Optimisation des allocations de données pour des applications du Calcul Haute Performance sur une architecture à mémoires hétérogènes

Hugo Brunie

THÈSE PRÉSENTÉE
POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE
SPÉCIALITÉ : INFORMATIQUE

Par **Hugo Brunie**

Optimisation des allocations de données pour des applications du Calcul Haute Performance sur une architecture à mémoires hétérogènes

---

Sous la direction de : **Denis BARTHOU**, Bordeaux INP
Co-directeur : **Patrick CARRIBAULT**, CEA

Soutenue le 28 janvier 2019

Membres du jury :

Gaël THOMAS,	*Professeur*, Telecom SudParis	Rapporteur
Philippe CLAUSS,	*Professeur*, Université de Strasbourg	Rapporteur
Emmanuel JEANNOT,	*DR*, INRIA Bordeaux Sud-Ouest	Président du jury
Pascale ROSSE-LAURENT,	*Experte Architecte*, ATOS/Bull	Examinatrice
Philippe THIERRY,	*Ingénieur Principal*, INTEL	Examinateur
Julien JAEGER,	*Ingénieur Chercheur*, CEA	Examinateur

– 2019 –

Mots-clés Profilage, Allocation de données, Memoires hétérogènes, Optimisation combinatoire

Résumé Le Calcul Haute Performance, regroupant l'ensemble des acteurs responsables de l'amélioration des performances de calcul des applications scientifiques sur supercalculateurs, s'est donné pour objectif d'atteindre des performances exaflopiques. Cette course à la performance se caractérise aujourd'hui par la fabrication de machines hétérogènes dans lesquelles chaque composant est spécialisé. Parmi ces composants, les mémoires du système se spécialisent, et la tendance va vers une architecture composée de plusieurs mémoires aux caractéristiques complémentaires. La question se pose alors de l'utilisation de ces nouvelles machines dont la performance pratique dépend du placement des données de l'application sur les différentes mémoires.

Dans cette thèse, nous avons développé une formulation du problème d'allocation de donnée sur une Architecture à Mémoires Hétérogènes. Dans cette formulation, nous avons fait apparaître le bénéfice que pourrait apporter une analyse temporelle du problème, parce que de nombreux travaux reposaient uniquement sur une approche spatiale. À partir de cette formulation, nous avons développé un outil de profilage hors ligne pour approximer les coefficients de la fonction objective afin de résoudre le problème d'allocation et d'optimiser l'allocation des données sur une architecture composée deux de mémoires principales aux caractéristiques complémentaires. Afin de réduire la quantité de modifications nécessaires pour prendre en compte la stratégie d'allocation recommandée par notre boîte à outils, nous avons développé un outil capable de rediriger automatiquement les allocations de données à partir d'un minimum d'instrumentation dans le code source.

Les gains de performances obtenus sur des mini-applications représentatives des applications scientifiques codées par la communauté permet d'affirmer qu'une allocation intelligente des données est nécessaire pour bénéficier pleinement de ressources mémoires hétérogènes. Sur certaines tailles de problèmes, le gain entre un placement naïf est une allocation instruite peut atteindre un facteur $\times 3.75$.

Laboratoire d'accueil CEA, DAM, DIF, Bruyères-le-Châtel, F-91297 Arpajon

Title Data Allocation Optimisation for High Performance Computing Application on Heterogeneous Architecture

Keywords Data allocation, Memory management, Heterogeneous Memory Architecture, Profiling, Combinatorial optimization

Abstract High Performance Computing, which brings together all the players responsible for improving the computing performance of scientific applications on supercomputers, aims to achieve exaflopic performance. This race for performance is today characterized by the manufacture of heterogeneous machines in which each component is specialized. Among these components, system memories specialize too, and the trend is towards an architecture composed of several memories with complementary characteristics. The question arises then of these new machines use whose practical performance depends on the application data placement on the different memories. Compromising code update against performance is challenging.

In this thesis, we have developed a data allocation on Heterogeneous Memory Architecture problem formulation. In this formulation, we have shown the benefit of a temporal analysis of the problem, because many studies were based solely on a spatial approach this result highlight their weakness. From this formulation, we developed an offline profiling tool to approximate the coefficients of the objective function in order to solve the allocation problem and optimize the allocation of data on a composite architecture composed of two main memories with complementary characteristics. In order to reduce the amount of code changes needed to execute an application according to our toolbox recommended allocation strategy, we have developed a tool that can automatically redirect data allocations from a minimum source code instrumentation.

The performance gains obtained on mini-applications representative of the scientific applications coded by the community make it possible to assert that intelligent data allocation is necessary to fully benefit from heterogeneous memory resources. On some problem sizes, the gain between a naive data placement strategy, and an educated data allocation one, can reach up to $\times 3.75$ speedup.

Remerciement

Je remercie chaleureusement mes directeurs de thèse Denis et Patrick, ainsi que mon encadrant Julien, pour tout ce qu'ils m'ont enseigné. Merci aussi à Marc pour ses questions pertinentes qui m'auront préparé à la soutenance.

Je remercie aussi très chaleureusement mes amis et collègues du CEA pour les discussions enrichissantes et le soutien : Guillaume, Éloïse, Nestor, Arthur, THugo, Rémi, Adrien, Hoby, Gautier, Sébastien, Thomas, Julien, Antoine et Jean-Baptiste. De même, je tiens à remercier mes amis et collègues de l'INRIA : Pierre, Adrien, Emmanuelle, Philippe, Léo, Chiheb, Corentin, Romain, Idriss et Maël. Je tiens particulièrement à remercier Emmanuelle, qui m'a guidé dans mon stage de fin d'étude, dans la fin de ma thèse, et dans la recherche d'un post-doc. Je remercie enfin mes amis de l'école qui sont pour certains restés à l'INRIA ! Grégoire, Aurélien, Gaëlle, Gaëtan, Guillaume et Guillaume, Louis, Gilles, Thibault, Simon, Pierre, Florian et Yoan. Je remercie aussi Nicolas D. pour les conversations qu'on a pu avoir sur la zone commune à nos sujets de thèses.

Un grand merci à la team escalade : Rémi, Adrien, Arthur, des vacances à Mallorque je garde un souvenir impérissable, des vacances en Grèce je garde un souvenir inoubliable ! Je remercie particulièrement Rémi pour m'avoir mis à la grimpe et au Viet Vo Dao et je suis très heureux qu'il continue de grimper sans crainte ! Un grand merci à Thugo et Arthur, c'est dans la difficulté qu'on se sert le plus les coudes.

Merci à mes frères Nicolas et Jonas qui m'ont soutenu et m'ont permis de garder l'esprit ouvert. Merci à mes parents, Maddy et Bernard, malgré votre incompréhension du sujet et des difficultés que je pouvais avoir, vous étiez toujours présent pour me pousser à continuer, mais aussi pour discuter d'autres choses et m'aérer l'esprit.

Je remercie mes ami.e.s nîmois, Steffi, Manon et Sam, les festivals de musique permettent de bien décompresser. Je remercie mes amis de LA et Villeneuve de m'avoir supporté (surtout Pierre lors de la coloc) : Pierre, Matthieu, Matthias, Florent, Linda, Lucas, Kévin, Arthur. J'ai parfois eu la mauvaise idée de refuser une sortie ski sous la pression de la rédaction, mais vous avez su passer l'éponge. Merci à mes amis d'enfance pour n'être finalement jamais très loin : Cyril, Benoît, Chloé, Benjamin, Victor, Nicolas.

Je remercie particulièrement Pierre et Pierre pour m'avoir corrigé lors de la dernière pré-soutenance.

Cette thèse, comme la grande majorité des thèses de ce que j'ai pu entendre, s'est faite dans la joie et la souffrance. Mes directeurs de thèse, Denis et Patrick, m'ont apporté un éclairage inépuisable et indispensable. C'est à la lumière de leurs lanternes que j'ai pu mener cette thèse à bon port. Il est difficile de rester concentré sur un sujet précis durant trois ans, et je ne remercierai jamais assez Julien, Patrick et Denis pour leur encadrement.

Enfin et surtout, je remercie Philippe Clauss, Gaël Thomas, Pascale Rossé-Laurent et Philippe Thierry, pour leurs remarques et critiques pertinentes.

Table des matières

Table des figures

Liste des tableaux

Introduction

Depuis les débuts de l'informatique, cette discipline a toujours été partagée entre deux penchants distincts. D'une part, l'informatique comme science fondamentale de l'information, développant des algorithmes et calculant des complexités dans des domaines aussi vastes que : la théorie des graphes, celle des automates et langages, la sémantique des processus, la modélisation du parallélisme, la vérification des systèmes informatiques, la théorie combinatoire des groupes, etc. D'autre part, l'informatique comme outil de calcul performant avec des machines fournissant toujours plus de puissance, mais dont la complexité croît exponentiellement. De cette complexité, est naît une science ingénieuse de l'informatique, celle de la recherche d'une utilisation optimale de l'architecture sous-jacente.

Cette science cherche à réduire le temps d'exécution des applications scientifiques, et à réduire la consommation énergétique des machines exécutant ces applications. Pour cela le matériel doit être construit pour maximiser la quantité d'opérations qu'il est possible d'effectuer en une seconde, et les couches logicielles cherchent à utiliser aux mieux le matériel tout en conservant une abstraction indispensable au développement productif de code de calcul.

Dans cette course à la performance, l'évolution des machines tend vers toujours plus d'hétérogénéité, celle-ci se développant à plusieurs niveaux. Elle se développe au niveau des nœuds de calcul avec la juxtaposition d'accélérateurs (General Purpose Graphical Processing Unit (GPGPU), FPGA, etc.) et de processeurs (ou Central Processing Unit ou processeur (CPU)) devant se partager le travail à effectuer. Elle se développe aussi au niveau du système mémoire, avec une hiérarchisation des mémoires caches en plusieurs niveaux, et la cohabitation de mémoires aux caractéristiques variées au-delà : mémoires principales, disque externe, bande magnétique, etc.). Cette hétérogénéité pose de nombreux problèmes, comme la cohérence des données à travers tout le système.

Dans cette thèse nous avons cherché à caractériser les différentes mémoires qu'on peut trouver sur une Architecture à Mémoires Hétérogènes (AMH), afin d'être capable d'optimiser les performances des applications scientifiques sur ce type de machines sans entreprendre de grandes modifications des codes sources.

Dans ce but, nous avons développé une formulation du problème d'allocation de donnée sur une Architecture à Mémoires Hétérogènes. Dans cette formulation, nous avons fait apparaître le bénéfice que pourrait apporter une analyse temporelle du problème, parce que de nombreux travaux reposaient uniquement sur une approche spatiale. À partir de cette formulation, nous avons développé un outil de profilage hors ligne pour approximer les coefficients de la fonction objective afin de résoudre le problème d'allocation et d'optimiser l'allocation des données sur une architecture composée deux de mémoires principales aux caractéristiques complémentaires. Afin de réduire la quantité de modifications nécessaires pour prendre en compte la stratégie d'allocation recommandée par notre boîte à outils,

nous avons développé un outil capable de rediriger automatiquement les allocations de données à partir d'un minimum d'instrumentation dans le code source.

Les gains de performances obtenus sur des mini-applications représentatives des applications scientifiques codées par la communauté permet d'affirmer qu'une allocation intelligente des données est nécessaire pour bénéficier pleinement de ressources mémoires hétérogènes. Sur certaines tailles de problèmes, le gain entre un placement naïf est une allocation instruite peut atteindre un facteur $\times 3.75$.

Ce document est organisé en 6 chapitres. Le premier introduit la problématique après avoir détaillé le contexte du sujet de thèse. Le second propose une vision ordonnée de l'état de l'art au commencement de ma thèse. Les trois chapitres suivant détaillent les contributions de mes travaux. Enfin, le sixième chapitre conclut et développent des perspectives pour des travaux de recherches suivant ceux de cette thèse.

Chapitre 1

Contexte

Les supercalculateurs peuvent effectuer plusieurs dizaines de millions de milliard d'opérations par seconde, mais leur fabrication et leur entretien se comptent en millions de dollars/euros. Cependant, pour des entreprises pouvant le financer, ces machines permettent de remplacer des expériences réelles encore plus coûteuses par une simulation logicielle nécessitant une grande puissance de calcul. Au-delà des entreprises, les supercalculateurs sont aussi utilisés dans certains centres de recherche. Le CERN en Suisse utilise plus de 10 000 serveurs, 90 000 cœurs de processeurs, entreposés dans des salles adaptées (cf figure 1.1.1), pour stocker et distribuer sur un réseau mondial les 25 Petaoctets [1] de données par an qui proviennent des collisions de particules dans le Grand Collisionneur de Hadrons (LHC). Le supercalculateur MIRA, aux États-Unis, permet à des chercheurs d'exécuter un logiciel qu'ils ont développé afin de prédire, avec une grande précision, le résultat des collisions qui ont lieu au LHC. La comparaison de ces prédictions, basée sur le Modèle Standard de la physique des particules [sta,], avec les données fournies par les expériences effectuées au LHC permettront de mettre en évidence les défauts de la théorie actuelle et de développer ainsi notre compréhension de l'Univers.

1. 1 petaoctet = 10^{15} octets

Aujourd'hui le Calcul Haute Performance (HPC), c'est-à-dire l'ensemble des acteurs gravitant autour de la conception, le développement et l'utilisation des supercalculateurs, représente une part importante des investissements dans la recherche et le développement dans le monde [Advisory and Services, 2018]. Ces investissements ont pour but de développer un supercalculateur exascale d'ici à 2022. Cette course mondiale à l'augmentation de la puissance de calcul des supercalculateurs pour passer de l'aire petaflopique (10^{15} Floating Point Operation (FLOP)/seconde) à l'aire exaflopique (10^{18} FLOP/seconde) est la conséquence directe d'un des enjeux du HPC : la réduction du temps d'exécution des applications scientifiques. Sachant que l'énergie dont a besoin un supercalculateur est un des principaux facteurs de son coût d'entretien [Ludwig, 2012], il n'est pas envisageable de se contenter d'utiliser les processeurs des supercalculateurs petaflopiques pour assembler un supercalculateur exascale, car un tel supercalculateur dépenserait 200MW, soit environ 175 millions dollars par an (pour 0.1$ par kWh). Cet objectif induit donc un second enjeu majeur pour le HPC : maximiser le rapport puissance de calcul par unité d'énergie dépensée. Dans ce manuscrit nous nous intéressons essentiellement au premier objectif majeur : la réduction du temps d'exécution.

Pour répondre à ces enjeux il existe trois leviers : le matériel, les couches logicielles de support exécutifs, et l'algorithme/le code de l'application elle-même. C'est en optimisant la synergie entre ces trois leviers que le temps d'exécution des applications peut être réduit.

1.1 Le Calcul Haute Performance

Figure 1.1.1 – Baie de serveurs au CERN [cer,]

Parmi les 3 grandes catégories d'acteurs du HPC, les fabricants de matériels sont ceux qui fournissent les processeurs, les réseaux d'interconnexion et l'infrastructure des supercalculateurs. Les développeurs de la pile logicielle fournissent des compilateurs pour

traduire le code d'un langage humainement compréhensible au langage machine, des systèmes d'exploitations pour, entre autre, abstraire la complexité de la gestion mémoire et de l'ordonnancement des processus, des logiciels de support exécutifs pour simplifier l'écriture de code. Dans la pile logicielle on retrouve aussi des outils d'aide à la correction d'erreurs dans le code, et des outils d'aide à la mise en évidence des régions du code qui sous utilisent les ressources matérielles disponibles. Enfin, les développeurs d'applications scientifiques développent : algorithmes scientifiques, applications scientifiques, benchmarks, mini-applications. Nous allons voir comment ces acteurs s'organisent pour réduire le temps d'exécution des applications. Nous verrons rapidement les grandes évolutions matérielles et comment les couches logicielles s'organisent pour en tirer profit. La figure 1.1.2 représente une vue simplifiée des différentes strates de la pile logicielle et matérielle intervenant dans l'exécution d'une application sur une machine.

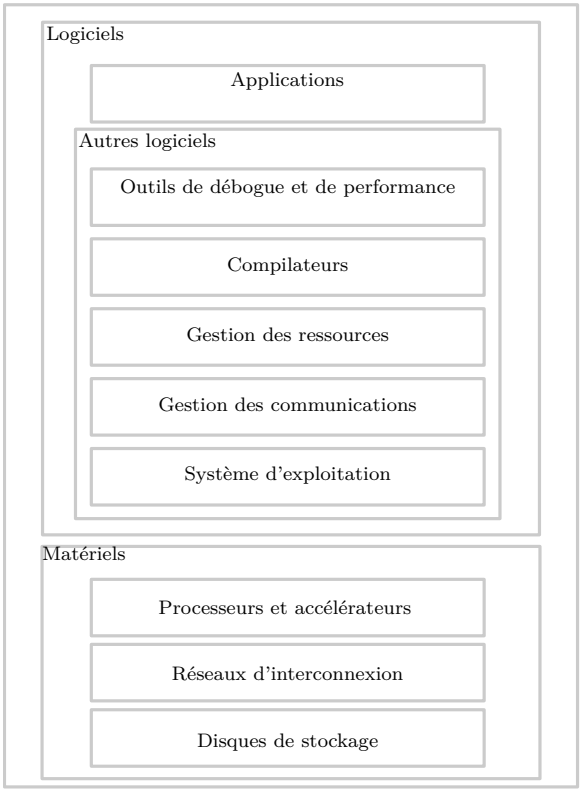


FIGURE 1.1.2 – Pile logicielle et matérielle

1.1.1 Matériel

Les supercalculateurs sont composés de centaines de nœuds de calcul, eux-mêmes constitués d'un ensemble de processeurs et/ou accélérateurs. Les processeurs et accélérateurs sont reliés entre eux, au sein d'un même nœud et entre différents nœuds, par des réseaux rapides permettant aux processus de communiquer entre eux. Les applications scientifiques s'exécutent en parallèle en tentant d'exploiter toutes les ressources d'un supercalculateur.

