc qu'il est nécessaire de modéliser un niveau de cache (L3 en l'occurrence) pour prendre en compte les effets de mouvements et de réutilisation des données entre les tâches.

5.3 Impact de la réutilisation des données et de leurs mouvements sur la simulation

5.3.1 Le modèle COMM+CACHE

A l'image du modèle précédent, le modèle COMM+CACHE est une incrémentation du modèle COMM (où seuls les accès mémoires étaient considérés sans prendre en compte les mouvements des données). Le but du modèle COMM+CACHE est donc de savoir dans quels caches se trouvent les données manipulées par les tâches à n'importe quel moment de l'exécution, ce qui permettra de mieux modéliser les transferts de données entre les cœurs du CPU, les caches et la mémoire principale.

Pour réaliser cela, l'objectif est donc d'implémenter un mécanisme de cache pour simuler les mouvements des données : on modélise donc les caches L3 des architectures étudiées. Comme mentionné précédemment, le choix de ne considérer que les LLC se justifie par le fait que les tailles de blocs utilisées dans les applications d'algèbre linéaire étudiées dépassent largement les tailles des caches L1 et L2. D'un autre point de vue, et comme explicité précédemment dans les motivations du projet de pouvoir réaliser des simulations rapides, on fait un compromis entre coût de simulation et précision : la simulation des autres niveaux de caches amènerait un niveau de détails plus important au profit d'une simulation beaucoup moins rapide et pas forcément plus précise, puisque les caches L1 et L2 ne sont pas partagés entre tous les cœurs d'un banc NUMA : ils ne présentent donc pas de phénomènes relatifs à la localité des données que nous aurions intérêt à modéliser.

5.3.2 Design d'implémentation

Modélisation des caches

Pour implémenter donc la prise en compte des mouvements dans la mémoire, nous considérons la taille réelle du cache L3 dans l'architecture cible et la taille entière (atomique) d'un bloc de matrice. Le cache est alors modélisé sous forme de créneaux ou *slots*, leur nombre étant égal à :

$$N_{slots} = \frac{taille_{cache}(octets)}{taille_{bloc}(octets)} \tag{5.4}$$

Chaque bloc de matrice occupe donc un *slot* dans le cache dont le comportement est implémenté de façon à reproduire le comportement d'un cache L3 réel : lorsque les données y sont insérées, on utilise l'algorithme de remplacement LRU (*Least Recently Used*), dont l'idée sous-jacente est de garder les données récemment utilisées (conformément au principe de localité) et d'évincer les données les moins récemment accédées. Il s'agit là d'une approximation simple de l'associativité réelle des caches, puisqu'on ignore par exemple les conflits de défaut de cache (*cache miss conflicts*). Par ailleurs, lors de l'exécution de la tâche, les données sont verrouillées dans le cache (c'est-à-dire qu'elles ne peuvent pas en être évincées), puisqu'elles sont requises par la tâche tout au long de l'exécution.

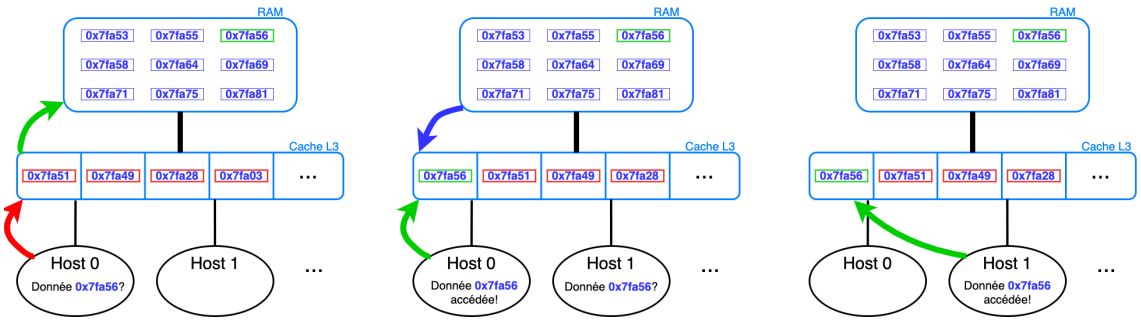


FIGURE 5.4 : Simulation des transferts mémoire dans sOMP : ici (de gauche à droite), le *host* 0 requiert une donnée qui ne se trouve pas dans le cache local (flèche rouge) mais dans la RAM (flèche verte) : elle est donc rapatriée depuis la mémoire principale vers le cache L3 (flèche bleue) ; ensuite, lorsque le *host* 1 requiert la même donnée, il accède directement au cache local pour la récupérer (flèche verte)

Modélisation des accès mémoire

Tandis que le modèle précédent (COMM) enregistrait la localité des blocs d'une matrice dans la mémoire principale (RAM) associée aux nœuds NUMA en fonction des tâches d'allocation, le modèle COMM+CACHE permet de suivre les mouvements des données dans les caches et la mémoire principale de la plateforme. La notion de localité est donc plus complexe, et plusieurs scénarios sont possibles :

1. les données sont dans le cache L3 local ;
2. les données sont dans un cache L3 distant ;
3. les données sont dans la RAM du nœud NUMA local ;
4. les données sont dans la RAM d'un nœud NUMA distant ;

A part pour le premier cas, les données doivent donc être transférées de l'emplacement distant vers le cache L3 local avant que le cœur d'un CPU ne puisse les charger pour exécuter les calculs : par conséquent, on modélise cela par une communication SimGrid entre le cache ou RAM distants et le cache local, suivie d'une communication entre le cache local et le cœur simulé (*host*), comme explicité sur la figure 5.4. Aussi, si une autre tâche exécutée sur un cœur du même banc NUMA requiert l'accès à ces mêmes données, seul un transfert entre le cache L3 local et le cœur est simulé, si ces données n'en ont pas été évincées entre-temps, ce qui permet de décongestionner les liens inter-sockets, modélisant ainsi le comportement réel de l'application sur la machine.

D'autre part, lorsqu'une tâche modifie un bloc de matrice, on supprime le bloc de tous les autres caches L3 s'il s'y trouve, de sorte que les cœurs soient obligés de le recharger s'ils exécutent des tâches qui ont besoin des nouvelles valeurs du bloc (pour simuler le fait que les cœurs doivent récupérer la nouvelle version du bloc). Par ailleurs, lorsque le bloc modifié est supprimé du cache L3, son contenu doit être retransféré vers la RAM de nœud NUMA correspondant, modélisé par une communication entre le cache L3 et la RAM où la tâche d'allocation l'a initialement alloué.

Dans le cas de la factorisation QR, l'une des opérandes des blocs utilisés par les tâches est un espace de travail (*workspace*) temporaire et est présent en permanence dans le cache L3 local (dans le *stack*). Cela signifie que cet espace de travail (appelé *scratch*) est en fait stocké par cœur et continue d'être réutilisé par les tâches exécutées sur le même cœur : on le modélise alors avec un bloc de matrice par cœur (qui occupe donc en permanence un *slot* du cache) sur lequel les tâches écrivent uniquement, ce qui nous permet de modéliser correctement le fait que les caches L3 stockent les espaces de travail de leurs cœurs correspondants.

5.3.3 Discussion

Tandis que le modèle COMM utilise une politique de localité des données "pessimiste" en considérant que toutes les données sont distantes et statiques, le modèle COMM+CACHE permet la prise en compte des mouvements entre caches et mémoires principales, permettant ainsi l'amélioration des simulations en modélisant au plus près l'occupation des caches L3 et les transferts de données entre les caches L3 et la RAM pour une meilleure prédiction du comportement de l'application.

Comme précisé dans la section 5.3.1, un compromis entre la précision de la simulation et son coût est fait et justifie la modélisation des LLC uniquement. Cette modélisation est bien adaptée aux noyaux d'algèbre linéaire dense (qui est notre cas d'étude ici), cependant, pour des applications d'algèbre linéaire creuse, une modélisation beaucoup plus complexe du comportement des tâches est nécessaire : on ne pourra pas simplement diviser le cache en *slots* puisque les données auront des tailles différentes. De plus, le comportement calcul des noyaux sera probablement irrégulier, comme observé dans le chapitre suivant dans le cas de la factorisation LU (dû aux tâches *lawsp*).

Par ailleurs et comme explicité précédemment, les simulations qu'on effectue font la distinction entre calcul et transferts mémoire, et leur entrelacement est représenté par le facteur de chevauchement discuté dans la section 5.2.1. Cependant, dans l'exécution réelle, calculs et accès mémoires ne se chevauchent pas seulement, mais aussi influent l'un sur l'autre : lorsqu'un cœur de CPU effectue des calculs, il importe les données nécessaires à partir du cache local au fur et à mesure des instructions arithmétiques exécutées. Ce transfert dépend alors :

- de la fréquence à laquelle le cœur effectue les calculs, ce qui libère plus ou moins de la bande passante sur les liens traversés ;
- de la bande passante disponible, qui dictera en conséquent la vitesse avec laquelle le cœur effectuera les calculs ;

Ce sont donc des effets de **fluctuation de bande passante** qui ne sont pas pris en charge dans la simulation. Par conséquent, et pour intriquer calculs et opérations mémoire, des modifications à SimGrid même devraient être introduites pour changer le modèle de simulation utilisé par le framework, ce qui sera discuté dans la chapitre 7.