La puissance de calcul d'une machine, et par assemblage du supercalculateur, peut être caractérisée par la quantité théorique, ou pratique, d'opérations qu'une machine est capable d'effectuer en une seconde. Qu'elle soit théorique ou pratique, cette quantité dépend de la fréquence, des fonctionnalités et du nombre de cœurs et d'unités de calcul par cœur qui composent le processeur étudié. La puissance théorique s'obtient en multipliant tous les facteurs entre eux, comme si les unités de calcul fonctionnaient à plein régime. Contrairement à celle-ci, la puissance de calcul pratique d'un processeur se mesure en exécutant un benchmark stressant beaucoup toutes les unités de calcul, comme ceux de la suite de benchmarks LINPACK [Dongarra, 1988, Dongarra et al., 2003]. Bien que LINPACK soit critiquée et que de nouveaux benchmarks plus représentatifs des applications sont développés [Heroux et al., 2013, Heroux and Dongarra, 2013, Dongarra et al., 2015], elle continue d'être utilisée aujourd'hui pour comparer les supercalculateurs entre eux et définir la liste semestrielle des 500 supercalculateurs possédant les meilleurs résultats (le TOP500 [Strohmaier et al., b, Strohmaier, 2006]). Pour mieux tenir compte des objectifs de limitation en énergie dépensée, le GREEN500 [Strohmaier et al., a] classe les 500 premiers supercalculateurs qui ont le meilleur rapport FLOP/s/Watt.

Les techniques utilisées par les fabricants de matériels pour augmenter la puissance de calcul des processeurs ont évoluées régulièrement. Au cours de la fin du XX^{e} siècle, il était encore possible de doubler régulièrement la puissance de calcul en se concentrant sur l'augmentation de la fréquence du processeur et de son nombre de transistors. Dans les années 60, Gordon Moore, co-fondateur d'Intel Corporation, a établi une loi [Moore,] prédisant (ou incitant) cette augmentation exponentielle du nombre de transistors. L'énergie générée par les circuits électroniques dépendant, entre autre, du carré de la fréquence de fonctionnement, la dissipation de cette énergie est un réel obstacle à l'augmentation de la fréquence.

La barrière énergétique empêchant l'augmentation continue de la fréquence d'exécution des processeurs d'une génération à l'autre, les techniques de fabrication se sont concentrées sur le traitement en paralléle du calcul. Le parallélisme consiste à traiter des instructions en parallèle, c'est-à-dire "en même temps". Ce parallélisme dans le traitement des instructions par un cœur et entre différents cœurs d'une même puce entraîne des changements au niveau logiciel : il ne suffit plus à un code de calcul de s'exécuter sur une machine plus récente pour voir ses performances augmenter. Un effort d'écriture de code est nécessaire pour paralléliser l'algorithme, et/ou le code, afin d'exploiter le potentiel de performance du processeur.

Selon la taxonomie de Flynn [Flynn, 1972], la majorité des architectures de machine parallèle disponibles aujourd'hui font partie de la catégorie « multiples flux d'instructions, multiples flux de données » (MIMD). Ces architectures peuvent être déclinées en deux variantes principales : les MIMD à mémoires partagées et celles à mémoires distribuées. Dans le premier cas, les unités de calcul ont accès à la mémoire sous la forme logicielle d'un espace d'adressage global. Tout modification de l'état d'une case mémoire est vu par les autres unités de calcul « instantanément ». Les unités de calcul peuvent donc communiquer via la mémoire globale, partagée. Dans le second cas, chaque unité de calcul possède sa propre mémoire et son propre système d'exploitation. Ce second cas de figure nécessite un

middleware pour la synchronisation et la communication.

En pratique, on peut observer un écart entre les performances théoriques (matérielles) et les performances pratiques (logicielles) mesurées lors de l'exécution d'une application scientifique. Alors que la puissance de calcul d'un processeur est liée à sa fréquence de fonctionnement, à ses fonctionnalités et au nombre d'unités de calcul qui le composent, la quantité d'opérations par secondes effectuées par l'application se situe systématiquement un ou deux ordres de magnitude en deçà. Cet écart est dû notamment à la difficulté qu'ont les processeurs, malgré les systèmes de pré-lecture et de ré-ordonnancement des instructions (Out Of Order (OoO)), à masquer par du calcul les temps d'accès aux données (lecture/écriture) se trouvant en mémoire.

L'écart de performance observé est notamment dû aux mécanismes du système de traitement des requêtes mémoires, puisque les temps d'accès ne sont pas complètement masqués. Les performances sont d'autant plus dégradées que la latence d'accès à la mémoire est longue. Cette latence peut être due à de nombreuses combinaisons possibles de mécanismes matériels et/ou logiciels : localité des données avec différents niveaux de mémoires caches et/ou différents nœuds au sein d'une architecture NUMA (cc-NUMA ou nc-NUMA). Ou bien dans le cas d'une machine composée de plusieurs processeurs reliés entre eux sur ce qu'on appelle une machine NUMA, lorsqu'une unité de calcul émet une requête pour une donnée se trouvant sur la mémoire d'un autre processeur, la latence est plus longue que si l'accès avait été local.

Finalement, la nature des limitations en performance d'une région de code dépend de la nature des instructions qui s'enchaînent. Pour résumer, une région de code d'une application scientifique est limitée, soit par les performances de la mémoire hors puce (locale ou distante dans le cas NUMA), soit par les performances d'une des mémoires caches, soit par la vitesse de traitement des instructions par les unités de calcul.

1.1.2 Pile logicielle

Afin d'abstraire la complexité de la machine, différents logiciels s'entrelacent par couches successives. Une partie d'entre eux sont représentés de façon simplifiée sur la figure 1.1.2. Nous n'avons pas présenté les outils de débogage qui ne sont pas à proprement parler une couche d'abstraction de la machine pour le logiciel, mais qui représentent une couche essentielle de la pile logicielle dans l'aide à la programmation d'une application.

Le système d'exploitation permet d'abstraire la complexité de la machine (gestion mémoire, entrée/sortie, ordonnancement de processus, etc.). L'abstraction de la complexité matérielle est d'autant plus essentielle en HPC car les codes des applications sont écrits par des scientifiques qui ne sont pas experts informatiques, et les machines parallèles sont de plus en plus complexes. Pour chaque niveau de granularité du parallélisme, un corps de logiciel différent intervient. Les compilateurs sont historiquement orientés vers le parallélisme à grain fin, qu'on nomme parallélisme à mémoire partagée : boucles itératives parallélisées sur plusieurs cœurs ("standard" OpenMP), vectorisation. Au-dessus, à plus gros grain il existe ce qu'on nomme le parallélisme à mémoire distribuée. Au sein de ce paradigme, le "standard" le plus utilisé pour la gestion des communications inter-nœud est MPI (Message Passing Interface) : différentes implémentations existent et c'est au

programmeur d'exprimer le parallélisme. Pour faciliter l'écriture de programmes parallèles à grandes échelles, certaines bibliothèques (StarPU [Augonnet et al., 2009]) permettent d'exprimer le programme sous forme de tâches, avec leurs dépendances, et c'est le support exécutif qui se charge de trouver le parallélisme entre les tâches et d'exploiter les ressources matérielles en conséquence. Enfin le système d'exploitation permet d'abstraire la complexité de la machine (gestion mémoire, entrée/sortie, ordonnancement de processus, etc.). Bien qu'il reste indispensable, certaines recherches tendent à minimiser ses fonctionnalités pour mieux maîtriser son impact sur les performances.

1.1.3 Applications scientifiques

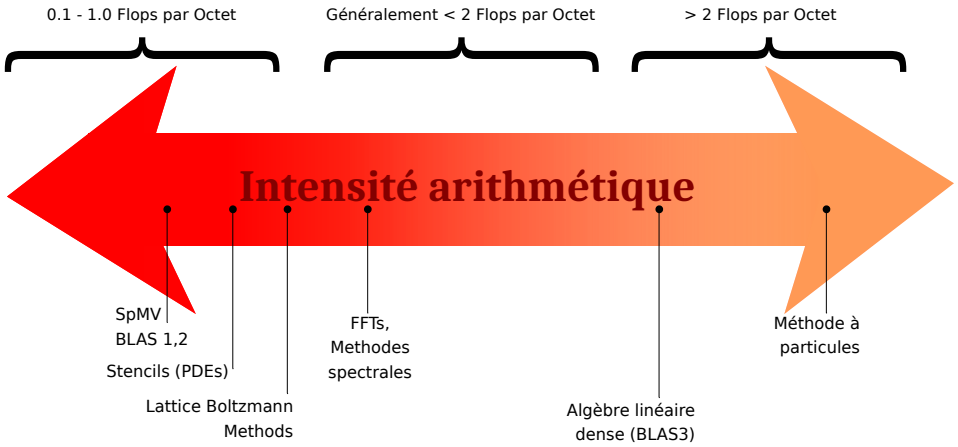


FIGURE 1.1.3 – Classification de quelques grandes catégories d'applications scientifiques selon leur intensité arithmétique [Williams et al.,]

Les applications scientifiques sont développées dans le but de résoudre des problèmes complexes. Il existe des applications de simulation numérique dans de nombreux domaines : climatique, génomique, systèmes biologiques, ingénierie, etc. Chaque domaine possède un but différent : certains cherchent à résoudre des équations dans le but de mieux comprendre certains phénomènes naturels, ou bien ces résolutions peuvent permettre de dimensionner une pièce industrielle, d'autres applications encore ont pour but de séquencer un génome et de déterminer la fonction et l'expression des gènes séquencés, etc.

Ces applications peuvent être caractérisées par la manière dont elles exploitent les ressources d'un processeur. Le « modèle en toit » (Roofline model) [Williams et al., 2009, Ilic et al., 2014] permet de caractériser les processeurs et les applications, et d'évaluer le fossé existant entre le temps d'exécution théorique et celui réellement pris par l'application. Ce modèle repose sur la notion d'intensité arithmétique qui est le ratio entre la quantité d'instruction d'opérations flottantes contenues dans le code de l'application et la quantité de données accédées (lectures et écritures confondues).

La figure 1.1.3 représente une classification de quelques grandes catégories d'applications scientifiques selon leur intensité arithmétique. Plus l'application se situe à gauche de l'axe et plus son temps d'exécution est tributaire des performances du système mémoire à

traiter les requêtes d'accès mémoire. Au contraire, plus elle se situe à droite et plus elle est dépendante du parallélisme des unités de calcul, de la fréquence du processeur, et de la capacité de celui-ci à masquer les temps d'attente des accès mémoires (mécanisme de ré-ordonnancement des instructions à la volée et de pré-lecture).

À partir de l'intensité arithmétique il est aussi possible de prédire la limite théorique des performances d'une application sur une machine. Pour cela il faut aussi calculer l'équilibre de la machine qui est le ratio du pic théorique d'opérations par seconde sur le pic théorique de la bande passante. En théorie selon ce modèle, si l'intensité arithmétique est plus faible que l'équilibre, l'application est limitée par la bande passante, sinon elle est limitée par le calcul. En pratique, le modèle est plus complexe : il existe un « toit » pour la bande passante de chaque niveau de cache, un pour les performances de calcul scalaire, un pour celles du calcul vectoriel, etc.

Si le ce model peut permettre de mettre en évidence le fait qu'une application ou région d'un code soit est limitée par la bande passante de tel niveau mémoire ou bien par les performances scalaire du processeur, celui-ci ne permet pas de mettre en évidence les données responsables de cette perte de performance (cf figure 1.1.4). De plus l'optimisation la plus rentable à effectuer ne saute pas toujours aux yeux au vue de l'étude avec ce modèle.

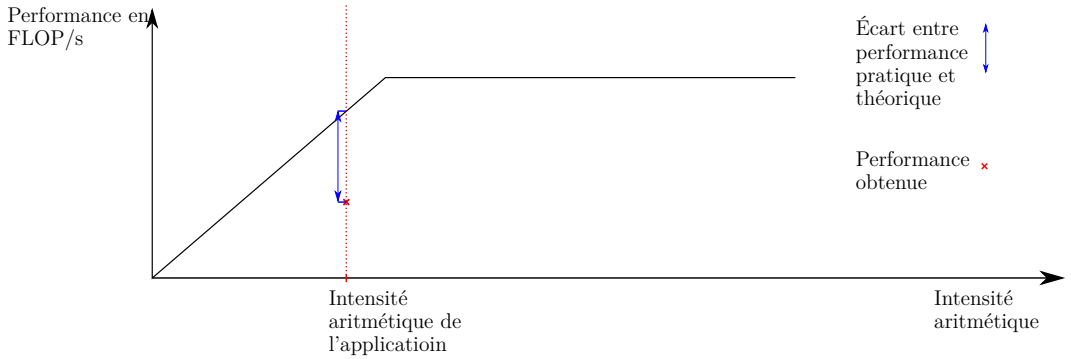


FIGURE 1.1.4 – Schéma du Roofline Model appliqué à une application et à une machine en particulier

De nombreuses applications pourraient avoir de meilleures performances avec un système mémoire traitant plus rapidement les requêtes mémoires, elles se situent à gauche du diagramme de la figure 1.1.3 ; on dit qu'elles sont *sensibles aux performances mémoires*. Avant l'avènement du parallélisme dans les architectures matérielles, certains prédisaient déjà que le fossé grandissant entre les performances des unités de calcul et les performances des systèmes mémoires aurait pour conséquence que de plus en plus d'applications scientifiques verraient leur performance limitée par la mémoire [Wulf and McKee, 1995]. La latence mémoire ne peut pas être réduite indéfiniment, parce qu'il y existe une limite physique au rapprochement de la mémoire et des unités de calcul. Cependant, le parallélisme des processeurs, leurs mécanismes de recouvrement de l'attente de donnée en mémoire par du calcul, et le parallélisme du système de traitement des requêtes mémoires,

permettent d'augmenter les performances des applications sensibles aux temps d'accès mémoire. Malgré ces efforts, la divergence entre le débit maximal d'opérations traitées par les unités de calcul d'un processeur et le débit de traitement de données par le système mémoire continue de croître. Cela entraine de plus en plus d'application à être limitée par les performances mémoires, que ce soit par les performances de latence mémoire, ou celle de bande passante mémoire, comme nous le détaillerons par la suite. C'est dans ce contexte qu'on observe l'émergence de nouvelles technologies mémoires et nous nous intéressons particulièrement à l'adaptation des applications sensibles aux performances mémoires à ces nouvelles architectures.

1.2 Influence des accès mémoires sur le temps d'exécution

Avant de décrire ces nouvelles technologies mémoires qui pourraient permettre de réduire le fossé entre les performances des unités de calcul et celles du système mémoire, nous allons d'abord approfondir la notion de sensibilité aux performances mémoires. Dans un premier temps les différentes caractéristiques du système physique de traitement des requêtes mémoires sont expliquées, puis les codes sensibles aux performances mémoires sont détaillés.

1.2.1 Caractéristiques matérielles du système mémoire

Une mémoire physique possède quatre caractéristiques : la latence, la bande passante, la capacité et la volatilité des données. La capacité mémoire théorique est donnée par le fabricant et peut être récupérée lors de l'exécution d'un programme en allant lire le fichier contenant ces informations dans le système de fichiers. Il ne faut pas confondre la capacité mémoire et l'usage mémoire. Alors que la première est propre à chaque machine, l'usage mémoire est propre à chaque application. L'usage mémoire dépend de la quantité de données allouées par l'application et il varie en fonction des allocations et libérations de données. Entre l'instant auquel une donnée est allouée, et l'instant auquel elle est libérée, on dit d'une donnée qu'elle est vivante. L'usage mémoire représente donc la somme des usages mémoires des données vivantes au même instant de l'exécution du programme. Le pic d'usage mémoire représente l'usage mémoire maximal atteint au cours d'une exécution.

La latence mémoire est le temps que met une requête mémoire pour être traitée de bout en bout. C'est-à-dire que la latence mémoire mesure le temps entre l'envoi d'une requête par une unité de calcul et le rapatriement de la donnée dans son registre. Durant ce temps la requête a été traduite d'une adresse virtuelle vers l'adresse physique correspondante par un cache matériel ou, si besoin, le système d'exploitation. Puis, le contrôleur mémoire accède à la donnée en empruntant le canal, le rang, le banc, la colonne et la rangée qui correspond à la donnée en mémoire physique. Ensuite la donnée est copiée de la mémoire principale dans les différents niveaux de cache, en fonction de la politique implémentée dans le matériel, puis dans le registre matériel de l'unité de calcul. Chaque composant matériel pouvant contenir une donnée (registre, mémoire cache, mémoire principale, disque dur externe, etc.) possède une latence dépendant de sa distance aux unités de calcul et de

la complexité logicielle et matérielle des algorithmes permettant de ramener la donnée.

Amener une donnée depuis une mémoire cache jusque dans un registre prend un ordre de grandeur de moins, en nombre de cycles, que de la ramener depuis la mémoire principale. C'est pourquoi de nombreux travaux de recherche ont porté et portent encore sur la réutilisation des données en mémoire cache. Il s'agit d'utiliser aux mieux la localité spatiale et temporelle des accès en lecture et en écriture du code de calcul [Wolf and Lam, 1991, Lam et al., 1991, Carr et al., 1994, Rivers et al., 1998, Ding and Zhong, 2003, Kowarschik and Weiß, 20 Albericio et al., 2013]. À partir de telle stratégie d'utilisation de donnée on pourrait en déduire que plus la taille du cache est grande meilleures seront les performances. Cependant, [Eklov et al., 2011] ont comparé les courbes de Cycle Per Instruction (CPI) et Miss Ratio Curve (MRC) de deux micros benchmarks de SPEC2006 [Henning, 2006] et ils ont ainsi mis en évidence que la MRC en fonction de la taille du cache n'est pas nécessairement corrélée à la courbe de CPI en fonction de la taille du cache.