Chapitre 6

Évaluation

Sommaire

Après avoir présenté la conception du simulateur et des modèles de performances qui permettent de prendre en charge les effets NUMA, nous présentons dans ce chapitre les résultats de notre approche. Après avoir défini une métrique pour mieux quantifier la précision des simulations, nous exposerons la précision de sOMP en fonction des algorithmes d'algèbre linéaire dense présentés dans la section 4.3 sur les deux architectures discutées dans la section 3.2.1 en fonction du nombre de cœurs. On montrera aussi que notre outil est capable de fournir des prédictions dans un temps inférieur au temps nécessaire à l'exécution réelle. Enfin, on présentera un exemple d'utilisation de sOMP dans le cadre de la recherche en ordonnancement de tâches.

6.1 Méthodologie

Les graphes présentés dans ce chapitre (sauf mention contraire) sont en fonction du nombre de cœurs. Effectivement, l'idée est d'observer les effets architecturaux. Dans les figures de résultat pour la plateforme AMD par exemple, comme dans la figure 6.1, le début de la courbe (jusqu'à 4 cœurs) correspond à l'exécution sur un CCX, puis correspond au premier socket entre 4 et 32 cœurs, puis avec les deux sockets pour les points restants.

La figure 6.1 présente les performances du simulateur comparé à une exécution native en termes de GFlops. On constate que l'efficacité d'exécution de l'application est de moins en moins bonne (coefficient directeur important jusqu'à 16 cœurs qui

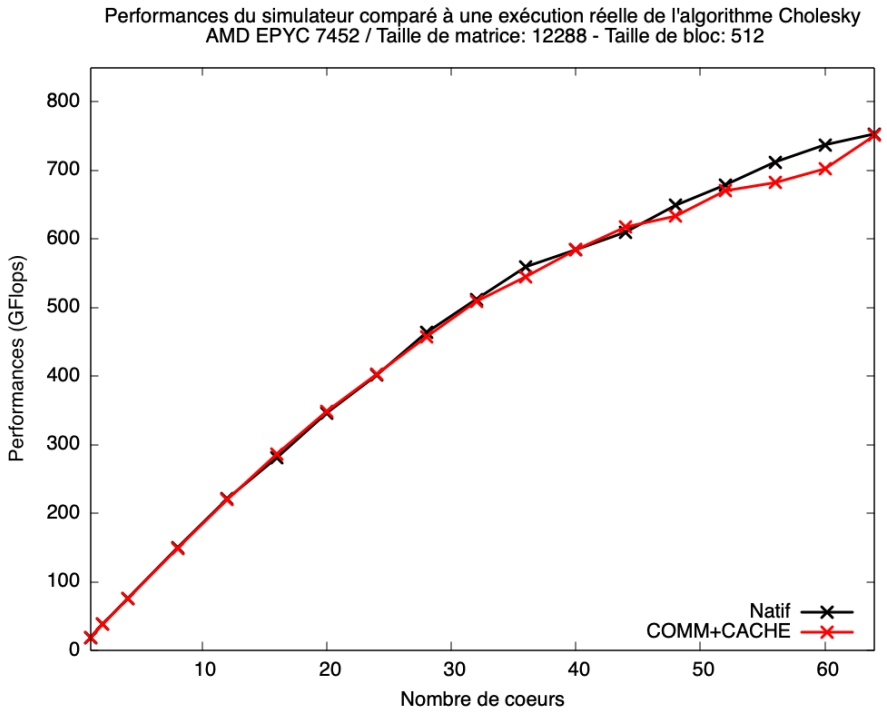


FIGURE 6.1 : Performances du simulateur en terme de GFlops sur la machine AMD

diminue entre 12 et 32 cœurs, puis diminue plus significativement dans le reste de la courbe). C'est justement cette perte de performances que l'on cherche à simuler. On voit par contre que la différence entre le cas natif et la simulation est faible et difficilement quantifiable sur une telle figure, ce qui nécessite en vue de comparer les modèles de définir une métrique $Erreur_{precision}$ qui exprime l'erreur de précision relative du simulateur :

$$Erreur_{precision} = \frac{T_{nat} - T_{sim}}{T_{nat}} \tag{6.1}$$

avec T_{sim} le temps d'exécution simulé et T_{nat} le temps d'exécution réel.

Par conséquent, lorsque l'erreur de précision est positive, cela signifie qu'on "sous-simule" le temps d'exécution réel, en d'autres termes, que notre prédiction est optimiste. Une erreur de précision négative signifie que l'on "sur-simule", d'où une prédiction pessimiste. Plus généralement, une courbe proche de 0 signifie donc une meilleure précision.

6.2 Précision des simulations

6.2.1 Résultats sur la plateforme Intel

Les résultats obtenus dans le cas de l'algorithme Cholesky sur la plateforme Intel sont présentés dans la figure 6.2. Comme explicité dans la section 5.1.1, le modèle TASK ne prend en considération que les temps d'exécution des tâches. On observe que ce modèle est moins précis au-delà de 18 cœurs, soit le nombre de cœurs du premier domaine NUMA de la machine (qui est également un socket). Le modèle

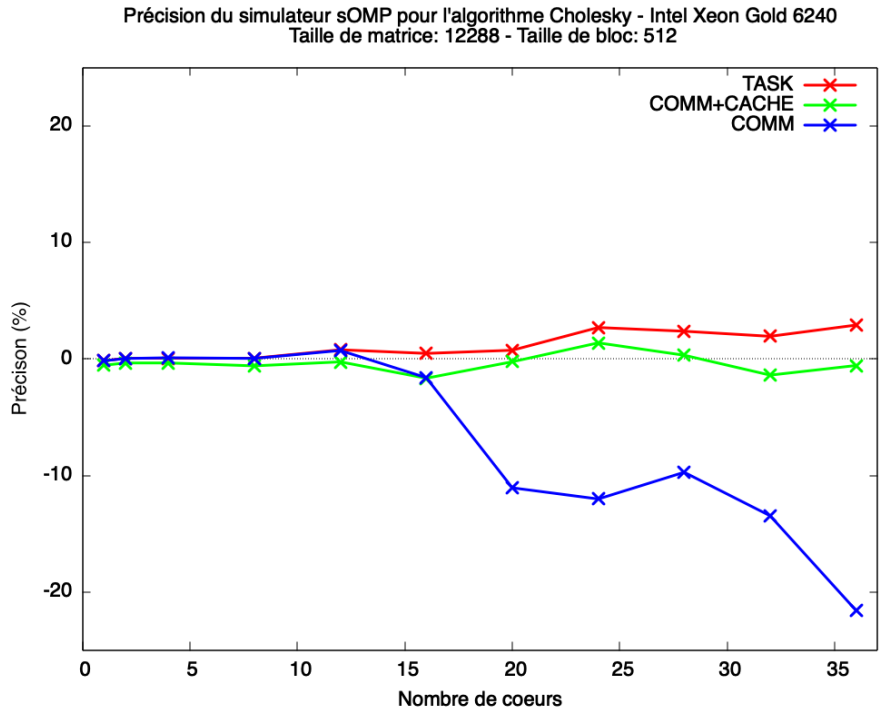


FIGURE 6.2 : Précision de sOMP pour l'algorithme Cholesky sur la machine Intel

TASK est donc optimiste (environ +4% d'erreur de précision pour 36 cœurs) ce qui était le comportement attendu, puisque la localité des données et les accès mémoire ne sont pas pris en compte. Par ailleurs, le modèle COMM est quant à lui beaucoup trop pessimiste puisqu'il considère que toutes les données sont statiques et dans la mémoire principale (RAM). La machine Intel ne contenant qu'un seul banc NUMA par socket pour 18 cœurs, la perte de précision est très significative puisque tous les cœurs simulés provoquent par leurs accès à la RAM une contention très importante qui résulte alors sur une précision très pessimiste, ce qui était encore une fois le comportement attendu puisque les mouvements des données et leur localité dans le cache ne sont pas pris en compte. Le modèle COMM+CACHE quant à lui permet, par son intégration d'un mécanisme de cache, de corriger le pessimisme du modèle COMM mais aussi d'offrir une meilleure précision au-delà de 18 cœurs et jusqu'à 30 cœurs. Lorsqu'on utilise encore plus de cœurs, le modèle offre une meilleure précision que le modèle TASK avec une précision d'environ -0.5% pour 36 cœurs.