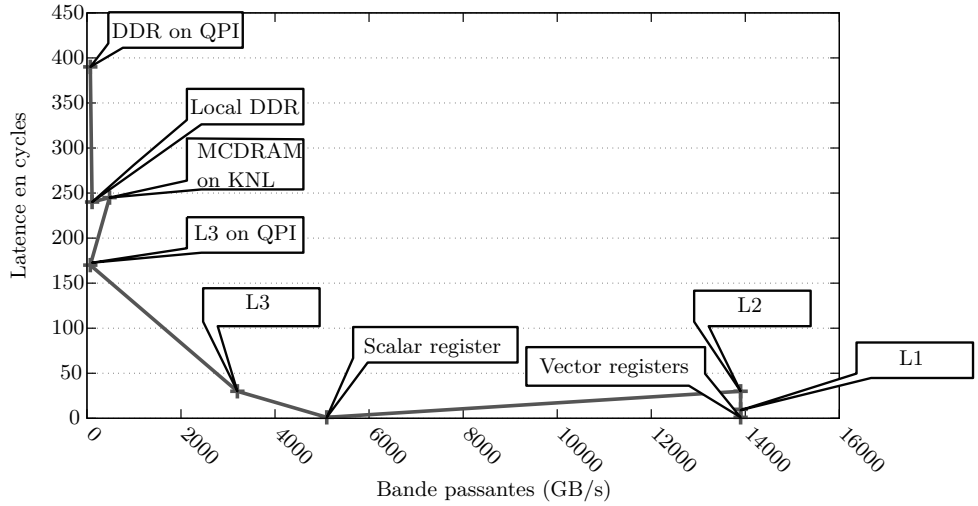


FIGURE 1.2.1 – Classification des différentes technologies de stockage de données en fonction de leur latence et bande passante.

Dans cet article, les auteurs expliquent que ce phénomène est dû au fait que la courbe MRC ne prend que les LLCmiss en compte et pas les déplacement de donnée depuis la mémoire vers le registre (fetch). Ce fetch a très bien pu être demandé par le pré-lecteur matériel. Il est donc possible, en augmentant la taille du cache, de faire travailler davantage le pré-lecteur. Si celui-ci ne se trompe pas trop, le nombre de LLCmiss devrait diminuer. Cependant si les données ne sont pas réutilisées une fois dans le cache, comme pour le benchmark STREAM par exemple, le nombre de LLCmiss en moins correspond directement au nombre de fetch en plus. Ainsi, le trafic mémoire reste le même et les performances globales ne sont pas améliorées pour autant ! Les auteurs justifient ainsi que le ratio de LLCmiss pris seul n'est pas suffisant pour évaluer l'impact de la taille du cache sur les performances d'une application. De plus, puisque le trafic mémoire est constitué des LLCmiss et des fetch, leurs expériences montrent aussi que le compte de LLCmiss seul ne suffit pas à évaluer le trafic mémoire, c'est à dire la bande passante mémoire consommée.

Nous reviendrons sur ce résultat dans la partie 4.1.1 du chapitre 4.

Le débit de données est lié à la latence par la loi de Little représentée sur la figure 1.2.1. La loi de Little prévoit une latence égale à l'inverse de la bande passante, ce résultat peut être observé sur la figure. Cependant, la latence ainsi représentée ne doit pas être confondue avec la latence moyenne d'un accès à la mémoire, qui dépend de la bande passante utilisée, c'est-à-dire du niveau de congestion du système mémoire, comme on le verra dans la partie suivante (1.2.2) sur le lien entre la latence et la bande passante.

1.2.2 Codes sensibles aux performances mémoires

Micro benchmarks sensibles aux performances mémoires

Comme expliqué dans la partie 1.1.3 le temps d'exécution de certains codes est principalement composé de temps d'attente des unités de calcul sur des données contenues en mémoire. C'est le cas notamment du benchmark STREAM [McCalpin, 1995] et du benchmark de « pointeur chassant » (PointerChasing), détaillés dans cette partie. Lorsqu'on observe une application à travers les données auxquelles elle accède, on dira qu'une donnée est sensible à la latence s'il y a une différence de performance importante en mettant la donnée sur une mémoire avec une meilleure latence mais une bande passante équivalente, et vice versa pour une donnée sensible à la bande passante.

Les codes de type PointerChasing sont connus comme étant sensibles à la latence, cette propriété est d'ailleurs exploitée pour mesurer les différents niveaux de latences machines [McVoy and Staelin, 1996]. Ces codes sont caractérisés par le fait que chaque accès mémoire, pour être traité, dépend de l'accès mémoire précédent, parce que l'adresse suivante accédée est contenue dans la case mémoire de l'accès précédent. Ainsi, si les accès sont suffisamment imprévisible, le pré-lecteur matériel ne peut pas transformer cette sensibilité à la latence en une sensibilité à la bande passante.

Les benchmarks de la suite lmbench [McVoy and Staelin, 1996] permettent de mesurer les performances de mouvement de données à travers tout le système, et notamment la latence et la bande passante de la mémoire principale, de manière similaire à l'outil Intel Memory Latency Checker. La figure 1.2.1 montre les résultats obtenus avec cet outil d'Intel sur un nœud possédant une architecture NUMA, composé de deux sockets chacune d'elle possédant une architecture Uniform Memory Access (UMA). On parle d'une architecture NUMA lorsque les accès à la mémoire (cache et/ou principale) ne sont pas uniforme selon qu'on accède à une mémoire proche (locale) ou lointaine (externe), il peut y avoir plusieurs niveaux dans une telle architecture. Chaque nœud de l'architecture NUMA de la machine testée contient un processeur Sandy Bridge. Les nœuds UMA de l'architecture NUMA sont reliés avec un lien INTEL Quick Path Interconnect (QPI). Sur la figure 1.2.1, on peut voir que la latence d'un accès mémoire sur le nœud distant est plus importante que sur le nœud local.

Le benchmark STREAM sert à évaluer les performances de bande passante mémoires des machines en effectuant des accès mémoires très nombreux et très fréquents, ce qui stresse le système de traitement des requêtes mémoires. La bande passante fournie par une mémoire est mesurée en exécutant ce code et en mesurant son temps d'exécution.

	nœud	
nœud	0	1
0	9.3	12.2
1	12.3	9.3

TABLE 1.2.1 – latence d'un accès mémoire selon la localité NUMA en nanosecondes, obtenues avec Intel Memory Latency Checker sur Sandy Bridge

On peut alors diviser la quantité de données utiles au calcul par le temps d'exécution. Il est composé de quatre noyaux : copy, scale, add et triad. Copy correspond à une copie d'un tableau dans un autre. Scale est une copie avec multiplication de chaque élément par un scalaire, celui-ci se retrouvant en registre, les performances de Copy et Scale sont similaires. Add correspond à l'addition de deux tableaux avec écriture de la somme dans un troisième. Enfin, Triad est un mélange de Scale et Add.

	nœud	
nœud	0	1
0	44.1	20.3
1	20.3	43.7

(a) bande passante en GB/s selon la localité des accès NUMA

lectures uniquement	87.4
3 :1 lectures écritures	78.7
2 :1 lectures écritures	77.7
1 :1 lectures écritures	77.9
équivalent de STREAM triad	71.9

(b) bande passante en GB/s en fonction du ratio lecture/écriture. Chaque thread accède au nœud UMA local. Les résultats représentent la somme des bandes passantes sur chacun des deux nœuds UMA

TABLE 1.2.2 – Bande passante en GB/s selon la localité des accès et le ratio lecture/écriture résultats obtenus avec Intel Memory Latency Checker sur un processeur Sandy Bridge

Les effets NUMA qui influence la latence d'un accès à la mémoire se retrouve dans la mesure de la bande passante disponible. La bande passante atteignable est plus importante lorsque les accès sont locaux, figure 1.2.2a. De plus, le ratio d'accès en lecture et en écriture à une influence sur la bande passante mesurée, figure 1.2.2b. Effectuer uniquement des accès en lecture permet de transférer plus de données par unité de temps, c'est-à-dire que cela permet d'atteindre une bande passante plus haute. Lorsque des accès en écriture et en lecture sont effectués en même temps, plus le ratio avantage les accès en lecture meilleure est la bande passante atteinte. Un de nos outils repose sur l'utilisation d'une architecture NUMA, et nous reviendrons donc sur ces effets dans le chapitre 4.

```
1  for( Index_t k=0 ; k<numElem ; ++k )
2  {
3      [...]
4      for( Index_t lnode=0 ; lnode<8 ; ++lnode ) {
5          Index_t gnode = elemToNode[lnode];
6          domain.fx(gnode) += fx_local[lnode];
7          domain.fy(gnode) += fy_local[lnode];
8          domain.fz(gnode) += fz_local[lnode];
9      }
10 }
```

Listing 1.1 – Code extrait de LULESH

STREAM ou PointerChasing sont des benchmarks, c'est-à-dire des micros codes qui ne calculent rien d'utile pour un scientifique, mais il est possible de retrouver des régions de code du "type" de STREAM ou de PointerChasing dans une application scientifique. Dans cet exemple de LULESH (listing 1.1), une mini-application représentative des codes d'hydrodynamique moléculaire (détaillée dans la partie 5.1.2 du chapitre 5), les calculs effectués sur les lignes 6,7 et 8 sont équivalents au noyau STREAM add.

Amélioration des codes sensibles à la bande passante mémoire

Dans [Agarwal et al., 2015b, Agarwal et al., 2015a], les auteurs tracent la courbe cumulative du nombre d'accès par page, trié par objet de données de l'application. Ces graphiques montrent tout de suite si les accès mémoires sont plutôt concentrés sur quelques objets, et au sein même de ces objets de plusieurs méga octets, sur quelles pages de mémoire virtuelle. Les auteurs en déduisent alors que, si la courbe possède une pente variant beaucoup, il est possible que les performances de l'application soient accélérées en plaçant les "bonnes" pages sur une mémoire à forte bande passante. Cependant est-il possible de quantifier ce gain de performance ? Et, peut-on établir une limite, entre une application suffisamment limitée par la bande passante mémoire pour être bénéficiaire d'une nouvelle mémoire, et une application dont la limitation n'est pas suffisamment forte ?

$$BandePassante = tailleDonneeTransferee \times \frac{MemoryLevelParallelism(MLP)}{latence} \quad (1.1)$$

La bande passante mémoire utilisée par une application dépend en partie du degré de parallélisme des accès en mémoire qu'elle effectue [Eklov et al., 2013]. La bande passante mémoire maximale disponible est caractérisée par la quantité de requêtes que peut traiter le système mémoire en une seconde, lorsque tous les cœurs de la machine effectuent suffisamment de requêtes en parallèle pour saturer le système. Celle-ci dépend du placement des données : parallélisme des voies mémoires et des bancs mémoires [Yun et al., 2014], mais aussi de la taille des files d'attente, et de la répartition lecture/écriture.

L'équation 1.1 est tiré des travaux [Eklov et al., 2013]. La *latence* représente le temps moyen d'accès à une donnée, et la *tailleDonneeTransferee* représente la quantité de donnée transférée. Cette dernière est obtenue en multipliant la taille d'une ligne de cache (généralement 8 octets) par le nombre de ligne de cache transférées.

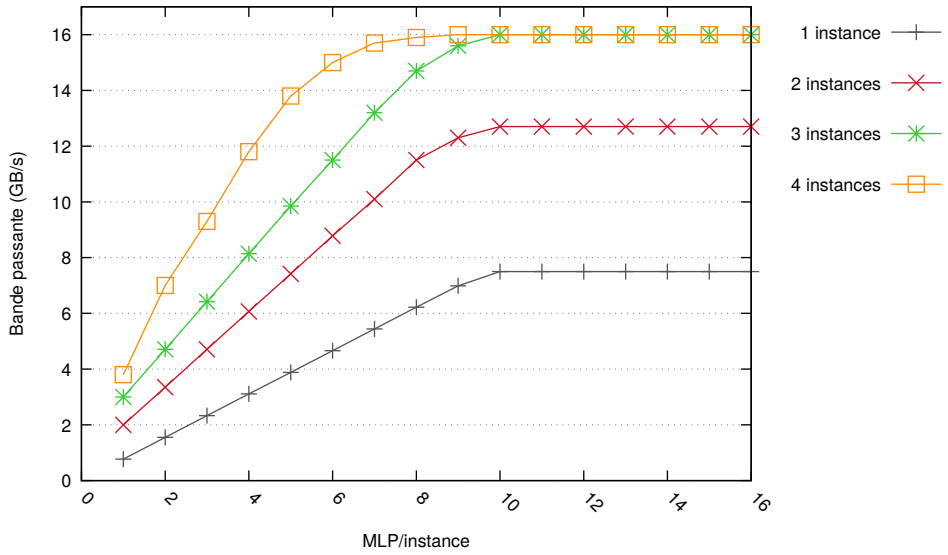


FIGURE 1.2.2 – Bande passante en fonction du niveau de parallélisme mémoire (MLP) sur un Intel Xeon E5520 4 cœurs (Nehalem) [Eklov et al., 2013]

Selon cette équation, la bande passante est directement proportionnelle au nombre de requêtes mémoires parallèles, mais cela reste vrai que jusqu'à un certain point. En effet, le système hiérarchique mémoire ne peut pas toujours traiter autant de requêtes en parallèle que ce que peut générer l'exécution du code de l'application. Cette limitation peut générer une contention, qui se caractérise par un plafond de performance qu'on peut observer sur la figure 1.2.2. Sur cette figure, le degré de parallélisme MLP correspond au nombre de tableaux parcourus par chaque processus léger, à chaque itération, le nombre de processus léger correspond quant à lui au nombre d'instance. Les différentes courbes représentent différentes instances s'exécutant simultanément, il y en a au plus 4, car le Nehalem possède quatre cœurs. Chaque courbe trace la bande passante utilisée par l'application en fonction du degré de parallélisme des accès mémoires instance.

Le MLP, et par définition la bande passante utilisée, dépendent du parallélisme atteint par les accès à la mémoire. Ce parallélisme possède des limites locales et globales, d'où les plafonds de bande passante mesurés. Les limites locales sont principalement dues au nombre de requêtes mémoires que chaque unité de calcul peut soumettre "en même temps". Les limites globales sont dues au nombre de requêtes mémoires pouvant être présentes "en même temps" dans la hiérarchie de mémoires partagées.

Bien que l'équation 1.1 permettent de rendre compte de l'accroissement linéaire de la bande passante en fonction de MLP, l'hypothèse d'une latence moyenne constante quelle que soit la bande passante mesurée ne correspond pas à la latence réellement subie par chaque accès mémoire. Sur la figure 1.2.3, représentant la latence d'un accès mémoire en fonction de la bande passante utilisée, on peut observer qu'à partir d'un certain nombres d'accès par seconde, la latence d'un accès croît exponentiellement [Jacob, 2009].

La figure 1.2.4, page 16, est tirée de l'article [Radulovic et al., 2015] présentant les

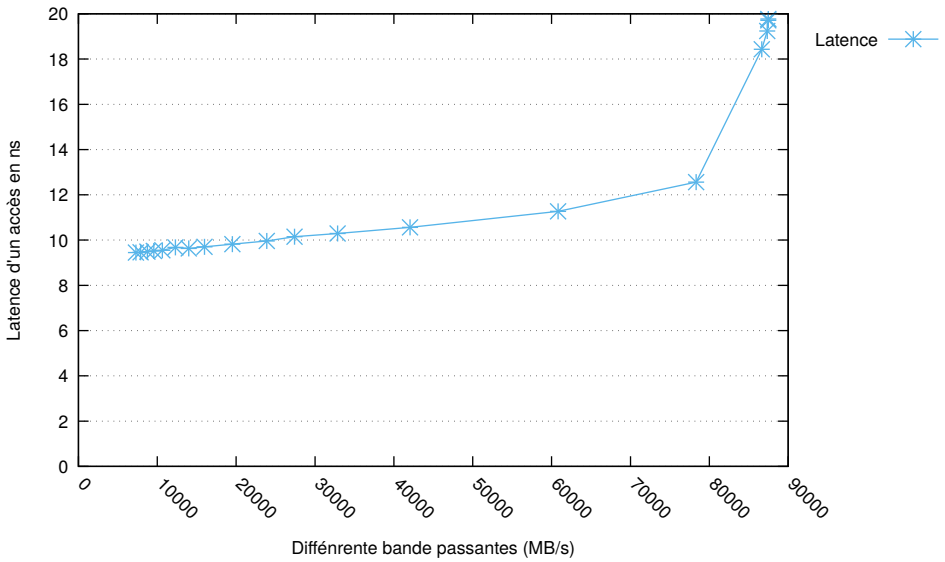


FIGURE 1.2.3 – Latence moyenne d'un accès mémoire en fonction de la bande passante utilisée sur Sandy Bridge (résultats recueillis avec Intel Memory Latency Checker)

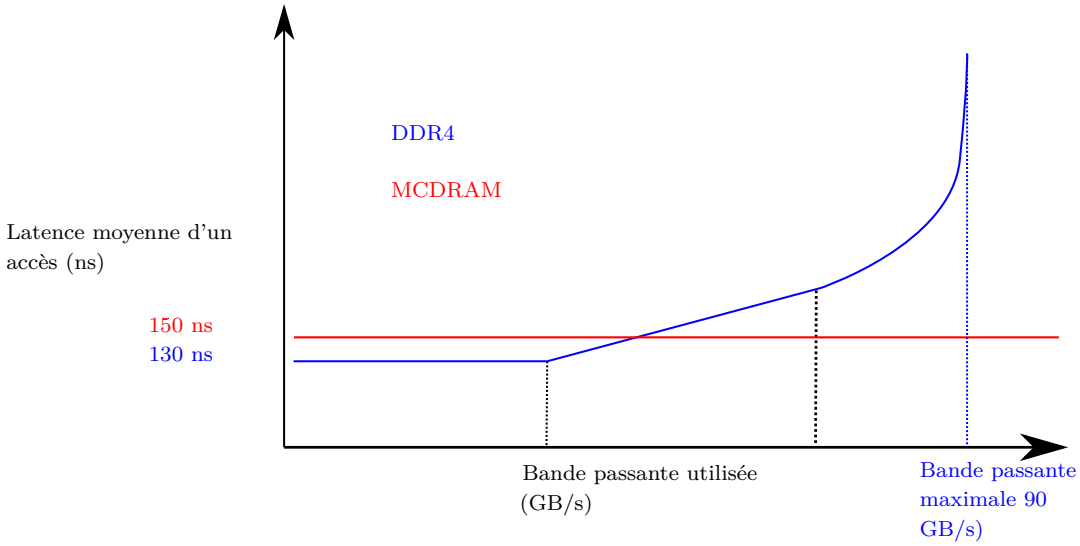


FIGURE 1.2.4 – Latence d'un accès mémoire en fonction de la bande passante mémoire utilisée par l'application, comparaison de deux mémoires aux caractéristiques de bande passante différentes

avantages pour le HPC d'une mémoire à forte bande passante. Les auteurs montrent qu'avec un système à deux mémoires dont une possède 5 fois plus de bande passante que

l'autre, la région au-delà des 80% d'utilisation de la bande passante maximale théorique dans laquelle la latence de chaque accès croît dramatiquement est repoussée d'un facteur 5. Ainsi il est possible d'effectuer jusqu'à cinq fois plus d'accès mémoire en parallèle, avant de voir la latence des accès croître exponentiellement. Il est donc bénéfique d'augmenter la bande passante mémoire physique du système sur lequel s'exécute une application utilisant suffisamment de bande passante mémoire, c'est-à-dire lorsque l'usage de bande passante mémoire de l'application dépasse l'abscisse correspondant à l'intersection des courbes rouge et bleue sur la figure 1.2.4. Ainsi, la bande passante utilisée ne sature plus la bande passante disponible et la latence de chaque accès mémoire diminue.

1.3 Émergence de systèmes à mémoires hétérogènes

Après un rapide tour d'horizon des technologies mémoires employées dans la fabrication des nœuds de calcul, nous verrons comment celles-ci peuvent être assemblées pour que leurs caractéristiques se complètent au sein d'une Architecture à Mémoires Hétérogènes (AMH). Ce type d'architecture est un système composé de différentes mémoires principales (supérieures à la centaine de MB) possédant des caractéristiques de latence, bande passante et capacité mémoire différentes. Nous verrons aussi en quoi il est difficile de bénéficier du potentiel de calcul d'une AMH et quelle est donc la problématique sous-jacente.

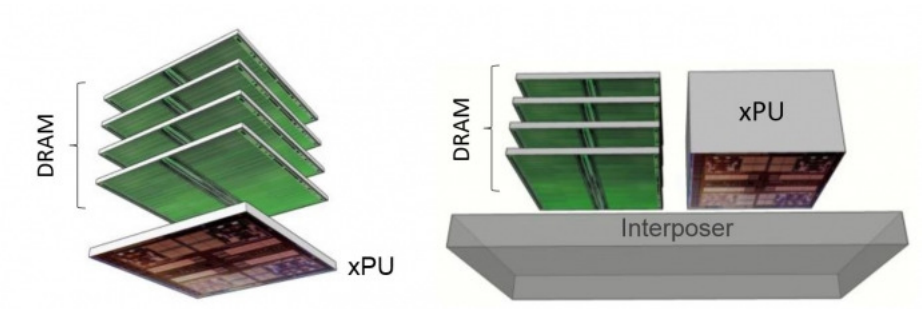
1.3.1 Quelques technologies mémoires

DRAM, SRAM et NVRAM

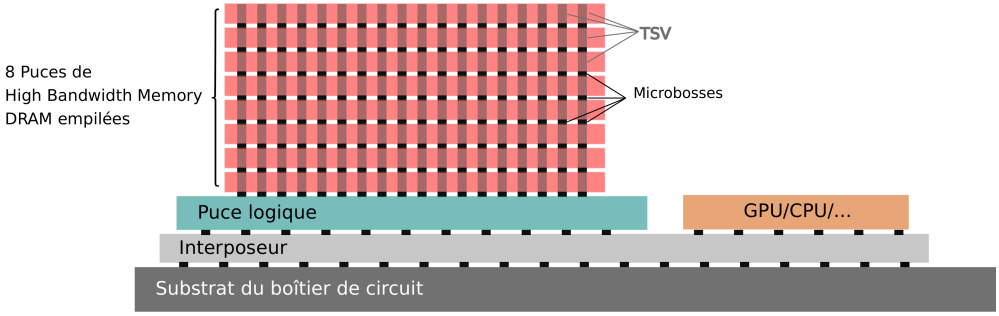
Historiquement, la Random Access Memory (RAM) est fabriquée comme un tableau dont on peut accéder à chaque bit individuellement. Une mémoire RAM peut être accédée aussi bien en lecture qu'en écriture et elle est volatile, c'est-à-dire qu'elle perd ses données avec le temps qui passe ou lorsque le courant est mis à zéro.

On peut différencier les mémoires statiques des mémoires dynamiques. Une mémoire statique (Static Random Access Memory (SRAM)) garde les données tant qu'un courant y est appliqué et la grande majorité des mémoires caches utilisent cette technique. En général, il faut 4 à 6 transistors pour fabriquer une cellule de SRAM contre 1 transistor et 1 condensateur pour une cellule de Dynamic Random Access Memory (DRAM) [Pav, 2008], cette dernière est donc plus dense et moins chère. C'est en partie pour cela que la mémoire hors puce, représentant la majeure partie de la capacité mémoire totale, est généralement composée de DRAM. Au contraire, la mémoire cache est généralement composée de SRAM parce qu'une cellule SRAM est plus rapide qu'une cellule de DRAM, étant donné que la DRAM perd ses données à moins d'être rafraichie périodiquement, ce qui a un coût sur les performances.

De nombreuses technologies de RAM non-volatile sont étudiées : RAM Ferroélectrique (FeRAM), la mémoire à changement de phase (PCM), la RAM magnétique (MRAM), les mémoires à transfert de couple de rotation (STTM) [Chen, 2016]. Ces technologies mémoires révolutionnent les temps de transferts de données entre la mémoire principale et le disque dur externe [Mittal and Vetter, 2016a]. Elles ne sont pas encore dans les outils



(a) Mémoire empilée : 3D sur CPU ou 2.5D sur un interposeur juxtaposé au CPU [(AMD),]



(b) Schéma d'une mémoire empilée en 2.5D

FIGURE 1.3.1 – Schémas d'une mémoire empilée

de production à grande échelle pour remplacer les technologies SRAM et DRAM, mais cela pourrait bien se passer au cours des années qui suivront cette thèse.

Mémoires empilées

La technologie mémoire de mémoire empilée (Stacked-DRAM) (cf figure 1.3.1) permet un bond en avant dans les performances mémoires de bande passante. Le Through Silicon Via (TSV) interconnect est un des composants essentiels des technologies de mémoire empilée. Il rend possible l'empilement 3D de DRAM via une interconnexion jusqu'à l'interposeur, lui-même faisant la connexion avec le CPU et aboutissant ainsi à une densité plus élevée du dispositif. Cette densité élevée permet d'augmenter la bande passante théorique délivrée par le système de mémoires physiques.

Il y a deux interfaces utilisées actuellement pour la construction de ces mémoires : l'interface HBM développée par AMD et Hynix et l'interface HMC développée par Micron technology. Leurs différences sont résumées dans le tableau en figure 1.3.1, d'après Open Silicon [Silicon, a, Silicon, b]. Une pile HBM2 (deuxième génération) peut contenir jusqu'à huit puces DRAM. Chaque pile possède deux canaux de 128 bits par puce pour un total de 16 canaux et une largeur de 2048 bits au total. Un Graphical Processing Unit (GPU) avec 4 piles possède donc un bus mémoire de 8192 bits de large. En comparaison, la largeur de bus d'une mémoire Graphic Double Data Rate Synchronous Dynamic Random Access Memory (DDR SDRAM) est 32 bits, avec 6 canaux, soit une interface de 512 bits. C'est-à-dire huit fois moins que l'interface HBM2. Ainsi la bande passante d'une mémoire

empilée est bien supérieure à celle d'une technologie DDR SDRAM ou même GDDR. Ces mémoires occupent déjà les GPU Nvidia, et elles sont en compétition pour l'ère exascale avec les autres mémoires.

	HBM2	HMC Gen3
Densité	8 GB (4GB)	8GB (4GB)
Bande passante	256GB/s	480GB/s (320GB/s)
Entrée/Sortie	Parallèle, 8 canaux, 182b par canal	SerDes
Type de paquet	Si-interposer	Discret (SerDes)
Capacité d'expansion	Non	Oui, par chaînage
Accès mémoire	DDR SDRAM	Basé sur paquet
Fournisseur	SK Hynix et Samsung	Micron
Dissipation thermique	Haute (Logique + DRAM dans un seul parquet 2.5D ASIC)	Faible (Circuits discrets))
Marché ciblé	Carte Graphiques, Réseaux, Mémoire accédée moins fréquemment, mini PC	HPC, Réseaux

Table 1.3.1 – Résumé des caractéristiques HMC et HBM

1.3.2 Systèmes composés de mémoires homogènes

Comme nous l'avons déjà évoqué dans la partie 1.1, l'assemblage de plusieurs CPU, ou autres accélérateurs, reliés par un réseau implique un temps de latence différents pour le traitement d'une requête selon que la donnée est locale ou distante, cet assemblage constitue une machine NUMA. Dans certains systèmes [Ziabari et al., 2016], le matériel abstrait cette complexité en permettant à un processus s'exécutant sur un processus d'accéder de manière transparente à la mémoire distante.

Sur de telles architectures il est nécessaire de tenir compte de la localité matérielle des données afin de réduire le temps d'accès et donc le temps d'exécution de l'application. Hwloc [Broquedis et al., 2010, Goglin, 2014, Goglin, 2016] est une bibliothèque permettant de tenir compte de la topologie de l'architecture dans le placement des données en dévoilant au logiciel cette topologie (cf capture d'écran de topologie rendue avec Hwloc en annexe). Celle-ci est représentée sous forme d'arbre : le nœud de calcul est vu comme le nœud zéro de l'arbre (Machine), les nœuds composant la machine à architecture NUMA sont vus comme des fils et ainsi de suite avec les mémoires, les cœurs et les unités de calcul au sein des cœurs.

Avec cette connaissance topologique, il est possible de définir des modèles de placement de données tenant compte du fait que le coût d'un accès mémoire dépend de la localité matérielle de la donnée rapatriée[Leon and Hautreux, 2018]. Le problème de réduction

du temps d'accès aux données sur une machine NUMA revient à un problème d'optimisation de la localité de donnée vis-à-vis des processus léger/processus accédant à cette donnée [Pérache et al., 2008, Zhang et al., 2015].

1.3.3 Systèmes composés de mémoires hétérogènes

Nous avons vu qu'il existe trois caractéristiques mémoires qui peuvent être limitantes lors de l'exécution d'une application scientifique : la latence, la bande passante et la capacité (volatile ou non). Il est possible d'optimiser une technologie mémoire pour qu'elle soit très bonne selon l'un de ces axes : par exemple une mémoire cache possède une très faible capacité mais une latence faible et une bande passante élevée, une mémoire empilée possède une très bonne bande passante mais une faible capacité et une latence moyenne, une DDR SDRAM possède une plus grande capacité mais une bande passante plus faible, une Non Volatile Memory (NVM) possède une capacité élevée et non volatile mais une bande passante moins élevée et une latence meilleure (selon la technologie). La technologie de mémoire empilée permet de réduire drastiquement le fossé entre les performances de débit mémoire et les performances de débit des unités arithmétiques et logiques vectorisées. Cependant, le niveau de compétence actuel ne permet pas d'intégrer ces mémoires en quantité suffisantes pour subvenir aux besoins de capacité mémoire des applications HPC.

Dans cet article [Vijayaraghavan et al., 2017], des chercheurs chez AMD en collaboration avec le Département à l'Énergie des États-Unis explicitent leur vision d'un nœud de calcul respectant les contraintes de performance pour assembler un supercalculateur exaflopique. L'architecture, et notamment le système mémoire correspondant à la définition d'une AMH, y sont détaillés. Les auteurs vont jusqu'à envisager une architecture avec 3 niveaux de mémoires hétérogènes : mémoire empilée, DDR SDRAM et NVM, chaque mémoire possédant des caractéristiques répondant à un besoin et les trois mémoires se complétant. Dans leur modèle, la mémoire empilée est placée au contact des accélérateurs (GPGPU), et les deux autres mémoires sont disposés autour, formant ainsi une machine NUMA. L'unification de la mémoire sur l'ensemble prédispose le programmeur à utiliser indifféremment chacune des mémoires, aussi bien pour le calcul sur accélérateur, que pour celui sur CPU. Finalement, les auteurs concluent quant à l'importance de placer les données fréquemment accédées sur la mémoire rapide du système pour les performances, et sur la mémoire à faible consommation dynamique d'énergie pour respecter la contrainte énergétique. Sur un système composé d'une NVM et d'une DDR SDRAM, ces deux objectifs correspondent à maximiser la fréquence des accès sur la DDR SDRAM par rapport à la NVM. La consommation statique de la NVM étant plus faible, il est intéressant de conserver les données peu accédées sur cette mémoire.

Au moment de ma thèse, il existe des Accelerated Processing Unit (APU) rentrant dans la définition d'une AMH, assemblées par AMD selon leur "standard" Heterogeneous System Architecture (HSA), mais elles sont peu présentes dans les supercalculateurs ; elles le sont surtout dans les consoles de jeux vidéos.

Le processeur KNL [Sodani, 2015] (Knight's Landing) est un autre exemple d'AMH dont le système mémoire est composé de deux niveaux de cache dont un partagé, d'une mémoire DDR4, mémoire classique DDR SDRAM de quatrième génération, et d'une

Mode d'opération	Flat-quadrant	Snc4-local	Scn4-distant
Latence maximum (ns)	156.1	153.6	164.7
Latence moyenne (ns)	154.0	150.5	156.8
Latence minimum (ns)	152.3	148.3	150.3
Écart type (ns)	1.0	0.9	3.1

(a) Latence de la Multi-Channel Dynamic Random Access Memory (MCDRAM)

Mode d'opération	Flat-quadrant	Snc4-local	Scn4-distant
Latence maximum (ns)	133.3	130.0	141.5
Latence moyenne (ns)	130.4	128.2	133.1
Latence minimum (ns)	128.2	125.4	126.5
Écart type (ns)	1.2	1.1	3.1

(b) Latence de la DDR4

Table 1.3.2 – Latence des différentes mémoires du KNL [McCalpin, 2016]

mémoire empilée (technologie HMC) appelée MCDRAM. Il est possible d'utiliser la MCDRAM du KNL en mode explicite pour la pile logicielle (flat) ou bien en mode cache (cache matériel) ou encore en mode hybrid (la moitié en cache matériel et l'autre moitié en flat).

La figure 1.3.2 est composée de deux tableaux contenant les chiffres des latences et bande passante mémoires mesurées sur KNL [McCalpin, 2016], et montrant que la DDR4 possède une meilleure latence. Le premier tableau, figure 1.3.2a, contient les latences d'accès aux deux mémoires principales du KNL : la MCDRAM et la DDR4. Les différents modes d'opérations correspondent aux différentes manières de configurer les interactions entre les nœuds de l'architecture NUMA du KNL. Le second tableau, figure 1.3.2b, contient les temps de latence d'accès à la DDR4. On note que la latence d'un accès à la MCDRAM est plus grande que la latence d'un accès à la DDR4, et cela quel que soit le mode choisi. Ces différents modes (Flat-quadrant, Snc4) correspondent à la manière dont les accès mémoires sont gérés par le matériel :

— Flat-quadrant : il n'y a pas de vision NUMA du point de vue logiciel. La localité n'est pas exploitée.
— Snc4 : la machine est vue comme un ensemble de 4 nœuds UMA par la pile logicielle, et il est donc possible d'exploiter la localité des accès mémoires.

Les tableaux de la figure 1.3.3 montrent les bande passantes mémoires de la MCDRAM et de la DDR4 du KNL, mettant en évidence les performances 5 fois meilleures de la première sur la seconde. La bande passante a été mesurée avec le benchmark STREAM [McCalpin, 1995], la mémoire MCDRAM a été testée dans le mode cache et le mode explicite (flat). On observe une meilleure bande passante mesurée lorsque la mémoire MCDRAM est explicitement gérée par les couches logicielles et que toutes les données sont placées dessus à l'aide de la commande *numactl*. Notons que, bien que l'usage mémoire du benchmark est très en deçà de la capacité mémoire de la MCDRAM, le mode cache ne permet pas de bénéficier de toute la bande passante mémoire disponible. Cela est en

Mode d'opération	Flat-quadrant	Cache-quadrant
Copy	389.5	244.5
Scale	363.8	242.9
Add	435.5	299.5
Triad	435.4	299.5

(a) Bande passante de la MCDRAM (GB/s)

Mode d'opération	Flat-quadrant
Copy	74.5
Scale	74.7
Add	83.0
Triad	83.0

(b) Bande passante de la DDR4 (GB/s)

TABLE 1.3.3 – Bandes passantes des différentes mémoires du KNL mesurées avec STREAM [McCalpin, 1995]. Tailles cumulées des tableaux : 3.6GB

partie dû au fait que les données sont d'abord déplacées de la DDR4 vers la MCDRAM par l'algorithme matériel avant d'être copiées dans les deux niveaux de caches puis les registres. Sur des patterns d'accès de ce type il est possible qu'une MCDRAM gérée par le logiciel permette de mieux placer les données qu'en utilisant le mode cache, puisque les données sont déjà présentes sur la mémoire à forte bande passante et on évite ainsi des mouvements de données parasites.