Il est important de noter que, dans le cas de l'architecture Intel, on voit que le modèle TASK fournit déjà une précision satisfaisante pour les algorithmes où les calculs sont plus dominants. La figure 6.4 confirme cette observation dans le cas de l'algorithme LU : bien que le pivotage de cet algorithme introduise un comportement chaotique lorsqu'on utilise peu de cœurs, le modèle TASK ré-affiche une bonne précision dès 16 cœurs jusqu'à remplir toute la machine (environ 1.6% d'erreur de précision pour 36 cœurs). Le même comportement est aussi affiché par les modèles COMM et COMM+CACHE, le premier étant très pessimiste à cause des accès considérés distants, et le second améliorant légèrement la précision par rapport au modèle TASK. Cependant, le comportement des modèles change dans le cas de l'algorithme QR, présenté dans la figure 6.3. Effectivement, le modèle TASK n'est

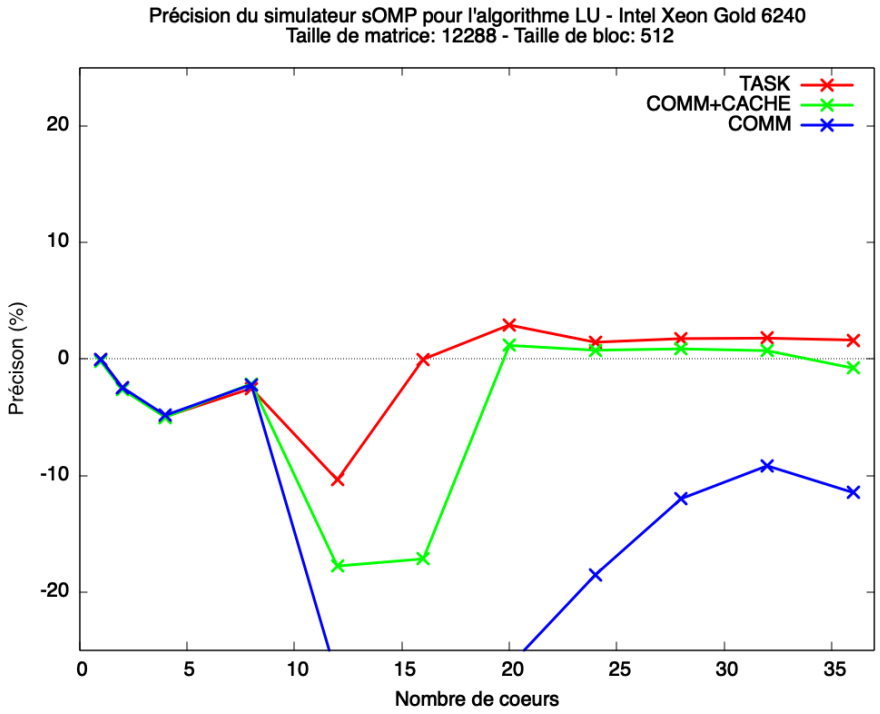


FIGURE 6.3 : Précision de sOMP pour l'algorithme LU sur la machine Intel

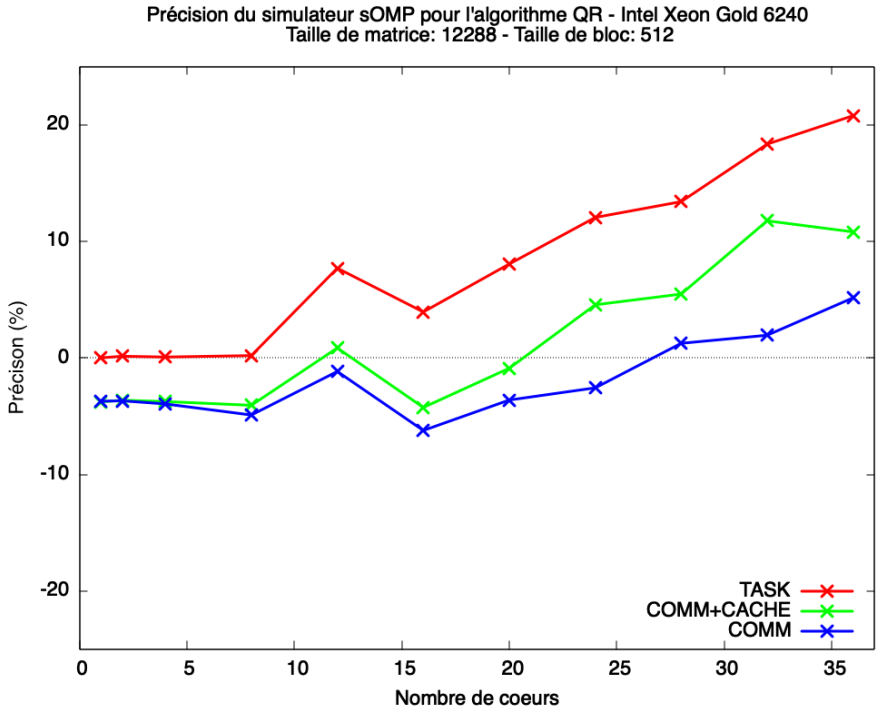


FIGURE 6.4 : Précision de sOMP pour l'algorithme QR sur la machine Intel

cette fois-ci plus suffisant pour obtenir une bonne précision, dépassant le seuil des 20% d'erreur de précision pour 36 cœurs. Tandis que le modèle COMM+CACHE améliore la précision du simulateur, le modèle COMM, par sa considération d'une mémoire statique, permet de compenser le manque de précision résultant sur une meilleure précision (environ 4% pour 36 cœurs).

De manière générale et d'après la précision du modèle TASK, on voit donc que la plateforme Intel n'affiche pas beaucoup d'effets NUMA pour les algorithmes liés aux calculs : la précision du modèle est déjà assez bonne, notamment sur le premier socket où nous avons en moyenne une erreur de précision de 0,8% dans le cas Cholesky par exemple. Cela est compréhensible puisque cette machine ne possède qu'un seul banc NUMA par socket, avec les 18 cœurs à l'intérieur du socket partageant le même cache L3. Cette configuration n'entraînera évidemment pas beaucoup de transferts de données à l'intérieur d'un même socket, par rapport à la machine AMD, qui comprend un total de 16 nœuds NUMA et 16 caches L3.

6.2.2 Résultats sur la plateforme AMD

Ainsi donc et comme présenté dans la figure 6.5 pour le cas de la factorisation Cholesky sur la plateforme AMD, le modèle TASK n'est plus précis par rapport aux résultats sur Intel. Sur le premier socket seul, on a en moyenne +3% d'erreur de précision, et on dépasse +9% lorsque tous les cœurs de la machine sont utilisés. La quantité de transferts des données (non simulée avec ce modèle) est en effet plus importante ici puisque nous avons plusieurs nœuds NUMA, et ne pas les prendre en compte est coûteux en précision. Le modèle COMM ne fournit pas non plus de résultats précis. Comme nous avons plus de transferts de données entre les nœuds NUMA sur cette machine, le modèle de communication est plus pessimiste avant même d'atteindre 32 cœurs (le nombre de cœurs sur le premier socket), et les communications inter-sockets accentuent encore le pessimisme de la simulation, jusqu'à atteindre -10%. Cette fois-ci encore, le modèle COMM+CACHE est le plus fiable en restant en plus cohérent quel que soit le nombre de cœurs utilisés : la modélisation de la réutilisation des données offre une meilleure précision (moins de 2% d'erreur de précision en moyenne).

Comme mentionné dans la section 3.2.2, le respect de la hiérarchie architecturale de la machine NUMA simulée est essentiel pour obtenir une simulation précise. Ceci est illustré par la courbe orange de la figure 6.5 (nommée "COMM+CACHE avec plateforme simple") qui présente les résultats de simulation lorsqu'on modélise la plateforme AMD sans prendre en considération la hiérarchie fine entre les CCX, les CCD et les nœuds NUMA, et que l'on utilise la bande passante indiquée par l'outil Intel MLC qui agrège les bandes passantes des différents nœuds NUMA d'une même socket. On voit donc que la précision est similaire au cas où seuls les temps d'exécutions sont considérés (modèle TASK) alors que le modèle COMM+CACHE est utilisé, ce qui prouve qu'un modèle de performance seul (aussi sophistiqué qu'il soit) n'est pas suffisant, et qu'une modélisation précise des hiérarchies entre les composants de la machine est une condition critique pour obtenir une simulation précise.

La figure 6.6 montre les résultats obtenus pour la factorisation QR. Le modèle TASK est optimiste car il diverge régulièrement, depuis l'exécution sur 9 cœurs et

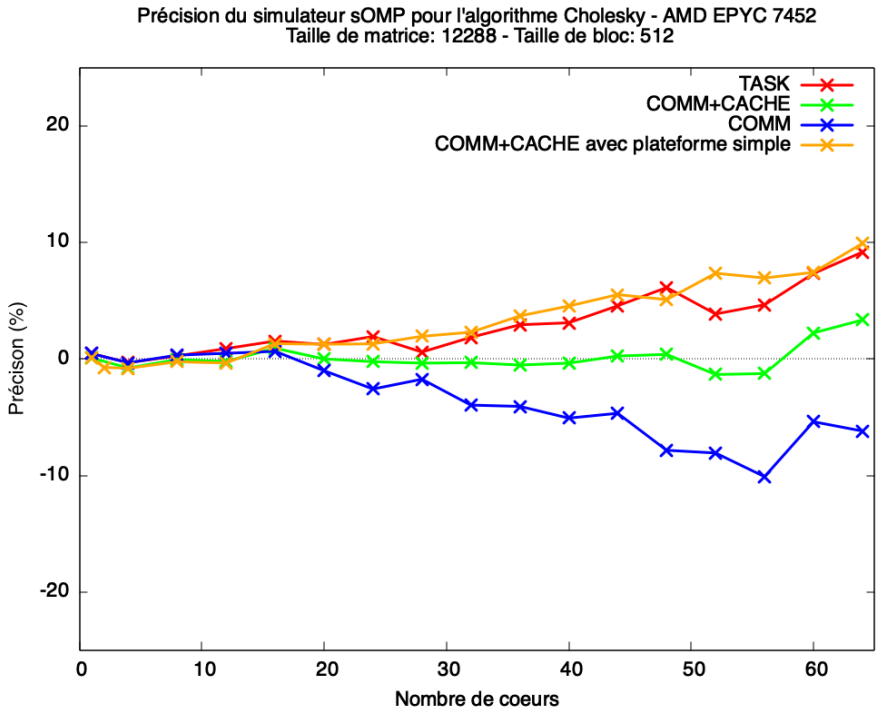


FIGURE 6.5 : Précision de sOMP pour l'algorithme Cholesky sur la machine AMD

jusqu'à l'exécution sur 64 cœurs, tandis que le modèle COMM est à nouveau très pessimiste. D'un autre côté, le modèle COMM+CACHE semble éviter de devenir trop optimiste et capte les effets de placement des données en cache L3. Cependant, cela ne semble pas éviter la divergence dont est affecté le modèle TASK pour les exécutions sur le second socket, à partir des exécutions sur 40 cœurs. Cette perte de précision peut s'expliquer par les effets de la variation de bande passante sur les noyaux des applications : comme nous utilisons plus de cœurs, la contention sur les liens de la machines réduit la bande passante disponible, modifiant ainsi le temps de calcul qui peut chevaucher les transferts de données ralentis, et résultant sur un temps d'exécution de l'application plus lent. Cela rend notre modèle COMM+CACHE optimiste lorsque de nombreux cœurs de la machine sont utilisés, comme observé sur la figure 6.6, qui est le scénario où ces effets ont le plus d'influence sur le temps d'exécution de l'application (factorisation QR). Considérer ces effets nécessiterait d'affiner le modèle d'exécution d'une tâche à l'intérieur même de SimGrid, pour enchevêtrer subtilement le temps d'exécution et les transferts de mémoire, ce qui dépasse le cadre de cette thèse.

Les résultats de la factorisation LU sont présentés dans la figure 6.7. L'erreur de précision des différents modèles varie significativement dans le cas d'exécutions avec peu de cœurs, comme observé dans la simulation LU sur la machine Intel. Ce sont en fait les mesures natives qui ont un comportement de performance chaotique. En effet, la factorisation LU utilise le pivotement, qui amène un comportement erratique en fonction du contenu des matrices, qui n'est pas pris en compte dans la modélisation des performances d'une tâche comme étant la moyenne de ses temps d'exécution obtenus dans la trace séquentielle. De ce fait, ce comportement fortement dépendant de ces valeurs n'est pas simulé. Comme dans le cas Intel aussi,

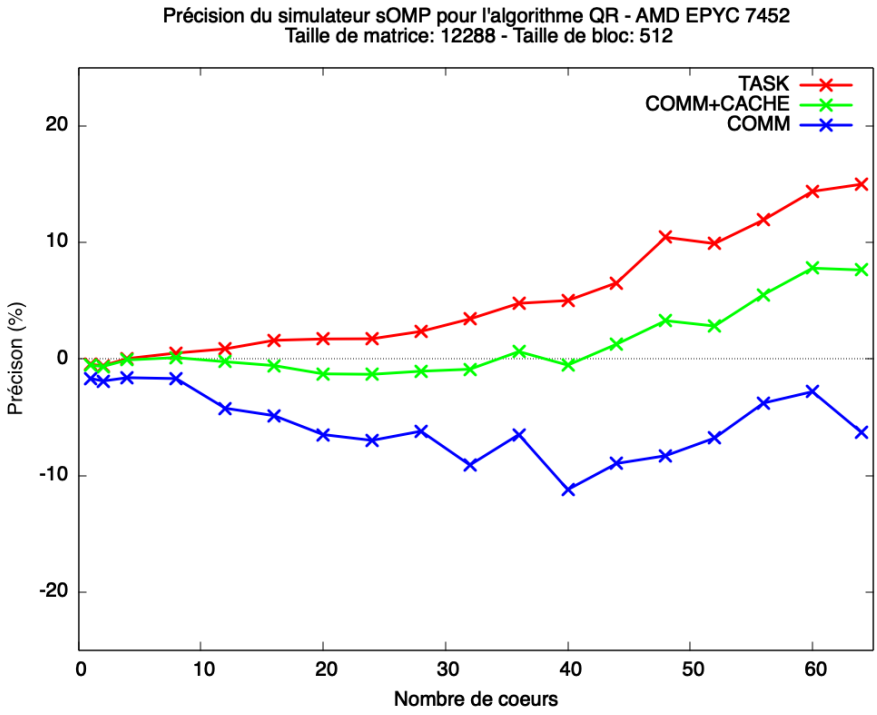


FIGURE 6.6 : Précision de sOMP pour l'algorithme QR sur la machine AMD

avec plus de cœurs, ce mauvais comportement dans l'exécution native s'aplanit et le modèle COMM+CACHE reproduit correctement le comportement global de calcul/communication de la factorisation, tandis que le modèle TASK reste trop optimiste et que le modèle COMM reste trop pessimiste.

6.2.3 Résultats pour différentes tailles de matrices

Dans l'ensemble, nous avons montré que nos modèles sont capables d'atteindre une bonne précision pour divers algorithmes d'algèbre linéaire dense. Les résultats ont jusqu'à présent été présentés pour une taille de matrice 12288 $\times$ 12288, mais même avec exactement les mêmes paramètres de simulation (modèle de plateforme, durée moyenne des tâches, facteur de chevauchement), d'autres tailles de matrice présentent le même type de résultats. Nous avons résumé les résultats pour les tailles de matrices plus grandes dans les tableaux 6.1, 6.2 et 6.3, qui montrent les erreurs de précision correspondantes lors de la simulation d'applications utilisant tous les cœurs de la machine. Nous observons que notre simulateur reste fiable malgré l'augmentation de la taille de la matrice, c'est-à-dire malgré une augmentation du nombre de tâches et de transferts de données. Le modèle COMM+CACHE reste précis, avec une moyenne d'erreur de précision d'environ 1,4% sur les tailles de matrice présentées pour la factorisation de Cholesky, 5% pour QR et 2.2% dans le cas LU. Pour résumer, nous obtenons de bons résultats sur la plateforme Intel qui est une architecture simple ne montrant pas beaucoup d'effets NUMA, mais aussi de bons résultats pour les mêmes algorithmes sur la plateforme AMD, malgré son architecture très complexe.

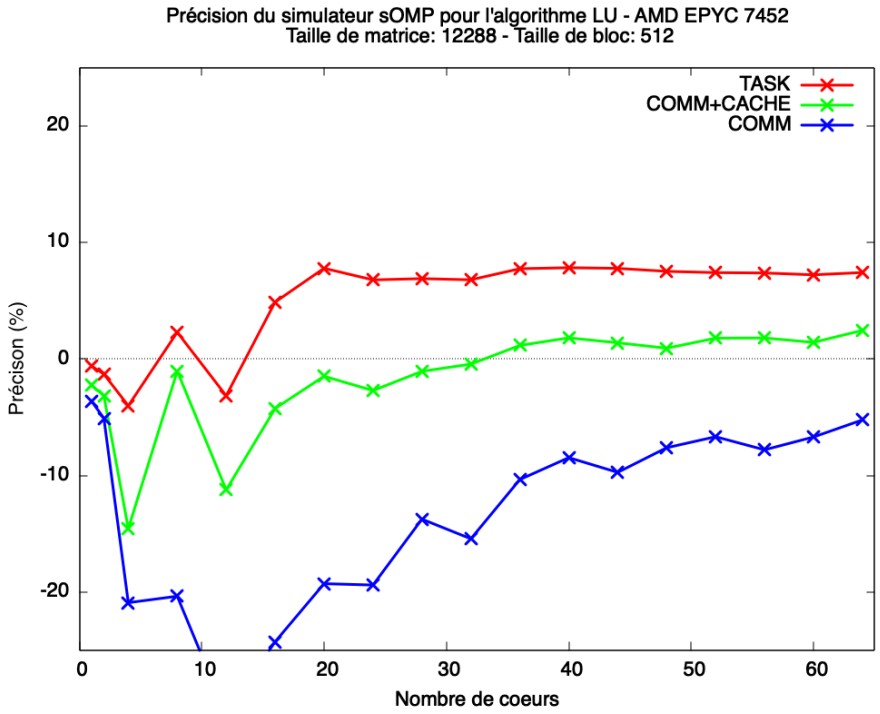


FIGURE 6.7 : Précision de sOMP pour l'algorithme LU sur la machine AMD

Model \ Matrix size	12288	16384	20480	24576
TASK	9.1%	7.5%	6.5%	5.1%
COMM	-6.1%	-12.4%	-6.9%	-7.2%
COMM+CACHE	**3.1%**	**0.1%**	**1.1%**	**1.4%**