Finalement, lorsque la MCDRAM est en mode cache, son utilisation est totalement transparente du point de vue de l'utilisateur final, des supports exécutifs et du système d'exploitation. Néanmoins, les 16 GB de capacité mémoire sont donc "perdus" puisque les données contenues seront redondantes des données sur la DDR4. De plus, comme nous l'avons vu, les performances en latence de la MCDRAM sont moins bonnes que celle de la DDR4, et ce n'est pas le comportement attendu d'une mémoire cache. C'est pourquoi, les algorithmes appliqués par le matériel pour sélectionner dynamiquement les données à mettre sur la MCDRAM en mode cache ne sont pas toujours adaptés au pattern d'accès à la mémoire des applications scientifiques.

1.4 Le placement/allocation de données sur une architecture complexe

Comme nous l'avons vu dans la partie 1.3.2, sur une architecture NUMA, la stratégie de placement de données sur les différentes mémoires, pour minimiser le temps d'exécution de l'application, dépend des localités relatives du processus léger accédant et de la donnée accédée [Pérache et al., 2008, Zhang et al., 2015]. Au contraire, sur une AMH les performances d'accès à une donnée ne dépendent pas de la localité du processus léger qui y accède mais uniquement des caractéristiques de la mémoire sur laquelle se situe la donnée, et des caractéristiques des accès mémoires du code s'exécutant (sensibilité à

la latence ou à la bande passante). De plus, **tous** les processus légers sont impactés de la même façon par le placement d'une donnée sur une mémoire plutôt qu'une autre sur une AMH, contrairement à une machine NUMA. Le problème de placement de données sur une AMH est donc différent de celui du placement de données sur une architecture NUMA [Meswani et al., 2015].

De nombreux travaux ont déjà été faits sur le placement de données sur une AMH dans le domaine de l'embarqué : [Idrissi Aouad and Zendra, 2007, Bai and Shrivastava, 2010, Kim et al., 2011, Guo et al., 2011, Zhuge et al., 2012, Guo et al., 2013, Che and Chatha, 2013, Vaumourin et al., 2016]. Cependant, ces travaux se concentrent sur une architecture composée de mémoire aux caractéristiques très différentes. Dans l'embarqué, on cherche à contrôler très finement le pire temps d'exécution, il est donc courant d'utiliser une » mémoire cache « qui soit gérée par les couches logicielles, appelée scratchpad. Ce n'est pas une mémoire cache, puisqu'elle ne suit pas la logique de placement de donnée d'une mémoire cache matérielle, mais elle en possède les caractéristiques physiques de latence, bande passante et capacité mémoire. Le problème consiste alors à sélectionner les données à placer sur le scratchpad, une mémoire possédant une meilleure latence et une meilleure bande passante en lecture et en écriture que la mémoire hors puce, mais avec une capacité mémoire de 2 à 4 ordres de magnitudes en deçà de celle de l'autre mémoire. Le problème de placement de données sur un système de mémoires hétérogènes étudié en HPC n'est donc pas identique au problème de placement étudié dans ces travaux.

Ces travaux ont néanmoins déjà différencié deux types d'approches [Egger et al., 2006] qu'on retrouvera sur la problématique HPC : les approches statiques cherchant à résoudre le problème avant l'exécution et les approches dynamiques modifiant la solution au cours de l'exécution. Les approches statiques résolvent donc un problème d'**allocation de données**, alors que les approchent dynamiques résolvent un problème de **placement de données**, parce que contrairement à l'allocation s'exerçant jusqu'à la fin de vie de la donnée, le placement peut être modifié alors que la donnée est toujours vivante. Certains travaux [Udayakumaran and Barua, 2003, Udayakumaran et al., 2006] ont montré qu'une approche dynamique dans la résolution du problème donne de meilleurs résultats pour une architecture embarquée contenant une mémoire scratchpad et des applications conçues pour de telles architectures. Cependant, il est moins complexe de s'attaquer à la résolution du problème d'allocation de données, puisque la liberté de mouvement en moins simplifie le problème. Dans ce mémoire, nous nous sommes concentrés sur la résolution du problème d'allocation de données.

Que ce soit pour une approche dynamique ou statique, tous ces travaux commencent généralement par définir le problème comme un problème linéaire en nombre entiers et à le résoudre. On peut résumer la méthode de placement/allocation des données en 5 étapes : formulation d'un problème linéaire en nombre entier, évaluation des coefficients (généralement avec un profilage de l'application), résolution, modification du code (automatique ou à la main) et exécution (avec ou non une bibliothèque pour le placement des données). Même si la formulation du problème d'allocation de données sur une AMH pourrait s'inspirer de leurs travaux, les coefficients d'un problème linéaire en nombres entiers ne seront pas les mêmes et ne seront pas calculés de la même façon.

1.5 Problématique

Les AMH composent le paysage des architectures utilisées dans le HPC et la course à l'exascale ne va que renforcer leur place dans ce domaine. L'émergence de la technologie de mémoire empilée est une opportunité pour réduire le fossé entre les performances des systèmes mémoires et celles des systèmes de calcul afin de repousser le mur mémoire [Wulf and McKee, 1995, Radulovic et al., 2015]. Bien que certaines études aient cherché à remplacer complètement la mémoire principale DDR SDRAM par de la mémoire empilée, ce n'est pas économiquement viable d'intégrer suffisamment de mémoire empilée pour au moins atteindre la capacité mémoire actuelle. De plus, des systèmes hétérogènes composés de mémoires NVM sont envisagés. Certains envisagent aussi des systèmes composés des trois types de mémoires [Vijayaraghavan et al., 2017]. Cela marque donc l'émergence de nouveaux systèmes à mémoires hétérogènes appelé aussi AMH. Un des premiers exemples de ces systèmes composés d'une mémoire empilée et d'une DDR SDRAM est le KNL.

Sur une AMH, comment allouer chaque donnée sur une mémoire de sorte à minimiser le temps d'exécution total de l'application, sans que l'usage mémoire ne dépasse la capacité mémoire d'aucune mémoire du système ? Comme nous l'avons vu sur l'exemple avec le KNL, il semble important de placer les données sensibles à la bande passante sur la MCDRAM.Ce problème de placement peut-être traité par la pile logicielle, le programmeur ou le matériel, ou peut-être une coopération de plusieurs acteurs.

1.6 Contributions

Dans ce manuscrit nous essaierons de répondre à la problématique en :

— formulant et en proposant une heuristique pour résoudre le problème d'allocation de données sur une architecture à mémoires hétérogènes dans le chapitre 3, partie 3.1,
— développant une méthode de profilage du code et de résolution du problème d'allocation de données, dans les parties 3.2 et 3.3 du chapitre 3,
— implémentant cette méthode pour répondre au cas particulier du processeur KNL, dans le chapitre 4,
— évaluant notre méthode sur une série de benchmarks représentatifs des applications scientifiques, dans le chapitre 5.

Chapitre 2

État de l'art

On peut définir deux grandes catégories dans la manière d'approcher le problème de placement de données. Une première catégorie serait constituée des approches réactives, par opposition aux approches proactives. On dit qu'une approche est réactive lorsqu'elle n'utilise pas a priori de connaissance spécifique à l'application qui s'exécute. Par exemple, le système d'éviction de lignes de cache, implémenté dans une hiérarchie de mémoires caches matérielles, est réactif. Il n'utilise pas de connaissance sur l'application qui s'exécute pour choisir la ligne de cache à évincer. De même, le système d'exploitation, s'il était amené à devoir choisir le placement des pages sur les différentes mémoires d'une AMH, ferait ses choix de manière réactive. Une approche peut être réactive par nécessité parce qu'il est difficile de faire parvenir des informations sur l'application au système prenant les décisions.

La deuxième catégorie est constituée d'approches dites proactives. Une approche proactive possède des informations sur le comportement de l'application. Ces informations sont propres à chaque application. Intuitivement, une approche proactive doit pouvoir prendre de meilleures décisions qu'une approche réactive sur le placement des données, puisqu'elle possède des informations spécifiques à l'application à exécuter. En effet, le placement de données peut prendre en compte les spécificités de chaque application, il peut donc être "taillé sur mesure". Cependant, la transparence de la solution réactive la rend intéressante du point de vue du programmeur.

2.1 Approches réactives au placement de donnée sur AMH

Dans cette première partie nous allons nous concentrer sur ce qui a été fait comme approche réactive au problème de placement de données. Les approches peuvent être décomposées en deux sous groupes : celles basées sur une technique matérielle reposant sur l'utilisation d'une des mémoires comme un niveau de cache supplémentaire, et celles basées sur une technique logicielle, reposant sur une extension du système d'exploitation.

2.1.1 Technique matérielle

Le nombre de cœurs sur puce a tendance à augmenter afin de permettre l'augmentation des performances des processeurs d'une génération à l'autre [Gepner and Kowalik, 2006, Borkar, 2007]. Ce qui a pour conséquence directe la diminution de capacité de mémoire cache par cœur, à moins de réussir à augmenter la capacité totale de mémoire cache disponible sur une puce. Cette augmentation est difficilement envisageable, car l'ajout de mémoire cache sur une puce est limitée par la taille de la mémoire cache elle-même (voir partie 1.3.1).

C'est pourquoi de nombreux travaux de recherche existent pour développer des mémoires caches fabriquées à partir de DRAM [Mittal and Vetter, 2016b]. L'émergence de technologies de mémoire empilée [Loh, 2008] pourrait rendre envisageable l'utilisation de DRAM comme cache de dernier niveau, autrement les performances en latence et bande passante de la DDR SDRAM ne permettaient pas de rivaliser avec la technologie SRAM. En théorie, une mémoire empilée possède le tiers voire la moitié de la latence de la DDR SDRAM, et une bande passante 5 à 15 fois supérieure. En pratique, la MCDRAM du KNL possède une latence 10% plus lente que la DDR SDRAM et une bande passante entre 4 et 5 fois plus élevée.

Dans [Chou et al., 2014] les auteurs étudient la possibilité d'utiliser une mémoire empilée comme mémoire principale gérée par le système d'exploitation en même temps d'être gérée par le matériel comme un cache de dernier niveau. Les auteurs souhaitent une mémoire transparente au niveau de la pile logicielle, mais dont la capacité contribue à la capacité totale de la mémoire principale. Ils proposent donc une organisation de la mémoire qui ressemble à une mémoire cache tout en permettant sa visibilité dans l'espace d'adressage mémoire. Leur système permet de gagner 78% de performance, contre 33% pour une utilisation de la mémoire empilée comme mémoire principale uniquement, et 50% pour une utilisation matérielle en mémoire cache uniquement. Ils considèrent cependant que la latence de la mémoire empilée est deux fois plus rapide que celle d'une DDR SDRAM, avec une bande passante au moins 4 fois plus rapide. Nous avons vu qu'en pratique, sur l'exemple du KNL la latence mesurée de la mémoire empilée n'est pas si rapide (page 21, figure 1.3.2).

Ces auteurs ont donc développé une architecture matérielle permettant le placement (et déplacement) de données entre la mémoire sur puce et la mémoire hors puce, de manière transparente pour l'utilisateur. Ils ont aussi montré que le surcoût dû à l'intervention du système d'exploitation dans le placement des données doit être sérieusement pris en compte.

Pour cela, ils ont simulé une architecture contenant une mémoire sur puce comme cache de dernier niveau (mémoire utilisant la technologie de mémoire empilée) et une mémoire hors puce (DDR SDRAM). Ils ont exécuté des codes de la suite RODINIA, [Che et al., 2009], et de la suite SPECCPU2006, [spe,], et quelques applications de bureautique Windows.

Dans le papier [Meswani et al., 2014] les auteurs présentent une étude comparée de plusieurs approches : cache matériel, OS, profilage dynamique et expert programmeur. Pour leurs expériences, ils simulent l'exécution d'un benchmark de code hydrodynamique sur une architecture composée d'une mémoire à forte bande passante mais possédant 8 fois moins de capacité que la mémoire à faible bande passante.

L'approche expert programmeur (PinU) consiste à suivre les directives de l'expertise de programmeurs. Sur LULESH, PinAB ne donne pas les mêmes tableaux à placer sur la mémoire rapide que PinU et les faibles performances de PinU montrent l'intérêt de développer des outils pour aider les programmeurs à gérer le placement de données sur une AMH. L'approche, PinAB, basée sur un profilage dynamique utilise la quantité de LLCmiss par donnée allouée, ensuite divisé par la taille de la donnée, pour guider les allocations.

Le trafic mémoire généré par l'approche matériel est de 720GB/s contre 450 pour l'OS et seulement 105GB/s pour PinAB et 20GB/s pour PinU. Ce trafic est le revers de la médaille d'un ratio d'accès à la mémoire rapide élevé : 95% et 99% pour, respectivement, l'approche matérielle et l'approche OS, contre seulement 45% pour l'approche PinAB, et 12% pour PinU. Ce trafic mémoire est en grande partie responsable de la consommation énergétique de la mémoire rapide et doit être pris en compte dans l'approche envisagée [Meswani et al., 2015].

Les auteurs concluent sur le fait qu'il est impossible en l'état de définir une meilleure stratégie dans l'utilisation de la mémoire empilée comme un cache de dernier niveau, une mémoire explicite au programmeur ou bien une gestion par le système d'exploitation. Bien que l'approche matérielle est attractive, le coût d'implémentation est difficile à évaluer selon eux. De plus, selon la nature de l'application ce n'est pas toujours la méthode matérielle qui semble apporter les meilleures performances. Les auteurs précisent que ces résultats n'invalident pas l'approche du placement de données conduit par le programmeur, mais ils notent que pour obtenir les meilleures performances, les programmeurs devront probablement redécouper leurs données afin de ne placer sur la mémoire rapide que les blocs les plus accédés.

L'augmentation du nombre de cœurs sur les machines *multi-core* et *many-core* tend à diminuer la capacité de mémoire cache par cœur, mais agrandir la capacité des mémoires caches grâce à la technologie DRAM est une manière de freiner cet effet. Cependant, des auteurs [Vijayaraghavan et al., 2017] ont montré qu'une architecture hétérogène répondant aux caractéristiques exascale ne pourrait pas utiliser sa mémoire à faible capacité et forte bande passante comme un cache matériel sans sacrifier près de 20% de sa capacité mémoire. De plus, la fabrication et l'utilisation de mémoire cache DRAM entraîne de nombreux challenges : performance, architecture et meta-données, fabrication et fiabilité [Mittal and Vetter, 2016b]. L'utilisation de DRAM comme cache matériel est donc un sujet de recherche en plein essor, mais il est possible que les performances des applications

scientifiques bénéficient plus d'une gestion logicielle de ces différentes mémoires.

2.1.2 Technique logicielle

Certains travaux, [Meswani et al., 2014, Meswani et al., 2015], proposent une approche logicielle au problème de placement de données. Ces approches sont basées sur des modifications du système d'exploitation. Dans leurs approches c'est le système d'exploitation qui se charge de placer les pages sur les différentes mémoires en fonction de la stratégie suivie. Pour permettre aux couches supérieures de la pile logicielle d'utiliser les mémoires de ces nouvelles architectures, des outils reposant sur les couches basses de la pile logicielle sont proposées. Dans la littérature, une extension du noyau Linux permettant de transférer efficacement les données d'une mémoire principale à l'autre a été étudiée [Lin and Liu, 2016]. Une bibliothèque, Memkind [Corp., b], a été développée pour permettre d'allouer, via un allocateur nommé *hbw_malloc*, des données sur une deuxième mémoire principale. L'allocateur memkind repose sur l'allocateur jemalloc [jem,]. Par-dessus cette bibliothèque a été développée une autre bibliothèque qui permet de placer de manière transparente à l'utilisateur les données sur une deuxième mémoire, c'est AutoHBW [Corp., a]. La stratégie de placement d'AutoHBW est la même que celle employée avec le logiciel numactl : premier arrivé premier servi. Cependant, avec AutoHBW il est possible de définir une taille minimale pour les données qu'on souhaite placer sur la mémoire empilée.

Dans [Meswani et al., 2015], les auteurs partent du constat suivant : les mémoires de type mémoire empilée feront très certainement partie des architectures exascale, mais elles ne permettront pas de fournir toute la capacité mémoire nécessaire aux unités de calcul. D'où la nécessité d'une AMH composée d'une mémoire sur puce et d'une mémoire hors puce (DDR SDRAM ou NVM). Pour cela, ils simulent une architecture composée d'une mémoire sur puce (mémoire empilée) et d'une mémoire hors puce (DDR SDRAM), sur laquelle ils comparent une approche matérielle, et plusieurs stratégies d'une approche OS. Cette dernière approche n'est pas implémentée en tant que telle dans un OS, mais consiste à considérer des pages 4Ko et un déplacement des données au niveau du gestionnaire des fautes de pages. Les traces contiennent les adresses physiques des accès en lecture/écriture, hiérarchisées dans le temps, car la simulation des interruptions régulières du système d'exploitation sur l'exécution le nécessite. Leur approche matérielle consiste à faire de la mémoire empilée une mémoire cache de dernier niveau.