TABLE 6.1 : Précision pour l'algorithme Cholesky pour différentes tailles de matrice pour une exécution à nœud plein

Model \ Matrix size	12288	16384	20480	24576
TASK	14.9%	10.7%	9.4%	9%
COMM	-6.3%	-15.9%	-17.5%	-21.8%
COMM+CACHE	**7.6%**	**5.2%**	**3.6%**	**3.9%**

TABLE 6.2 : Précision pour l'algorithme QR pour différentes tailles de matrice pour une exécution à nœud plein

Model \ Matrix size	12288	16384	20480	24576
TASK	7.4%	8.2%	8.5%	8.9%
COMM	-3.7%	-9.1%	-16.1%	-22.3%
COMM+CACHE	**3.5%**	**2.04%**	**1.9%**	**1.2%**

TABLE 6.3 : Précision pour l'algorithme LU pour différentes tailles de matrice pour une exécution à nœud plein

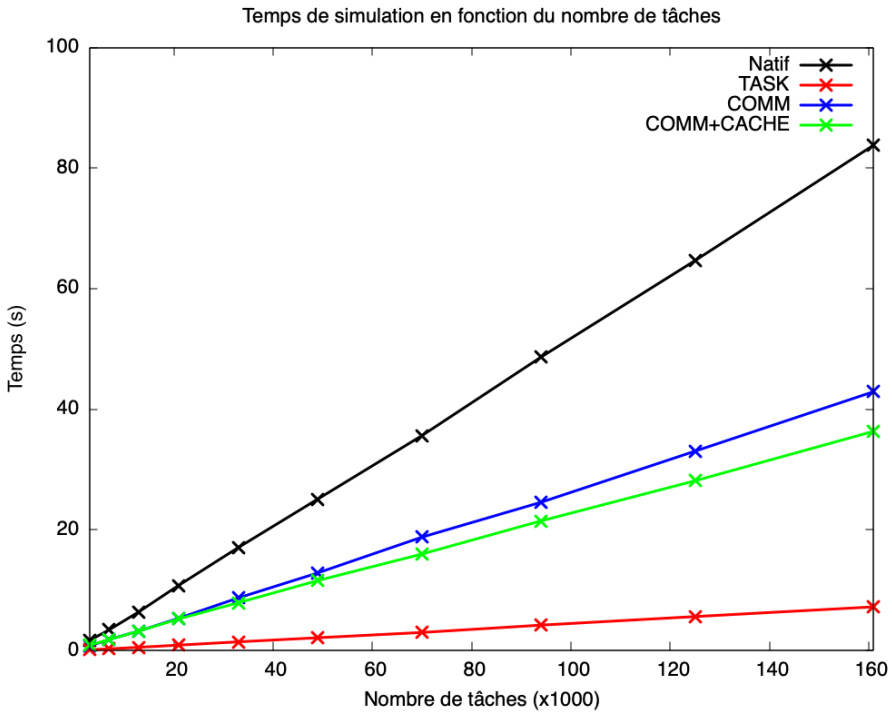


FIGURE 6.8 : Temps de simulation de l'algorithme Cholesky avec 64 cœurs sur un seul CPU Intel i5-8257U en fonction de différentes tailles de matrices représentées en termes du nombre de tâches, et avec une taille de bloc fixée à 512 x 512 (la même que celle utilisée pour les expérimentations précédentes)

6.3 Temps de simulation

En termes de performances, le simulateur sOMP répond à l'un des critères principaux énoncés dans la section 2.4, à savoir effectuer une simulation fiable et rapide. Ceci est démontré par les figures 6.8 et 6.9, qui présentent le temps que prend les simulations en fonction (resp.) du nombre de tâches et du nombre de cœurs. Effectivement, on observe dans la figure 6.8 que le temps de simulation de sOMP s'accroît linéairement en fonction du nombre de tâches, et est largement inférieur au temps d'exécution réel de l'application. Par exemple, pour une matrice de taille 49152 x 49152 avec des tuiles de 512 x 512, soit un peu plus de 161000 tâches sur tous les cœurs de la machine AMD (64 cœurs), l'exécution réelle est achevée en 83s, tandis que la simulation sur un ordinateur portable ordinaire (MacBook Pro 2019, équipé d'un CPU i5-8257U) avec le modèle le plus accompli (COMM+CACHE) prend moins de 37s, soit une réduction de plus de 50% du temps d'exécution natif pour une simulation exécutée sur un seul cœur.

Ce comportement est expliqué par la nature des simulations que l'on effectue : en effet, SimGrid utilise une simulation grossière et non une simulation au cycle précis (*cycle accurate*). Ainsi donc, tous les calculs réels d'une tâche sont remplacés par une seule étape de simulation, et toutes les opérations de lecture/écriture réelles pour un opérande de tâche donnée sont remplacées par une seule communication simulée.

Ces résultats sont aussi confirmés lorsqu'on ne considère que les simulations pour une seule taille de matrice mais pour des nombres de cœurs différents, comme

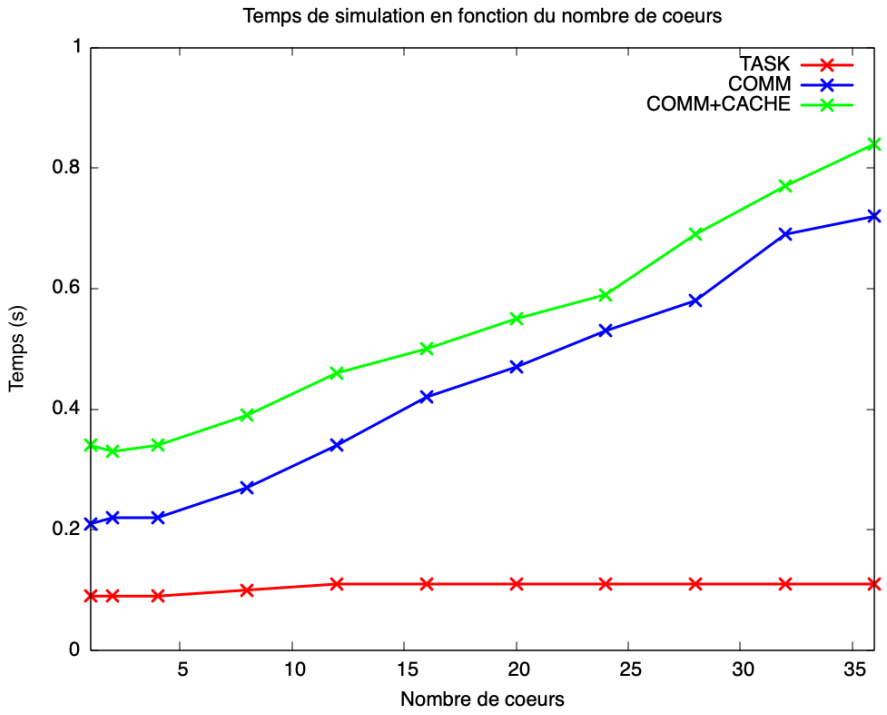


FIGURE 6.9 : Temps de simulation de l'algorithme Cholesky sur un CPU Intel i5-8257U en fonction du nombre de cœurs simulés pour plus de 3000 tâches

démontré sur la figure 6.9. On observe ainsi que pour une matrice de 12288 x 12288 avec des blocs de 512 x 512, le temps de simulation s'accroît aussi de façon linéaire en fonction du nombre de cœurs simulé pour les modèles COMM et COMM+CACHE, en raison du nombre accru de communications que SimGrid doit gérer simultanément, alors que le modèle TASK ne dépend que du nombre de tâches (puisqu'on n'y considère pas d'accès mémoire).

Il est cependant important de noter qu'on effectue ici une seule simulation à la fois : en d'autres termes, on pourrait faire du parallélisme massif en effectuant plusieurs simulations en simultané sur plusieurs cœurs.

6.4 Proof-of-concept : un ordonnanceur cache-aware

Les sous-sections précédentes ont montré que le modèle COMM+CACHE fournit des temps d'exécution simulés précis qui prennent en compte à la fois les effets des accès NUMA et de cache. Comme mentionné précédemment, des travaux antérieurs sur la simulation des plateformes basées sur GPU [7] avaient ouvert la porte à des recherches en ordonnancement sur de telles architectures à la fois réalistes et reproductibles [8]. Les résultats de simulation présentés dans cette thèse ouvrent maintenant de la même manière la porte à la recherche en ordonnancement réaliste et reproductible qui vise à optimiser l'affinité du cache. Ainsi donc, nous avons implémenté un ordonnanceur de tâches OpenMP *cache-aware* comme preuve de concept (*proof-of-concept*) : lorsqu'un cœur du CPU termine une tâche, la plupart des ordonnanceurs par défaut (et notamment l'ordonnanceur LLVM OpenMP utilisé dans

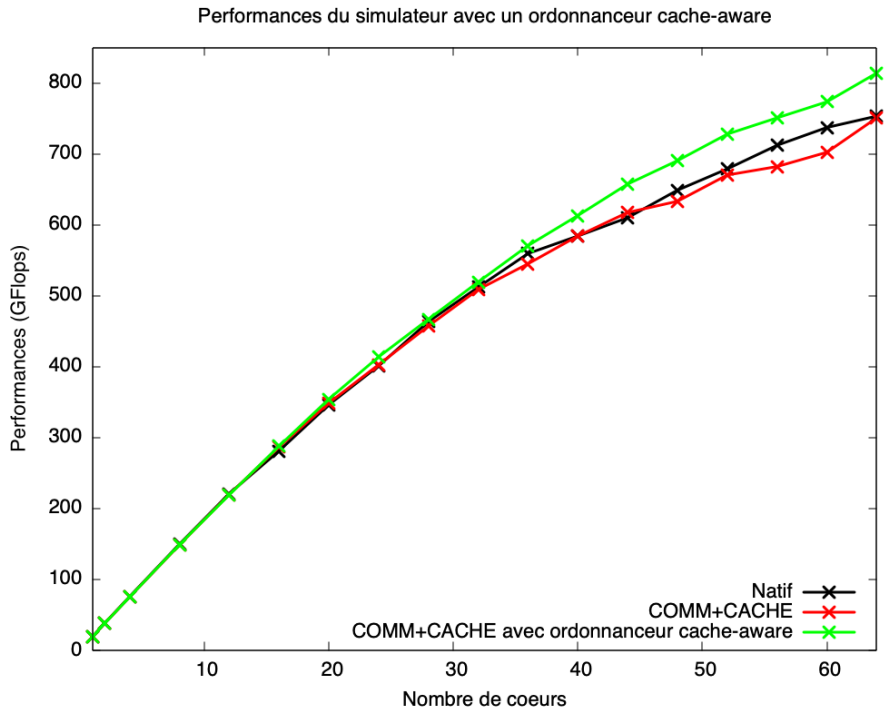


FIGURE 6.10 : Performance du simulateur sur un algorithme Cholesky avec et sans ordonnanceur cache-aware

ce travail) prennent la tâche suivante sans vraiment tenir compte de la localité [62]. Notre ordonnanceur quant à lui prend en considération le cache en privilégiant la sélection d'une tâche dont les opérandes de données sont déjà disponibles dans le cache L3 du cœur du processeur, réduisant ainsi les défauts de cache L3 (*cache miss*) et diminuant la quantité des transferts de données globaux sur la plateforme. Les résultats sont présentés sur la figure 6.10, en termes de GFlops selon le nombre de cœurs utilisés pour l'exécution.

On observe donc que le modèle COMM+CACHE arrive à bien reproduire le comportement de l'exécution native. Cependant, lorsqu'on fait en sorte que le simulateur utilise l'ordonnanceur amélioré, on remarque une amélioration des performances qui augmente avec le nombre de cœurs utilisés pour l'exécution, jusqu'à atteindre plus de 8%. Ceci prouve donc que les heuristiques aident à la scalabilité sur les systèmes multicœurs.

Il est important de noter que seul le modèle COMM+CACHE est capable de montrer cet effet en simulation, puisque ce sont les effets de cache qui apportent cette amélioration des performances.

Il est à noter aussi que l'implémentation actuelle de cet ordonnanceur est très simpliste : à chaque étape, on scanne toutes les tâches prêtes à être exécutées, pour en trouver une qui minimise les défauts de cache L3 (*cache miss*). Cela rend généralement la complexité égale au $(\textit{nombre de tâches})^2$. Dans notre cas, ceci n'est pas un problème avec notre simulation puisque les performances simulées de l'application ne sont pas affectées par le temps pris par l'ordonnanceur. Cependant, cette implémentation ne peut pas encore être intégrée dans un environnement OpenMP réel car son coût est prohibitif ; plus de travail d'un point de vue algorithmique est nécessaire pour

concevoir une implémentation avec une meilleure complexité.

C'est en fait là que résident les avantages de la simulation : on a pu prototyper rapidement un ordonnanceur et l'expérimenter pour observer les gains obtenus dans la simulation sans être affectés par le coût de l'ordonnanceur lui-même. Puisque ce dernier peut être mesuré séparément, cela permet de voir clairement le coût de l'ordonnanceur et les gains qu'il fournit. On peut donc affiner les heuristiques et essayer d'améliorer les gains de performances, sans se soucier de la complexité de l'implémentation, dans une première étape de prototypage. La simulation fournit également des résultats de performances beaucoup plus rapides que les exécutions réelles, sans tous les problèmes de réservation de la plateforme cible et sa variabilité qui entravent la reproductibilité des résultats. Il est même possible d'enquêter de manière reproductible sur les bugs d'ordonnancement : grâce à la trace de l'exécution simulée, il est possible de définir des points d'arrêt au moment exact où apparemment une mauvaise décision a été prise, et d'inspecter l'état de l'ordonnanceur.

Une fois qu'une heuristique qui fournit des gains solides est conçue, on passera du temps alors à produire une implémentation avec une complexité acceptable, pour une utilisation réelle dans un vrai runtime OpenMP. De même, les heuristiques de placement des données NUMA ont pu être expérimentées grâce à notre environnement reproductible, avant de passer du temps sur une implémentation réelle. À la différence du travail préliminaire présenté dans la thèse de Philippe Virouleau [42] et discuté dans la section 2.5.2, les simulations avec sOMP permettent de prédire avec précision les performances de l'implémentation de trois algorithmes d'algèbre linéaire dense, et non pas juste estimer une borne inférieure/supérieure des performances qu'il serait théoriquement possible d'atteindre, estimation qui d'ailleurs avait été faite pour le cas de la factorisation Cholesky uniquement.

De plus, le choix de présenter ici un *proof-of-concept* avec un ordonnanceur *cache-aware* n'est pas anodin : le but est de montrer que l'on peut aussi observer les performances apportées par des ordonnanceurs qui dépendent du cache, ce qui n'était pas possible avec son simulateur qui incluait un cache par cœur de taille infinie. Observer l'amélioration des performances de la figure 6.10 aurait été donc impossible.

Par ailleurs, notre simulation permet également la modification de la plateforme simulée et d'observer les effets obtenus sur les performances de l'application, ce qui était un axe majeur des perspectives discutées dans sa thèse : effectivement, les travaux de modélisation des effets NUMA et de l'impact des propriétés de la machine avaient été simplifiés en considérant uniquement les temps d'exécution de la tâche récupérés avec un outil qui permet le contrôle sur le placement des tâches ainsi que leurs données selon différents scénarios, ce qui rendait l'expérimentation avec une plateforme modifiée ou même imaginée par l'utilisateur impossible.

Chapitre 7

Conclusion et perspectives

Sommaire

7.1 Conclusion

Les systèmes multicœurs modernes sont basés sur une architecture d'accès mémoire non uniforme (NUMA). Dans cette thèse, nous avons présenté une façon de modéliser ce type d'architectures, mais aussi les applications d'algèbre linéaire dense à base de tâches qui peuvent être exécutées dessus.

Plus précisément, ce travail se concentre sur la simulation d'applications OpenMP à base de tâches dépendantes sur des architectures à mémoire partagée. D'abord, en utilisant les outils fournis par le framework SimGrid, nous avons réussi à exprimer une plateforme NUMA grâce aux éléments qui composent des architectures à mémoire distribuée (puisque notre modélisation d'une machine NUMA s'apparente à une machine distribuée), tout en respectant les hiérarchies architecturales des plateformes cibles et en modélisant finement les différentes interconnexions dans des micro-architectures plus ou moins complexes (une machine Intel avec un seul banc NUMA par socket, et une machine AMD comprenant 8 bancs NUMA par socket).

Nous avons ensuite montré qu'il était nécessaire de développer notre propre benchmark, afin d'obtenir les différentes informations des paramètres des liens qui inter-connectent les composants des architectures simulées. Plus précisément, l'obtention des valeurs de bande passante s'est révélé être difficile en raison de la multitude de liens ayant chacun leurs propres caractéristiques, et la documentation pour déterminer la topologie d'un socket est généralement sujette à des erreurs et fournit des valeurs largement optimistes : le développement de sOMP-BWB nous a donc permis d'effectuer nos propres mesures et d'obtenir chaque valeur souhaitée grâce au principe de lecteurs-écrivains, afin de fournir une modélisation précise des architectures à mémoire partagée.