L'approche OS, basée sur la même stratégie que celle d'un cache, consiste à considérer cette mémoire comme la mémoire principale et la DDR SDRAM devient une mémoire de swap. Trois autres stratégies consistent à découper le temps de l'application en époque (par exemple 100 millions de cycles) pour qu'à chaque nouvelle époque les données accédées "le plus fréquemment" soient transférées sur la mémoire empilée si elles n'y sont pas encore et qu'il reste de la place : FirstTouch, HotPage, HotPage-FirstTouch.

En l'état, les systèmes d'exploitations utilisés sur supercalculateurs ne possèdent pas la bonne infrastructure pour sélectionner les pages à mettre sur une mémoire à forte bande passante. En effet, une entrée dans la table de page ne contient qu'une poignée de bits (le bit de référence, et *dirty* bit sont fréquents) pour se renseigner sur l'état des pages. Cela ne suffit pas à savoir si une page est fréquemment accédée en mémoire et si elle serait donc une

bonne candidate pour la mémoire à forte bande passante. Les auteurs ont donc implémenté un compteur d'accès matériel, dans leur simulation, qui met à jour l'équivalent dans l'OS, ce dernier utilise les bits encore libres des entrées de la table de pages. Enfin, à intervalle régulier, lorsqu'une page doit être déplacée, leur version du système d'exploitation recopie toute une page d'une mémoire sur l'autre, et cela seulement après avoir déclenché une interruption sur le processeur. Après cette recopie, s'ensuit une mise à jour de l'entrée de la table des pages et une invalidation de cette entrée dans le Translation Lookaside Buffer (TLB).

2.1.3 Discussion

Une approche réactive, complètement transparente pour la pile logicielle, n'est pas nécessairement la meilleure approche. Même si une telle approche pourrait obtenir un très bon ratio d'accès validé sur la mémoire rapide, elle génèrerait beaucoup plus de trafic entre les deux mémoires principales [Meswani et al., 2014] qu'une approche proactive. Ce trafic est responsable en partie de la consommation énergétique de la machine. Il faut néanmoins nuancer cela par le fait qu'en pratique, il ne semble pas y avoir une grande différence de consommation énergétique sur une AMH comme le KNL entre une approche proactive logicielle et une approche réactive matérielle [Wu et al., 2017a].

Un autre argument contre les approches réactives réside dans l'observation des caractéristiques mémoires. Les latences des différentes mémoires d'une AMH peuvent être équivalentes, pis encore, il est possible que la mémoire avec la bande passante la plus élevée ait cependant la latence la moins performante (exemple MCDRAM, par rapport à la DDR SDRAM sur le KNL).

Comme nous l'avons vu dans le chapitre 1, on peut imaginer à peu près toutes les combinaisons possibles de systèmes à deux, voire trois mémoires principales. Celles-ci seraient composées de technologie diverse : mémoire empilée, NVM, DDR SDRAM. Ainsi l'hétérogénéité des caractéristiques de latence, bande passante et capacité mémoire de ces AMH n'est pas équivalente au fossé existant entre les latences, bandes passantes et capacités des différentes mémoires d'un système hiérarchique de mémoires caches. Ainsi, les méthodes éprouvées dans le placement de données sur un système hiérarchique de mémoires caches, cache matériel et/ou scratchpad (cf partie 1.4), ne peuvent pas se transposer trivialement à la problématique dans le domaine du HPC.

C'est pourquoi, il existe de nombreux travaux de recherches sur des approches proactives. Parmi les approches proactives, il y a d'une part celles reposant sur une analyse statique, à la compilation, d'autres part celles reposant sur une analyse dynamique, mettant en œuvre profilage en ligne ou hors ligne. Généralement, les méthodes basées sur un profilage de l'application impliquent un effort de la part du programmeur. Malgré cela, de telles méthodes sont courantes dans le domaine du HPC. Les utilisateurs finaux sont prêts à sacrifier du temps de développement pour l'optimisation de leur code si cela peut réduire significativement le temps d'exécution de leur application.

2.2 Approches proactives au placement de données sur AMH

La deuxième catégorie d'approches sont les approches dites proactives. Une approche proactive peut utiliser des connaissances obtenues sur le comportement de l'application soit par une analyse dynamique (profilage), statique (compilation) ou bien algorithmique (programmeur).

2.2.1 Modifications algorithmiques

Dans [Bender et al., 2015] les auteurs développent une étude préliminaire de co-conception, conception de machines à partir des applications, pour l'exécution d'un algorithme de tri sur une AMH. Le but de ces travaux est de montrer que certaines applications scientifiques bénéficieraient d'un système de mémoires hétérogènes composé d'une mémoire à forte bande passante, moyennant le renouvellement des algorithmes employés. Ils prédisent de manière théorique et pratique (simulation) quel est le nombre de cœurs minimal qu'un processeur doit avoir pour être sensible à la bande passante sur un algorithme de tri parallèle. Pour estimer la sensibilité à la bande passante de l'algorithme les auteurs prédisent les temps de calcul et de transfert mémoire attendus. Le premier est évalué à partir de la quantité de calcul et de la fréquence du processeur, le second à partir de la quantité de donnée à transférer et de la bande passante disponible. Finalement, l'algorithme est considéré comme sensible à la bande passante si le temps attendu des transferts mémoires est plus important que le temps de calcul attendu. Ils adaptent ensuite l'algorithme pour déplacer les bonnes données au bon moment sur la mémoire à bande passante élevée afin de minimiser le temps d'exécution de l'application. Ils concluent qu'une machine avec au moins 256 cœurs et une telle mémoire, 8 fois plus rapide que la seconde possédant une plus grande capacité, fournirait une accélération de 25% sur l'algorithme de tri fusion multi-voies GNU [Singler et al., 2007] tels qu'ils l'ont amélioré.

Leur étude se base sur le développement du KNL, qui n'était alors pas encore disponible publiquement. Ainsi la mémoire empilée qu'ils considèrent possède une meilleure bande passante que la DDR SDRAM avec par contre une latence équivalente. Ces deux mémoires se situent sur le même niveau dans la hiérarchie mémoire. Pour développer leur étude théorique, les auteurs augmentent le modèle de mémoire externe séquentiel [Aggarwal and Vitter, 1988] et parallèle (PEM) [Arge et al., 2008]. Les autres modèles existant ne sont pas adaptés à cette configuration de latence équivalentes et ne permettent pas de modéliser une telle architecture. Les auteurs représentent la différence de bande passante en considérant que les blocs de données transférés depuis la mémoire empilée sont plus grands que ceux provenant de la DDR SDRAM.

Les technologies de mémoire empilée et DDR SDRAM qui vont cohabiter sur les futures architectures n'auront certainement pas les caractéristiques des mémoires caches : plus une mémoire est petite, meilleure sont sa latence et sa bande passante. Sur ces architectures, les auteurs montrent que la prise en compte des caractéristiques mémoires de bande passante dans l'écriture de l'algorithme de certaines applications peut permettre d'accélérer leur temps d'exécution. Une telle architecture est donc viable.

2.2.2 Analyse statique

Sans entrer dans la réécriture des algorithmes existants, il est possible de chercher à optimiser le placement de données sur une AMH gérée par la pile logicielle en modifiant le code d'une application scientifique afin de placer les données sensibles à la bande passante (respectivement à la latence) sur la mémoire à forte bande passante (respectivement à meilleure latence). Une manière transparente du point de vue du programmeur d'opérer ce placement de donnée, c'est d'utiliser une passe de compilation modifiant automatiquement le code exécuté.

Dans [Khaldi and Chapman, 2016] les auteures introduisent une nouvelle analyse dans LLVM, un compilateur open-source, afin de repérer les données sensibles à la bande passante. Elles appliquent leur analyse sur le benchmark de gradient conjugué (CG) de la suite NAS parallèle [Bailey et al., 1991], sur un processeur KNL, une AMH composée d'une mémoire à forte bande passante, la MCDRAM, et d'une DDR4. L'analyse permet de modifier, à la compilation, les allocations dynamiques des données sensibles à la bande passante pour que celles-ci soient allouées, à travers la bibliothèque memkind, sur la MCDRAM. Elles obtiennent une accélération de 2.30x comparée à la version du code non modifiée, et pour laquelle toutes les données sont allouées sur la DDR4 du KNL.

Concrètement, la stratégie de sélection de données implémentée dans l'analyse statique proposée par [Khaldi and Chapman, 2016] repose sur une fonction de priorité. L'analyse de chaines Utilisation-Definition de LLVM est étendue à l'interprocédurale et permet de faire correspondre les références, instructions statiques (accès en lecture et en écriture) du code, aux différentes données allouées dynamiquement. Ensuite, on attribue une valeur à chaque accès effectué selon la formule suivante : $P(R) = bandwidth(R) \times \sum_{r \in R} cost(r) \times workshare(r)$

- R est l'ensemble des références sur le tableau analysé.
- bandwidth(R) est égal à 0 si les accès sont irréguliers (indirection) et 1 s'ils sont réguliers, c'est-à-dire si les accès peuvent être décrit comme une fonction linéaire des itérations de boucle.
- cost(r) est égal au nombre de références, instructions statiques, les accès en écriture comptant double et les accès dans des boucles ne comptent que pour un.
- workshare(r) est égal à 1 si l'accès est effectué dans un pragma OpenMP 0 sinon.

Ni les accès contenant des indirections, ni les accès fait en dehors de région OpenMP parallèle ne comptent. De plus, les accès en écriture sont privilégiés par rapport aux accès en lecture. Cette hypothèse, privilégiant les accès en écriture, est validée par [Laghari and Unat, 2017]. Dans ce papier les auteurs comparent les performances de micro benchmarks n'effectuant soit que des lectures, soit que des écritures sur KNL, en plaçant toutes les données sur MCDRAM, ou sur DDR4. Leurs résultats montrent que la bande passante maximale atteinte est plus élevée en lecture qu'en écriture, quelle que soit la mémoire utilisée. De plus, la bande passante atteinte avec des lectures sur DDR4 est moins élevée que celle atteinte en écriture sur MCDRAM. Selon leurs expériences, il est plus intéressant de placer d'abord les objets accédés en écriture sur la MCDRAM afin de relevé le niveau par le bas, ce qui permet d'améliorer les performances globales.

Les travaux de Chapman *et al.* mettent en évidence l'intérêt d'une approche statique pour sa transparence du point de vue utilisateur. Mais cette approche possède des limites : aliasing entre pointeurs, impossibilité de différencier les accès mémoire des accès dans le cache, restriction au parallélisme OpenMP. De plus, sur le benchmark de Gradient Conjugué utilisé pour évaluer leurs métriques, les auteures mettent en évidence l'importance d'une analyse dynamique supplémentaire pour avoir la taille des tableaux puisque le tableau qui apporte le plus d'accélération par son allocation sur la MCDRAM n'est pas celui qui obtient la plus haute évaluation avec leur fonction, mais celui qui a le plus grand usage mémoire.

2.2.3 Analyse dynamique

Nous avons déjà détaillé certains travaux [Meswani et al., 2015] comparant une approche proactive basée sur une analyse dynamique à une approche matérielle, réactive, dans la partie 2.1.1. Dans cette partie nous allons détailler d'autres travaux qui se sont intéressés à des approches proactives reposant sur une analyse dynamique et donc pouvant faire intervenir le programmeur dans une étape de profilage.

Outils de profilage

Il existe de nombreux outils de profilage temporel. Gprof [Graham et al., 1982] permet de faire un profilage de programme séquentiel en mesurant les temps relatifs passés dans chaque fonction d'un programme. Il fonctionne en échantillonnant l'exécution du programme. Ainsi il n'est pas possible de connaître le temps exact passé dans chaque fonction, mais il est possible de connaitre le pourcentage, par rapport au temps total, passé dans chaque fonction. Cela permet notamment de mettre en évidence les points chauds du programme. Oprofile [Levon and Elie, 2004], est un outil de profilage par échantillonnage pour une application aussi bien que pour un PC de bureau ou un serveur. Un administrateur peut l'utiliser pour profiler l'utilisation faite par tous les processus/applications s'exécutant sur une machine. Il est utilisable aussi en contexte HPC puisqu'il peut être lancé en même temps qu'une application parallèle pour profiler celle-ci. En plus des informations de temps relatif passé dans chaque fonction, Oprofile récupère les valeurs de certains compteurs matériels et peut ainsi donner des informations comme le nombre de LLCmiss par fonction. HPCToolkit [L. et al.,], Tau [Shende and Malony, 2006], et Vtune [Sridharan, 1997], Caliper [Hundt, 2000] sont des boîtes à outils plus ou moins complexes de profilage pour le HPC. Ces outils fournissent essentiellement un profilage orienté vers le code. C'est-à-dire qu'ils vont aider le programmeur à trouver les lignes de son code qui sont responsables des mauvaises performances de son application.

Lorsque les performances de l'application sont limitées par les performances du système mémoire, il est préférable de mettre en évidence les données de l'application responsables de ces mauvaises performances, plutôt que les lignes de codes. En effet, à partir d'une même ligne de code des accès à différentes données peuvent être effectués et qu'un objet peut être accédé depuis des lignes de codes différentes. C'est ce qu'expliquent les auteurs de MemSpy [Martonosi et al., 1992, Martonosi et al., 1995], les premiers à avoir introduit la notion de profilage orienté données. Cet outil permet de mettre en évidence des problèmes

de performance dûs à des accès mémoires dans l'exécution d'une application. Le but est de permettre aux applications scientifiques de bénéficier le plus possible de localité temporelle, et spatiale [Agawal and Gupta, 1988]. Pour cela, les auteurs soutiennent le fait qu'il est nécessaire d'avoir des informations par objet de données, de connaître le temps que le CPU passe à "ne rien faire" (stall) et de différencier la nature des LLCmiss.

La conception d'un outil de profilage orienté données soulève un certain nombre de problèmes : Comment identifier les objets de données ? Comment relier les informations au code source pour aider le programmeur ? L'identification des objets se fait par leur adresse virtuelle et la pile d'appels qui a mené à l'allocation dynamique. Ainsi certains objets de données sont agrégés mais cette agrégation a du sens puisqu'elle correspond à une même allocation du point de vue du programmeur. Une manière pragmatique de nommer les objets de données consiste à concaténer le nom de la variable, son type, ainsi que la pile d'appels (nom de fonction+compteur de programme). Pour relier les informations récupérées à l'objet de donnée correspondant, une simple comparaison entre l'adresse accédée et la plage d'adresse de l'allocation dynamique suffit. L'outil MemSpy fait aussi la différence de nature des accès : premier accès, éviction de ligne de cache ou bien invalidation. Techniquement, MemSpy instrumente les allocations dynamiques de l'application à la compilation afin de récupérer le pointeur, la taille et le nom de la variable. Dynamiquement, il est possible de récupérer la taille et le pointeur en capturant les appels à la libc qui alloue dynamiquement de la mémoire, mais le nom de la variable ne peut être connue qu'avec une instrumentation préalable.

Depuis, d'autres outils de profilage orienté données ont été développés : une extension de la suite d'outils de Sun ONE Studio^{TM} [Itzkowitz et al., 2003] (propriétaire), ADAMANT [Cicotti and Carrington, 2016] (open-source), MemAxes [Giménez et al., 2014] (open-source) et une extension de CallGrind [Pena and Balaji, 2015, Peña and Balaji, 2016, Peña and Balaji, 2014a] (non disponible). Certains de ces outils se basent sur l'utilisation de compteurs matériels [Itzkowitz et al., 2003, Cicotti and Carrington, 2016, Giménez et al., 2014], d'autres sur l'utilisation d'un simulateur [Pena and Balaji, 2015, Peña and Balaji, 2016, Peña and Balaji, 2014a] et Gleipnir [Janjusic and Kavi, 2013]. Ce choix est un compromis entre le surcoût à l'exécution et la précision souhaité de l'analyse.

Avec un outil basé sur Valgrind [Nethercote and Seward, 2007], une trace est générée et celle-ci peut être utilisée sur un simulateur afin de reproduire le comportement d'une machine qui n'est pas nécessairement disponible. Cependant, le surcoût à l'exécution avec un tel outil est de plusieurs ordres de magnitude, ce qui rend impossible son utilisation sur une application "réelle". C'est le cas de l'outil Gleipnir [Janjusic and Kavi, 2013], qui permet de générer une trace, ensuite analysée ou bien utilisée pour simuler une exécution. Ainsi il est possible de connaître avec une fine précision le nombre de référence et de LLCmiss. Les auteurs mettent en évidence sur une application de parcours d'arbre la différence qu'il peut y avoir entre la quantité de référence à une donnée et la quantité d'accès manqué au cache de dernier niveau. La trace permet alors de voir, pour toutes les structures de données instrumentées (sur la pile, sur le tas ou bien globale), la quantité d'accès mémoire, dans quelle région du code ils ont été effectués. Pour utiliser leur outil une instrumentation par le programmeur des structures de données du code source est

nécessaire.

L'outil [Peña and Balaji, 2014a], de Balaji *et al.*, est basé sur Valgrind, pour utiliser le simulateur de mémoire cache de CallGrind [Weidendorfer et al., 2004], et suit le même principe que Gleipnir, basé lui sur Dinero IV [Hill,]. Le fait d'utiliser un outil avec le simulateur de cache intégré permet d'éviter de générer une trace puis de l'analyser après. Les auteurs différencient aussi les objets alloués dynamiquement de ceux alloués statiquement. Cependant, ici l'outil ne nécessite pas d'instrumentation des structures de données dans le code par le programmeur. Les données allouées statiquement sont récupérées en utilisant la méthode implémentée dans Memcheck [Seward and Nethercote, 2005], un autre outil basé sur Valgrind : à partir d'un pointeur d'instruction, la méthode consiste à parcourir la liste des objets contenus dans les informations de debug (standard DWARF, option -g à la compilation), et à extraire l'ensemble des variables dont la portée correspond au pointeur de programme courant du niveau courant de la pile. Les objets de données dynamiques sont identifiées par leur adresse virtuelle. Les informations peuvent néanmoins être agglomérées pour tous les objets alloués sur un appel provenant de la même ligne de code, comme les allocations faites dans une boucle.