Ensuite, nous avons développé sOMP, un simulateur pour applications à base de tâches afin de prédire leurs performances de manière précise et rapide. Destiné

aux chercheurs dans le domaine de l'ordonnancement, cet outil permet en effet l'implémentation et l'étude des heuristiques d'ordonnancement des tâches et de leurs performances dans un environnement reproductible. En utilisant les traces obtenues avec TiKKi, sOMP permet de simuler l'exécution des applications cibles sur les architectures NUMA modélisées avec SimGrid.

Enfin, sur ces structures NUMA, nous avons démontré que la prise en compte des effets mémoire est cruciale pour obtenir des simulations précises des applications d'algèbre linéaire dense : la modélisation du temps d'exécution d'une tâche seul n'est pas suffisant. Nous avons donc introduit différents modèles de performances qui permettent une modélisation plus ou moins fine des accès mémoires :

- **le modèle TASK** ne prend en considération que les temps d'exécution des tâches. Comme mentionné précédemment, cette approche ne permet pas d'obtenir une simulation précise des performances des applications étudiée : la prise en compte des opérations mémoire s'impose ;

- **le modèle COMM** permet de prendre en considération les effets des accès mémoire en exploitant les fonctionnalités offertes par SimGrid : bien qu'orienté vers la simulation sur des architectures distribuées, nous avons montré que l'on peut détourner l'usage de ce framework afin de construire un modèle qui prend en compte les opérations mémoires performées par chaque tâche sous forme des communications entre les différents composants de l'architecture simulée grâce aux informations de localité des données obtenues via les traces. Cependant, nous avons montré que ce modèle avait ses limites puisqu'on y considère que toutes les données sont distantes et statiques, ce qui n'est pas le cas dans les exécutions réelles ;

- **le modèle COMM+CACHE** permet, en plus du temps d'exécution et des opérations mémoires de chaque tâche, de prendre en compte les mouvements des données dans les architectures NUMA. Ces plateformes permettent une gestion particulière des données via les caches : ainsi donc, des données requises par une tâche et récupérées depuis la mémoire principale seront placées dans le cache L3 local : une autre tâche nécessitant l'accès à ces mêmes données les récupérera depuis ce cache local, libérant ainsi de la bande passante sur les liens mémoire et permettant donc d'obtenir de meilleures performances. La modélisation des caches L3 a donc été nécessaire pour prendre en compte les mouvements des données, résultant ainsi sur une meilleure prédiction des performances des applications cibles.

Par ailleurs, au cours du développement de ce projet, en plus des effets NUMA, nous avons rencontré différents effets qu'on a choisit de ne pas simuler :

- les effets de **fluctuation de la fréquence de cœurs** d'un CPU, gérée par le *frequency governor*, influent sur le temps d'exécution d'une tâche. Effectivement, l'augmentation (ou la baisse) de la fréquence d'un cœur impacte non seulement la vitesse des calculs, mais aussi les bandes passantes des liens entre la mémoire (ou le cache) et ce même cœur. Dans cette thèse, nous avons choisi d'éviter ces variations en définissant la politique du *governor* en *powersave*, ce qui fixe

la fréquence des cœurs à leur valeur minimale et nous permet aussi d'éviter d'éventuels effets de **dissipation thermique** ;

- les effets de **chevauchement entre instructions arithmétiques et instructions mémoire** impactent le temps d'exécution d'une tâche en l'augmentant ou en le diminuant en fonction de la quantité d'accès mémoire qui peut être recouverte par les calculs. Dans ce travail, nous avons défini un facteur de chevauchement par l'expérimentation en fonction de l'architecture et de l'algorithme ;

- les effets de **fluctuation de la bande passante** impactent, à l'instar des effets de fluctuation de la fréquence des cœurs, les temps d'exécution en fonction de la contention sur les liens mémoires : moins de bande passante entraînera un ralentissement des calculs (puisque le cœur a besoin des données pour les effectuer), ce qui modifiera la quantité de chevauchement entre calculs et accès mémoire modifiant ainsi le temps nécessaire pour achever l'exécution de la tâche.

7.2 Perspectives

Comme discuté dans la section précédente, notre simulateur sOMP permet la prédiction des performances des applications parallèles à base de tâches sur des architectures à mémoire partagée en prenant en considération les effets NUMA, tout en faisant le choix d'ignorer d'autres effets comme la fluctuation de la fréquence des cœurs. Ce travail ouvre la porte à une recherche **reproductible** en ordonnancement de tâches avec la prise en compte de la localité des données, et diverses perspectives s'ouvrent comme suite logique de ces travaux, en commençant d'abord par une généralisation des simulations pour les autres types d'applications d'algèbre linéaire (notamment l'algèbre linéaire creuse). L'étude pourra aussi être étendue pour des simulations sur plus de plateformes. Ensuite, l'intégration d'une collection d'ordonnanceurs de tâches OpenMP pourrait être envisagée, en plus d'un ensemble de politiques de placement de données (*first-touch*, *round-robin*...).

Une évolution naturelle de ces travaux aussi serait d'étendre les modèles de performances développés pour prendre en compte les effets non simulés par notre approche actuelle. Dans le cas du chevauchement entre les instructions arithmétiques et les instructions mémoire, la caractérisation du ratio d'*overlap* peut être affinée en définissant ce facteur par type de tâches : il faudra donc observer les compteurs de performances (fournis par PAPI en l'occurrence). Cela permettra de quantifier plus finement le chevauchement et d'avoir une idée de la qualité de l'implémentation de chaque type de tâche, qui pourra d'ailleurs être améliorée si le ratio tend vers 0.

Par ailleurs, l'utilisation de l'IA peut être une piste simple et efficace pour caractériser le comportement des *frequency governor* et en déduire l'impact sur les temps d'exécution des tâches. Dans l'état actuel, on fixe la politique du *governor* en *powersave*, mais une collecte (conséquente ?) de données sur les variations de la bande passante peut permettre d'entraîner facilement un réseau de neurones simple (en 3 couches) pour prédire la fréquence des cœurs et en déduire l'impact sur les temps d'exécution.

Pour les effets de fluctuation de la bande passante, il est important de noter que la difficulté de modéliser cet effet prend source dans la conception même des simulations dans SimGrid. Comme mentionné dans le Chapitre 4, sOMP n'effectue pas de calculs : on résume l'exécution des instructions arithmétiques d'une tâche par le temps obtenu via les traces TiKKi. Or pour modéliser les effets de fluctuation de la bande passante, il est nécessaire de simuler l'exécution des instructions en temps réel afin de connaître la bande passante disponible (en temps réel aussi) et en tenir compte lors de l'exécution simultanée d'autres tâches. Bien que SimGrid offre la possibilité de simuler les calculs en fonction de la puissance théorique des *hosts*, la nature *non cycle-accurate* de ce framework impose un changement même dans la conception de ses simulations.

Ensuite, comme explicité dans la discussion du Chapitre 3, la simulation d'autres types d'architectures peut nécessiter une approche différente dans la modélisation des interconnexions des composants de la plateforme. Dans le cas des systèmes à mémoire hétérogènes plus particulièrement, la prise en charge des interactions RAM/CPU et RAM/GPU est essentielle pour obtenir des simulations fiables. D'autres types d'architectures peuvent nécessiter l'implémentation de grilles d'interconnexion à la place des réseaux implémentés dans l'approche de cette thèse, ou encore la considération des interactions RAM/CPU et RAM/réseau.

Enfin et dans un contexte plus large, ce travail cible la simulation des applications OpenMP à base de tâches. Une évolution logique serait donc la simulation d'autres paradigmes de programmation OpenMP, notamment les boucles parallèles, ce qui nécessitera une approche différente pour la collecte des traces et la simulation dans sOMP. Atteindre cet objectif ouvrira finalement la porte à la simulation du parallélisme hybride, c'est-à-dire combinant MPI et OpenMP, ce qui peut être effectué en conjonction avec sMPI, le simulateur des applications MPI déjà implémenté dans le framework SimGrid, pour déboucher sur la simulation MPI+OpenMP+CUDA, qui impliquera donc la prise en compte de toutes les interactions RAM/CPU, RAM/GPU et RAM/réseau en même temps.

Bibliographie

[1] T. Fisher, "Intel chipset drivers 10.1.18793." `https://www.lifewire.com/intel-chipset-drivers-2619202`, 2021.

[2] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[3] E. E. Johnson, "Completing an mimd multiprocessor taxonomy," *ACM SIGARCH Computer Architecture News*, vol. 16, no. 3, pp. 44–47, 1988.

[4] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture : a hardware/software approach*. Gulf Professional Publishing, 1999.

[5] N. Manchanda and K. Anand, "Non-uniform memory access (numa)," *New York University*, vol. 4, 2010.

[6] R. Shannon, "Introduction to the art and science of simulation," in *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*, vol. 1, pp. 7–14 vol.1, 1998.

[7] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures," *Concurrency and Computation : Practice and Experience*, vol. 27, no. 16, pp. 4075–4090, 2015.

[8] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are Static Schedules so Bad ? A Case Study on Cholesky Factorization," in *International Parallel & Distributed Processing Symposium, IPDPS'16*, 2016.

[9] T. Harmer, "Intel skylake-x core i7-7800x review – the true hedt entry-point ?." `https://www.vortez.net/articles_image/36664.html`, 2017.

[10] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proceedings of the 11th annual international symposium on Computer architecture*, pp. 348–354, 1984.

[11] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[12] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE*

Transactions on Parallel and Distributed Systems, vol. 20, no. 3, pp. 404–418, 2009.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk : An efficient multithreaded runtime system," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, 1996.

[14] F. Galilee, G. Cavalheiro, J.-L. Roch, and M. Doreille, "Athapascan-1 : On-line building data flow graph in a parallel language," in *Parallel Architectures and Compilation Techniques*, oct 1998.

[15] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations," in *Europar 2010 - 16th International Euro-Par Conference on Parallel Processing* (P. D'Ambra, M. R. Guarracino, and D. Talia, eds.), vol. 6272 of *Lecture Notes in Computer Science*, (Ischia-Naples, Italy), pp. 235–246, Springer, Aug. 2010.

[16] C. Augonnet, S. Thibault, R. Namyst, and P.-A. 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, pp. 187–198, Feb. 2011.

[17] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proceedings of the 15th Euro-Par Conference*, (Delft, The Netherlands), Aug. 2009.

[18] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi : A runtime system for data-flow task programming on heterogeneous architectures," in *Parallel & Distributed Processing (IPDPS)*, IEEE, 2013.

[19] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[20] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Productive cluster programming with OmpSs," in *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par '11, 2011.

[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10 : An object-oriented approach to non-uniform cluster computing," *SIGPLAN Notices*, vol. 40, pp. 519–538, Oct. 2005.

[22] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI : A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-Processors," in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, 2007.

[23] G. Zheng, G. Kakulapati, and L. V. Kalé, "Bigsim : A parallel simulator for performance prediction of extremely large parallel machines," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, p. 78, IEEE, 2004.

[24] C. Engelmann, "Scaling to a million cores and beyond : Using light-weight simulation to understand the challenges ahead on the road to exascale," *Future Generation Computer Systems*, vol. 30, pp. 59 – 65, 2014.

[25] S. Girona and J. Labarta, "Sensitivity of performance prediction of message passing programs," *The Journal of Supercomputing*, vol. 17, 2000.

[26] P. Czarnul, J. Kuchta, M. Matuszek, J. Proficz, P. Rościszewski, M. Wójcik, and J. Szymański, "Merpsys : An environment for simulation of parallel application execution on large scale hpc systems," *Simulation Modelling Practice and Theory*, vol. 77, pp. 124 – 140, 2017.

[27] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 2899–2917, June 2014.

[28] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim : a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software : Practice and Experience*, vol. 41, no. 1, 2011.

[29] K. S. U. Kliazovich Dzmitry, Bouvry Pascal, "Greencloud : a packet-level simulator of energy-aware cloud computing data centers," *The Journal of Supercomputing*, 2012.

[30] R. Aversa, B. Di Martino, M. Rak, S. Venticinque, and U. Villano, "Performance prediction through simulation of a hybrid MPI/OpenMP application," *Parallel Computing*, vol. 31, no. 10, 2005. OpenMP.

[31] B. Haugen, J. Kurzak, A. YarKhan, P. Luszczek, and J. Dongarra, "Parallel simulation of superscalar scheduling," in *2014 43rd International Conference on Parallel Processing*, pp. 121–130, 2014.

[32] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *International Symposium on Performance Analysis of Systems and Software*, 2011.

[33] S. Shudler, A. Calotoiu, T. Hoefler, and F. Wolf, "Isoefficiency in practice : Configuring and understanding the performance of task-based applications," *SIGPLAN Notices*, vol. 52, no. 8, 2017.

[34] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau, "Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers," in *The 21st IEEE International Conference on Parallel and Distributed Systems*, (Melbourne, Australia), Dec. 2015.

[35] J. Tao, M. Schulz, and W. Karl, "Simulation as a tool for optimizing memory accesses on NUMA machines," *Performance Evaluation*, vol. 60, no. 1, 2005.

[36] F. Heinrich, *Modeling, Prediction and Optimization of Energy Consumption of MPI Applications using SimGrid.* Theses, Université Grenoble Alpes, May 2019.

[37] N. Denoyelle, B. Goglin, A. Ilic, E. Jeannot, and L. Sousa, "Modeling non-uniform memory access on large compute nodes with the cache-aware roofline model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 6, 2019.

[38] B. Haugen, *Performance analysis and modeling of task-based runtimes.* PhD thesis, 2016.

[39] A. Mohammed, A. Eleliemy, F. M. Ciorba, F. Kasielke, and I. Banicescu, "Experimental verification and analysis of dynamic loop scheduling in scientific applications," in *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE, 2018.

[40] Y. Liu, Y. Zhu, X. Li, Z. Ni, T. Liu, Y. Chen, and J. Wu, "SimNUMA : Simulating NUMA-Architecture Multiprocessor Systems Efficiently," in *2013 International Conference on Parallel and Distributed Systems*, Dec 2013.

[41] M. Slimane and L. Sekhri, "HLSMN : High Level Multicore NUMA Simulator," *Electrotehnica, Electronica, Automatica*, vol. 65, no. 3, 2017.

[42] P. Virouleau, *Etude et amélioration de l'exploitation des architectures NUMA à travers des supports exécutifs.* Theses, Université Grenoble Alpes, June 2018.

[43] M.-E. Frincu, M. Quinson, and F. Suter, "Handling Very Large Platforms with the New SimGrid Platform Description Formalism," Technical Report RT-0348, INRIA, 2008.

[44] P. Velho and A. Legrand, "Accuracy Study and Improvement of Network Simulation in the SimGrid Framework," in *SIMUTools'09, 2nd International Conference on Simulation Tools and Techniques*, (Rome, Italy), Mar. 2009.

[45] M. Geier, L. Nussbaum, and M. Quinson, "On the convergence of experimental methodologies for distributed systems : Where do we stand ?," in *WATERS-4th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2013.

[46] K. Fujiwara and H. Casanova, "Speed and accuracy of network simulation in the simgrid framework," in *Proceedings of the 2nd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '07, (Brussels, BEL), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[47] S. Linux, "Optimizing linux for amd epyc 7002 series processors with suse linux enterprise 15 sp1." `https://documentation.suse.com/sbp/all/html/SBP-AMD-EPYC-2-SLES15SP1/index.html#fig-epyc_topology_highlevel`, 2019.

[48] P. Kennedy, "Amd epyc 7000 series architecture overview for non-ce or ee majors." `https://www.servethehome.com/amd-epyc-7000-series-architecture-overview-non-ce-ee-majors/`, 2017.

[49] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[50] J. Treibig, G. Hager, and G. Wellein, "likwid-bench : An extensible micro-benchmarking platform for x86 multicore compute nodes," in *Tools for High Performance Computing 2011*, pp. 27–36, 2012.

[51] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "HWLoc : a Generic Framework for Managing Hardware Affinities in HPC Applications," in *International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, (Pisa, Italia), pp. 180–186, Feb. 2010.

[52] P. Virouleau, A. Roussel, F. Broquedis, T. Gautier, F. Rastello, and J.-M. Gratien, "Description, implementation and evaluation of an affinity clause for task directives," in *International Workshop on OpenMP*, pp. 61–73, Springer, 2016.

[53] J. C. de Kergommeaux, C. Guilloud, and B. de Oliveira Stein, "Flexible performance debugging of parallel and distributed applications," in *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference,*, vol. 2790 of *Lecture Notes in Computer Science*, Springer, 2003.

[54] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT : An OpenMP Tools Application Programming Interface for Performance Analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, (Berlin, Heidelberg), pp. 171–185, Springer, 2013.

[55] OpenMP Architecture Review Board, "Openmp application programming interface - version 5.0." `https://www.openmp.org`, 2018.

[56] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[57] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, "Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite," in *International Workshop on OpenMP*, Springer, 2014.

[58] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Lapack working note 191 : A class of parallel tiled linear algebra algorithms for multicore rarchitectures," 2007.

[59] I. Daoudi, P. Virouleau, T. Gautier, S. Thibault, and O. Aumage, "sOMP : Simulating OpenMP Task-Based Applications with NUMA Effects," in *IWOMP 2020 - 16th International Workshop on OpenMP*, Sept. 2020.

[60] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *SC '12 : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2012.

[61] H. Casanova, "Modeling large-scale platforms for the analysis and the simulation of scheduling strategies," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 170–, April 2004.

[62] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, "Locality-aware task scheduling and data distribution for openmp programs on numa systems and manycore processors," *Scientific Programming*, vol. 2015, 2015.