L'outil de Sun est basé sur les compteurs matériels. Dans la présentation de l'extension de la suite d'outils de Sun ONE Studio^{TM}, les auteurs montrent qu'avec leur outil ils sont capables de mettre en évidence un problème de performance dans une boucle de calcul à cause des accès aux éléments d'une structure C. Leur outil fonctionne avec une première étape de compilation qui permet de faire le lien entre les variables et les accès mémoire. En ajoutant du padding et en changeant la taille des pages du segment du tas ils obtiennent un gain de 20% en performance sur le benchmark MCF de la suite SPECCPU2000.

Contrairement à [Itzkowitz et al., 2003] et [Pena and Balaji, 2015, Peña and Balaji, 2016, Peña and Balaji, 2014a], ADAMANT [Cicotti and Carrington, 2016] et MemAxes [Giménez et al., 2014] sont open-source. La première étape de l'analyse faite par ADAMANT consiste à identifier les différents objets de l'application. Ils distinguent les objets alloués dynamiquement sur le tas de ceux alloués statiquement dans la section donnée d'un programme (les variables globales ou statiques) et ceux alloués automatiquement sur la pile. À chaque fois qu'une nouvelle allocation dynamique est effectuée, l'objet correspondant est enregistré et identifié à partir de l'adresse virtuelle de son premier élément. En analysant la répartition mémoire du processus, ADAMANT parvient à identifier les objets alloués statiquement (variables globales). Les variables locales allouées sur la pile ne sont pas analysées. Les auteurs considèrent que les objets alloués sur la pile ont un rôle secondaire comparés aux autres dans les accès aux mémoires hors cache. Ce résultat est montré sur une application de parcours d'arbre dans le papier présentant Gleipnir [Janjusic and Kavi, 2013].

Techniques reposant sur ces outils

Dans [Pena and Balaji, 2015], Balaji *et al.* utilisent l'outil qu'ils ont présenté dans [Peña and Balaji, 2014a] afin d'étudier le placement de données sur deux AMH simulées et montrer que le placement de données sur une telle architecture a un impact sur le temps d'exécution d'une application scientifique. Pour cela ils ont développé une méthode pour déterminer le placement de données optimal sur une AMH basée sur un profilage orienté

donnée. Dans le premier scénario l'architecture est constituée d'un scratchpad, d'une mémoire empilée et d'une DDR SDRAM. Dans le deuxième scénario les auteurs y ajoutent une NVM. Les expériences sont menées à chaque fois sur deux mini-applications : MiniMD et HPCCG du projet MANTEVO [Heroux et al., 2009]. Les codes sont interprétés par CallGrind qui permet de récupérer une trace des accès aux différents niveaux de cache ainsi qu'à la mémoire principale. Après quoi les auteurs considèrent que chaque mémoire est uniquement caractérisée par sa latence mémoire. Ils simulent alors chacune des AMH en multipliant la quantité d'accès dans chaque mémoire par la latence qu'ils lui ont attribuée, on appellera temps mémoire cette quantité. Le problème de placement de données est réduit au problème d'allocation et revient donc à allouer les données dans le but de minimiser le temps mémoire. Ne tenant compte que de la latence, leur méthode ne semble pas adapter à une machine comme le KNL contenant deux mémoires avec des latences très proches et des bandes passantes très différentes. Comme nous l'avons vu dans la partie 1.2.1 du chapitre 1, latence et bande passante sont liées mais correspondent à des patterns d'accès mémoires différents. De plus il n'est pas possible de différencier un code sensible à la latence d'un code sensible à la bande passante en observant uniquement le compte total d'accès à la mémoire sur l'intégralité de l'exécution de l'application, comme ils le font.

Dans [Cicotti and Carrington, 2016] les auteurs utilisent ADAMANT afin de mettre en évidence des modifications du code permettant de placer les données pour améliorer les performances de certaines applications sur une AMH qu'ils simulent. Pour cela ils ont étudié trois applications : Velvet, BT et BFS, que nous détaillerons plus tard. Avec les informations fournies par ADAMANT, ils ont été capables de modéliser et de comparer différentes configurations mémoires et placement de données. Pour BFS ils ont trouvé un placement qui surpasse les performances d'une stratégie de cache, en réduisant le ralentissement dû au placement, passant de 1.09x à 1.06x, et sur les autres applications, ils ont mis en évidence les données ralentissant l'exécution de l'application sur les architectures considérées.

Nom	L1	L2	L3	L4	DRAM mémoire principale	NVM mémoire principale
Base	32KB	256 KB	2.5 MB/c	0	4 GB/c	0
4LC	32KB	256 KB	2.5 MB/c	16 MB/c	0	4 GB/c
HMM	32KB	256 KB	2.5 MB/c	0	16 MB/c	4 GB/c
NMM	32KB	256 KB	2.5 MB/c	0	0	4 GB/c

Table 2.2.1 – Configuration des systèmes mémoires. Un 0 signifie que la mémoire n'est pas présente. 4 GB/cœur est suffisant pour contenir l'empreinte mémoire de toutes les applications testées. taille/c signifie taille/cœur

Les 4 architectures simulées sont résumées dans le tableau 2.2.1. Une des configurations ne possède qu'une mémoire DDR SDRAM (nommé Base). Une autre ne possède qu'une mémoire NVM (nommée NMM). Deux configurations sont des AMH composées chacune d'une NVM et d'une DDR SDRAM. Dans l'une des deux la DDR SDRAM est utilisée

comme un cache de dernier niveau (nommée 4LC). Dans l'autre (nommée HMM) la DDR SDRAM et la NVM sont vues comme deux nœuds NUMA distinct et les données peuvent être allouées sur l'une ou l'autre, au choix du programmeur. À chaque fois, lorsque la NVM est simulée, ses caractéristiques sont définies de sorte que chaque accès en lecture est trois fois plus lent et chaque accès en écriture est cinq fois plus lent qu'un accès sur la DDR SDRAM. Les temps d'accès en lecture et en écriture sur la DDR SDRAM sont égaux, et sa capacité est moins importante que celle de la NVM.

Trois mini-application sont émulées sur ces architectures : NAS BT, le noyau BFS de Graph500, et Velvet. Cette dernière mini-application est un code d'assemblage d'ADN conçu pour des lectures courtes [Zerbino and Birney, 2008], il est écrit de sorte que de nombreux petits objets sont alloués. En agglomérant ces objets par leur taille, les auteurs remarquent que la plupart de l'empreinte mémoire et la plupart des références mémoires sont dues aux objets d'une taille unique. En étudiant le code, ils remarquent que ce phénomène est dû au fait que l'application implémente son propre allocateur qui alloue des objets d'une taille fixe. Les auteurs en déduisent donc qu'il est très difficile d'établir une stratégie d'allocation de données sur AMH pour un tel code, étant donné qu'ils sont incapables de différencier les objets de données. Pour ce code le système 4LC fournit un ralentissement de 2.38x contre 2.53x en NMM et seulement 2.48x en HMM, pour la raison invoquée.

Objets	taille (B)	Empreinte mémoire %	Ref%	Mld%	Mst%	Mem%
fields	1705261136	100.0	26.9	100.0	99.9	100.0
[stack]	135168	0.0	38.0	0.0	0.0	0.0
work lhs	84824	0.0	27.1	0.0	0.1	0.0
work ld	13696	0.0	1.7	10.0	0.0	0.0
...	...	...	...	...	...	...

TABLE 2.2.2 – Résumé incomplet des informations extraites avec ADAMANT sur NAS BT. Les objets sont triés par tailles décroissantes. Ref% représente le pourcentage de référence mémoire. Mem% représente le pourcentage d'accès à la mémoire, il est aussi décliné en lecture seulement (Mld) et écriture seulement (Mst).

Le tableau 2.2.2 résume les informations rapportées par ADAMANT sur NAS BT pour une analyse post-mortem. La plupart des références mémoires adressent les trois objets de données les plus volumineux. Mais alors que pour work lhs et [stack] la plupart des références accèdent aux mémoires caches, les accès sur le tableau fields se font en mémoire principale. Cependant, sur le système HMM, l'usage mémoire de fields dépasse la capacité de la DRAM et il est donc nécessaire d'allouer ce tableau en NVM, malgré la latence plus lente de celle-ci comparée à celle de la DRAM. Ainsi ils observent un ralentissement de 2.13 pour les configurations HMM et NMM comparée à la configuration de base (DRAM uniquement). Le ralentissement de la version cache (4LC) reste très élevé à 2.02, même s'il est meilleur que les autres techniques, selon eux ces mauvaises performances sont dues à la mauvaise localité spatial des accès mémoires sur fields.

Objets	taille (MB)	Empreinte mémoire %	Ref %	Mld %	Mst %	Mem %
edgemem	134.2	88.6	0.4	88.0	0.0	73.3
has_edges	4.2	2.8	0.0	10.4	0.0	8.7
pred, g oldq, g newq	4.2	8.2	0.1	1.4	99.5	17.8
...	...	...	...	...	...	...

TABLE 2.2.3 – Résumé des principaux objets alloués dans Graphe500/BFS. Les objets sont triés par tailles décroissantes. Ref% représente le pourcentage de référence mémoire. Mem% représente le pourcentage d'accès à la mémoire, il est aussi décliné en lecture seulement (Mld) et écriture seulement (Mst).

Enfin, le tableau 2.2.3 résume les informations rapportées par ADAMANT sur BFS. Les auteurs remarquent qu'en allouant uniquement has_eges, pred, g oldq et g newq sur la DRAM du système HMM ils obtiennent de meilleures performances (ralentissement de 1.06x) qu'en utilisant le système 4LC (1.09x) ou le système en NMM (1.10x). Ils en concluent donc que pour certaines applications, une stratégie proactive, statique (allocation) pourrait réduire d'avantage le temps d'exécution, qu'une solution réactive dynamique (cache matériel). Néanmoins, la méthode proactive statique montre aussi ses limites : la granularité des décisions peut empêcher de placer les données sensibles aux performances mémoires sur la mémoire avec la meilleure latence (fields dans l'application NAS BT), et l'identification des données par l'appel à la fonction d'allocation dynamique rend impossible la différentiation des objets et donc la prise d'une bonne décision d'allocation (Velvet). Ce sont des problèmes auxquels nous tentons de répondre dans le chapitre 4 implémentation de notre méthode, parties 4.3.

2.3 Conclusion et Limites de l'état de l'art

Le problème de placement de données sur une AMH est nouveau puisqu'il ne s'apparente pas à un problème de placement de données sur une machine NUMA (1.3.3), ni à un problème de placement de donnée sur différents niveaux de mémoires caches ou scratchpad (1.4). Bien que le problème consiste aussi à minimiser le temps d'exécution en plaçant au mieux les données sur les différentes mémoires, maximiser la synergie entre les spécificités des différentes mémoires (caractéristiques de latence, bande passante et capacité) avec les spécificités des régions temporelles et spatiales de l'application scientifique est plus complexe du fait de la plus grande hétérogénéité des caractéristiques mémoires.

On peut distinguer 4 catégories d'approches qui sont opposées deux à deux : les approches statiques (allocation de données) ou dynamiques (placement de données) et celles qui sont proactives ou réactives. Les travaux effectués sur les mémoires scratchpad concluent sur la supériorité des performances atteintes avec une approche dynamique, parce que celles-ci peuvent tenir compte des changements de comportement de l'application dans son exécution [Guo et al., 2011, Udayakumaran and Barua, 2003]. De ces travaux il ressort aussi, comme de ceux effectués sur une AMH, que les approches proactives sont

plus performantes que celles qui sont réactives parce qu'elles permettent de tenir compte des spécificités propres à chaque application. Finalement, pour placer au mieux les données sur une AMH une approche proactive et dynamique semble être une solution intéressante.

Les approches proactives reposant sur une technique logicielle utilisent principalement le compte de LLCmiss sur la totalité de l'exécution de l'application comme métrique afin de placer les données, alors que cette métrique ne suffit pas à rendre compte de tout le trafic mémoire, 1.2. De plus, afin de maximiser la synergie entre les spécificités du code et celles des mémoires, il est important de caractériser les besoins en bande passante et en latence de l'application pour chacune de ses données. Nous avons donc choisi de développer un outil de profilage adapté à ce qu'on souhaite extraire comme information. Dans notre démarche nous nous sommes concentrés sur l'amélioration des approches proactives, basées sur une technique logicielle, et cherchant à résoudre le problème d'allocation de données, statique. Il est probable qu'une approche proactive, mais dynamique, puisse donner de meilleurs résultats, mais le problème à poser est plus complexe, puisqu'il faut prendre en compte les coûts de déplacement de données entre les différentes mémoires. Notre étude consiste donc à développer une méthode de *profilage et d'allocation* de données pour une architecture AMH.

Au cours de ma thèse, d'autres travaux ont été effectués en parallèle dans le même but d'automatiser l'allocation et/ou le placement de données sur une architecture à mémoires hétérogènes. Dans [Servat et al., 2017] les auteurs développent une boîte à outils automatique de profilage et d'allocation de données sur KNL. Dans [Wu et al., 2017b, Wu et al., 2017a] les auteurs développent un support exécutif de déplacement de données sur une architecture composée d'une mémoire principale supplémentaire qui est non volatile. Nous discuterons des différences entre leurs travaux et les nôtres dans les chapitres de contributions concernés.

Chapitre 3

Formulation et résolution du problème d'optimisation

Comme nous l'avons souligné dans l'introduction, l'un des enjeux majeurs pour les acteurs du Calcul Haute Performance est la réduction du temps d'exécution des applications scientifiques, et cette course à la performance entraîne l'émergence d'architectures à mémoires hétérogènes. Sur de telles architectures nous avons vu qu'il est nécessaire d'allouer intelligemment les données accédées par l'application afin d'optimiser l'utilisation des différentes caractéristiques des différentes mémoires. L'état de l'art met en évidence deux catégories d'approches dans la recherche d'une allocation de données optimale sur les différentes mémoires d'un système. L'approche proactive nous semble intéressante puisqu'elle peut bénéficier des spécificités de l'application dans la résolution du problème d'optimisation, contrairement à une approche réactive. Les formulations du problème d'allocation présentes dans la littérature ne permettent pas de tenir compte du temps d'exécution total de l'application dans l'évaluation des coefficients de la fonction objective. C'est pourquoi nous commençons par développer une formulation du problème d'optimisation avant de chercher à le résoudre.

3.1 Formulations du problème d'allocation

Nous commençons par présenter une forme non linéaire de la formulation du problème d'allocation mettant en évidence le lien avec le temps d'exécution, puis nous présentons une linéarisation de ce problème sous certaines conditions.

3.1.1 Formulation non linéaire

Ici nous cherchons à formuler le problème de sorte à rendre évident l'aspect temporel. Pour cela considérons une application allouant N données et s'exécutant sur une AMH composée de M mémoires, la *décision d'allocation* de la $i^{ème}$ donnée sur la $j^{ème}$ mémoire est représentée par la paire (i, j). On définit une *stratégie d'allocation* comme l'ensemble des décisions d'allocation où chacune des N données du programme est présente exactement une fois en association avec une unique mémoire, c'est la contrainte d'unicité. Cette contrainte est une hypothèse permettant de simplifier le problème, puisque nous ne considérons pas la possibilité de duplication d'une donnée sur plusieurs mémoires et les problèmes de cohérence que cela implique.

Chaque stratégie d'allocation possède le même nombre d'éléments égal à N. Soit $\mathcal{F}$ l'ensemble contenant toutes les stratégies d'allocation. Sur un système à M mémoires, et un programme contenant N données à être allouée, $\mathcal{F}$ contient M^N stratégies d'allocation. Cet ensemble représente toutes les stratégies d'allocation possible pour toutes les données du programme sur toutes les mémoires. Résoudre le problème d'allocation consiste à trouver la stratégie d'allocation dans cet ensemble, minimisant le temps d'exécution de l'application, tout en respectant les contraintes de capacité de chaque mémoire à tout instant de l'exécution du programme.

Dans la formulation 3.1, $\mathcal{T}$ retourne le temps d'exécution du programme en fonction d'une stratégie d'allocation, respectant la contrainte d'unicité. Cette fonction n'est pas linéaire, car le lien entre latence et bande passante étant complexe (partie 1.2 de l'introduction et partie 3.2.1, page 47), les performances obtenues avec l'allocation d'une donnée sur une mémoire dépendent du pattern d'accès à cette donnée, mais aussi des accès simultanés aux autres données sur cette même mémoire. La fonction ${\mathcal{L}_{\text{usageMémoire}}}^j$ retourne le pic d'usage mémoire de l'application sur la mémoire j considérée, elle est linéaire. Ces deux fonctions dépendent des paramètres d'entrée du programme à exécuter : taille du problème traité, nombre d'itérations (si itératif), choix des algorithmes, etc. Enfin, $\mathcal{C}_j$ représente la capacité mémoire de la $j^{ème}$ mémoire.

$$\begin{array}{ll} \min\limits_{s \in \mathcal{F}} & \mathcal{T}(s) \\ \text{sujet à} & \forall j \in [\![1; M]\!], {\mathcal{L}_{\text{usageMémoire}}}^j(s) < \mathcal{C}_j \end{array} \tag{3.1}$$

3.1.2 Formulation linéaire

La fonction objective

Commençons par définir un temps de référence. Soit une stratégie d'allocation a faite de décisions d'allocation, et t_a le temps d'exécution de l'application dont les données sont allouées en fonction de cette stratégie. Afin de comparer différentes stratégies d'allocation, les paramètres d'entrée du programme, influençant le temps d'exécution de l'application, doivent être maintenus constant, donc nous considérons qu'ils sont fixés. Supposons qu'une mémoire ait la capacité pour contenir toutes les données du programme à tout instant, c'est-à-dire que le pic d'usage mémoire de l'application ne dépasse pas la capacité de cette mémoire. On appelle cette mémoire la mémoire *principale*, c'est la $M^{ème}$ mémoire

de l'AMH. L'équation 3.2 représente le temps d'exécution de référence t, c'est le temps correspondant à la stratégie composée des décisions allouant leur donnée sur la mémoire principale.

$$t = t_{\{(1,M),(2,M),...,(i,M),...(N,M)\}} \tag{3.2}$$

On définit alors $a_{[ij]}$ comme étant la stratégie d'allocation dans laquelle la donnée i est allouée sur la mémoire j et toutes les autres données sont allouées sur la mémoire principale. Dans cette notation seule une décision est explicitée, les autres sont implicites. Donc, $t_{[ij]}$ est le temps associé avec la stratégie d'allocation $a_{[ij]}$ (voir équation 3.3). Remarquons que si $j = M$, alors $t_{[ij]} = t$.

$$t_{[ij]} = t_{\{(1,M),(2,M),...,(i,j),...(N,M)\}} \tag{3.3}$$

En supposant que $N > 2$ et en suivant les mêmes notations, $t_{[ij_i],[i'j_{i'}]}$ est donc le temps associé à la stratégie d'allocation $a_{[ij_i],[i'j_{i'}]}$. Dans cette stratégie d'allocation la donnée i est allouée sur la mémoire j_i et la donnée i' sur la mémoire $j_{i'}$ (voir équation 3.4). Ici encore, toutes les autres données sont allouées sur la mémoire principale. Notons que $i \neq i'$ mais j_i et $j_{i'}$ peuvent identifier la même mémoire.

$$t_{[ij_i],[i'j_{i'}]} = t_{\{(1,M),(2,M),...,(i,j_i),...,(i',j_{i'}),...(N,M)\}} \tag{3.4}$$

On définit alors un *choix* comme étant une décision d'allocation explicitée. Notons que la décision consistant à allouer toutes les données sur la mémoire principale, qui est la décision par défaut, peut aussi être un choix. Le nombre d'éléments explicités dans une stratégie d'allocation sont donc les *choix*, ceux qui ne sont pas explicités sont sur la mémoire principale par défaut. Par exemple, le nombre de choix dans $a_{[ij],[i'j']}$ est égal à 2. En généralisant à K choix, on peut écrire le temps d'exécution de n'importe quelle stratégie d'allocation a comme décrit dans l'équation 3.5.

$$\begin{aligned} t_a &= t_{[1j_1],[2j_2],...,[ij_i],...,[Kj_K]} \\ &= t_{\{(1,j_1),(2,j_2),...,(i,j_i),...,(K,j_K),(K+1,M),...(N,M)\}} \end{aligned} \tag{3.5}$$

La variation du temps d'exécution induite par l'allocation de la donnée i sur la mémoire j, par rapport à son allocation par défaut sur la mémoire principale, s'écrit $t - t_{[ij]}$. Afin de construire une **formulation linéaire** du problème d'allocation nous faisons l'hypothèse que les performances des accès mémoires sur un objet de données sont indépendantes des accès aux autres objets de données alloués sur cette mémoire. Précisément, cela signifie que nous ignorons les effets d'utilisation simultanée de bande passante mémoire par des accès à différentes données. Nous introduirons les limites de cette hypothèse dans la discussion

(partie 3.1.3), puis nous proposerons une extension possible de la formulation pour étendre ces limites dans la partie 3.2.

$$\begin{aligned} t_{[ij_i],[i'j_{i'}]} &= t - (t - t_{[ij_i]}) - (t - t_{[i'j_{i'}]}) \\ &= t_{[ij_i]} - (t - t_{[i'j_{i'}]}) \\ &= t_{[ij_i]} + t_{[i'j_{i'}]} - t \times (2-1) \end{aligned} \tag{3.6}$$

Sous cette hypothèse, l'équation 3.6 montre que le temps d'exécution $t_{[ij],[i'j']}$, associé avec la stratégie d'allocation $a_{[ij],[i'j']}$, peut être exprimé comme une fonction linéaire du temps d'exécution des deux stratégies d'allocation $a_{[ij]}$ et $a_{[i'j']}$. Ce résultat peut se généraliser à K comme montré dans le lemme 3.1.1.

Lemma 3.1.1 *En considérant une application qui alloue N données et une stratégie d'allocation a faisant K choix, avec $K \leq N$,*

$$t_a = \sum_{i=1}^{K} (t_{[ij_i]}) - t \times (K-1)$$

Preuve Preuve par récurrence, initialisation :

If $K = 0$,
$t_a = t$, par définition toutes les données sont sur la mémoire principale
$= -t \times (K-1)$
If $K = 1$,
$t_a = t_{[1j_1]}$, par définition
$= t_{[1j_1]} - t \times (K-1)$
If $K = 2$,
$t_a = \sum_{i=1}^{K=2} (t_{[ij_i]}) - t \times (K-1)$, depuis l'équation 3.6

Soit $N > 2$, et supposons l'hypothèse de récurrence vraie pour $K < N$,
Montrons que l'hypothèse reste vraie pour $(K+1) \leq N$,

$$t_{a'} = t_{[1j_1],[2j_2],\ldots,[ij_i],\ldots,[(K+1)j_{K+1}]}$$

Ici on utilise l'hypothèse qu'on a faite sur l'indépendance des performances de traitement des accès mémoires simultanés sur les différents objets de données alloués sur une même

mémoire.

$$\begin{aligned}
t_{a'} &= t - (t - t_{[1j_1],[2j_2],\ldots,[ij_i],\ldots,[Kj_K]}) - (t - t_{[(K+1)j_{K+1}]}) \\
&= t - (t - \sum_{i=1}^{K}(t_{[ij_i]})) - t \times (K-1) - (t - t_{[(K+1)j_{K+1}]}) \\
&= \sum_{i=1}^{K}(t_{[ij_i]}) - t \times (K-1) - (t - t_{[(K+1)j_{K+1}]}) \\
&= \sum_{i=1}^{K}(t_{[ij_i]}) - t \times (K-1) + t_{[(K+1)j_{K+1}]} - t \\
&= \sum_{i=1}^{K+1}(t_{[ij_i]}) - t \times (K-1) - t \\
&= \sum_{i=1}^{K+1}(t_{[ij_i]}) - t \times ((K+1)-1)
\end{aligned}$$

$$\min_{c \in \mathcal{F}} \quad \sum_{(ij) \in c} t_{[ij]} \tag{3.7}$$

On peut écrire la fonction objective en étendant la définition à $K = N$, et en simplifiant les termes qui ne dépendent pas de $[ij]$. La formulation de la fonction objective du problème peut être simplifiée comme le montre l'équation 3.7. Finalement, on peut définir la variable de décision $x_{[ij]}$ qui est égale à 1 si $t_{[ij]}$ fait partie de la solution, 0 sinon. Que $t_{[ij]}$ fasse partie de la solution signifie que la décision d'allocation (i, j) fait partie de la stratégie d'allocation. La fonction objective devient donc :

$$\min_{x_{[ij]}} \quad \sum_{j=1}^{M}\sum_{i=1}^{N} t_{[ij]} \times x_{[ij]} \tag{3.8}$$

Contrainte d'unicité de la donnée

Comme formulé dans la partie 3.1.2, chacune des N données de l'application doit être associée une et une seule fois à chaque mémoire. Pour cela, la somme des variables de décision associées à chaque donnée doit être égale à 1 comme représenté dans l'équation 3.9.

$$\forall i \in [\![1; N]\!], \sum_{j=1}^{M} x_{[ij]} = 1 \tag{3.9}$$

Contrainte sur la capacité mémoire

À tout instant de l'exécution, l'usage mémoire du programme ne doit pas dépasser la capacité de la mémoire stockant ces données. Surtout dans le domaine du HPC, car sur certains supercalculateurs les mécanismes de swap sont désactivés. Pour chaque groupe de variables allouées sur une même mémoire et pour lesquelles il existe un instant où elles sont toutes en vie, leurs usages mémoires ne doivent pas dépasser la capacité de la mémoire sur laquelle elles sont allouées. Sur la figure 3.1.1 on peut voir notamment neuf groupes de données vivantes simultanément : {A},{A,D},{A,D,C},{A,D,C,B},{D,C,B},{C,B},{B},{B,E},

et {E}. Pour chacun de ces deux groupes, la somme des usages mémoires des données allouées sur la même mémoire ne doivent pas dépasser la capacité mémoire. Soit $\mathcal{E}$ l'ensemble

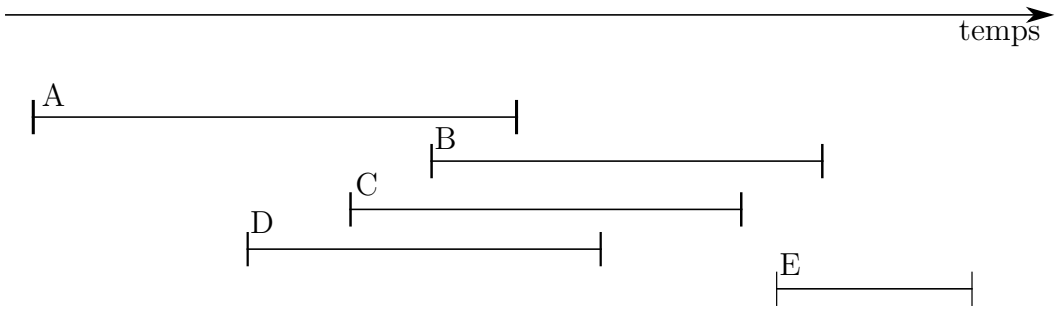


FIGURE 3.1.1 – Schémas de temps de vie des données

qui contient tous ces groupes de données. On définit $q_{i,l}$ l'opérateur booléen qui est à vrai si et seulement si la donnée i appartient au groupe l comme écrit dans l'équation 3.10.

$$\forall i \in [\![1; N]\!], \forall l \in \mathcal{E},$$
$$q_{i,l} = \begin{cases} 0 \text{ Si la donnée } i \notin l \\ 1 \text{ Si la donnée } i \in l \end{cases} . \tag{3.10}$$

En considérant que m_i est le pic d'usage mémoire de la donnée i et $\mathcal{C}_j$ est la capacité de la mémoire j, alors la somme de tous les usages mémoires des données m_i allouées sur la mémoire j, appartenant au même groupe l ne doivent pas dépasser la capacité mémoire $\mathcal{C}_j$ afin de respecter la contrainte de capacité, comme écrit dans l'équation 3.11.

$$\forall j \in [\![1; M]\!], \forall l \in \mathcal{E}, \sum_{i=1}^{N} q_{i,l} \times x_{[ij]} \times m_i < \mathcal{C}_j \tag{3.11}$$

Cette formulation (3.11) respecte la contrainte de capacité mémoire à tout instant de l'exécution de l'application pour tous les groupes. Cependant, si la contrainte est respectée par un groupe, tous les groupes inclus dans celui-ci respectent aussi la contrainte. Donc, l'ensemble $\mathcal{E}$ à considérer peut-être plus petit et continuerait toujours de garantir que la contrainte mémoire soit respectée. Cette propriété est démontrée ici. On définit l'ensemble $\mathcal{D}$ qui contient tous les groupes qui ne sont inclus dans aucun autre groupe (équation 3.12). $\mathcal{D}$ peut remplacer $\mathcal{E}$ dans l'équation 3.11.

$$\mathcal{D} = \{K \in \mathcal{E}, \nexists J \in \mathcal{E} \setminus \{K\},\ K \subset J\} \tag{3.12}$$

Preuve Considérons $X \in \mathcal{E}$ et $Y \in \mathcal{E}$, tels que $X \subset Y$. L'usage mémoire d'un ensemble de données est égal à la somme des usages mémoire des données qui le composent, parce que, par définition, leurs temps de vie se chevauchent. Il existe donc un instant pour lequel elles sont toutes en mémoire en même temps. De plus, la somme des usages mémoire d'un ensemble de données est une fonction croissante en la taille de cet ensemble, parce que l'usage mémoire est toujours positif. Donc, l'usage mémoire de Y est égal à l'usage mémoire de X plus l'usage mémoire $Y \setminus X$. Finalement, la contrainte de capacité mémoire est respectée par Y, elle est donc respectée par X qui est plus petit.

Algorithm 1: Calcul de $\mathcal{D}$

```
Input: Donnée allouée dynamiquement
Output: D
EnsembleDeDonnéeVivante ← EnsembleVide() ;
D ← EnsembleVide() ;
Function CaptureAllocation
    Data: DonnéeCourante
    EnsembleDeDonnéeVivante.add(DonnéeCourante) ;
Function CaptureLibération
    Data: DonnéeCourante
    Booléen AjouterEnsemble ← Vrai;
    foreach EnsembleDeDonnée ∈ D do
        if EnsembleDeDonnéeVivante⊆ EnsembleDeDonnée then
            AjouterEnsemble ← Faux;
            SortirDeLaBoucle() ;
        end
    end
    if AjouterEnsemble then
        D.add(EnsembleDeDonnéeVivante.copy()) ;
    end
    EnsembleDeDonnéeVivante.supprimer(DonnéeCourante) ;
```

L'ensemble $\mathcal{D}$ peut être construit dynamiquement ou statiquement. L'algorithme 1 décrit sa construction, en capturant les appels de fonctions effectuant les allocations et libérations dynamiques. La méthode de construction statique pourrait consister à considérer tous les chemins d'exécution possibles. Pour être correcte, les vies de données considérées doivent être celles qui engendrent le plus de chevauchement avec les autres données.

Conclusion

$$\begin{aligned} \min_{x_{[ij]}} \quad & \sum_{j=1}^{M}\sum_{i=1}^{N} t_{[ij]} \times x_{[ij]} \\ \text{sujet à} \quad & \forall j \in [\![1; M]\!], \forall l \in \mathcal{D}, \sum_{i=1}^{N} q_{i,l} \times x_{[ij]} \times m_i < \mathcal{C}_j, \\ & \forall j \in [\![1; M]\!], \sum_{i=1}^{N} x_{[ij]} = 1 \end{aligned} \tag{3.13}$$

Finalement, la formulation linéaire complète du problème d'allocation de données (cf équation 3.20) consiste à minimiser le temps d'exécution en prenant les meilleures décisions d'allocation respectant la contrainte de capacité mémoire et celle d'unicité de l'allocation. Dans cette formulation, les $x_{[ij]}$ sont les inconnues, variables du système. Les coefficients $t_{[ij]}$ doivent être mesurés, ou prédits, ils représentent le temps d'exécution total de l'application étant donné une décision d'allocation avec la donnée i sur la mémoire j et toutes les autres données sur la mémoire M, mémoire principale fixée arbitrairement. Les coefficients $q_{i,l}$ représentent l'appartenance des données aux groupes d'interférences afin de respecter les contraintes de capacité de chacune des mémoires du système. Enfin, les coefficients m_i et $\mathcal{C}_j$ représentent respectivement l'usage mémoire de la donnée i et la capacité mémoire de la mémoire j.

3.1.3 Limites de la formulation linéaire

Notre formulation repose sur une approche proactive et dépend d'une analyse spatiale dans l'évaluation des coefficients de la fonction objective et des contraintes. La granularité spatiale de cette formulation est limitée par la représentation des données. Cependant, à partir d'un allocateur mémoire il est théoriquement possible de découper les données à la granularité spatiale souhaitée. La granularité temporelle de la solution, quant à elle, est limitée par le caractère statique de l'approche. En effet, pour atteindre une meilleure solution, il est nécessaire de différencier le temps pendant lequel une donnée est accédée du temps pendant lequel elle est vivante. Cette différentiation n'est possible qu'en ayant la possibilité de déplacer une donnée. Pour cela, il serait nécessaire de prendre en compte le surcout dû au déplacement des données, et d'avoir un modèle de prédiction sur la rentabilité du déplacement [Wu et al., 2017a].

Dans certains cas de figure, l'hypothèse que nous avons faite pour écrire le problème sous une forme linéaire ne permet pas d'obtenir la stratégie de placement de données optimale. Si on considère, par exemple, un système composé de plusieurs mémoires identiques, il est impossible de bénéficier de la totalité de la bande passante disponible, égale à la somme des bandes passantes des mémoires, avec notre formulation. En effet, soit une application contenant des données accédées uniformément, et simultanément, sensibles à la bande passante mémoire, s'exécutant sur un système composé de deux mémoires identiques, numérotées 0 et 1, possédant chacune assez de capacité mémoire pour l'usage total de l'application. Alors l'ensemble des $t_{[i0]}$ seront tous égaux entre eux et strictement inférieurs

à t (temps d'exécution avec toutes les données sur la mémoire principale 1), car l'allocation de n'importe quelle donnée (i) sur l'autre mémoire (mémoire 0) permet de faire un meilleur usage de la bande passante totale disponible. Ainsi, la solution du problème linéaire, tel que formulé sous nos hypothèses, consiste à mettre toutes les données sur la mémoire 0, si la mémoire 1 est arbitrairement sélectionnée comme mémoire principale, ou bien réciproquement à tout allouer sur la mémoire 1.

Finalement, alors que la solution optimale serait de répartir équitablement les données entre les deux mémoires, pour équilibrer l'usage de bande passante, notre formulation linéaire ne permet pas d'aboutir à cette solution, parce que l'hypothèse que nous avons faite ne rends pas compte de la dépendance des performances des accès mémoires à un objet de données, aux accès mémoires simultanés à d'autres objets de données. Ceci est une limite de notre modèle que nous ne résoudrons pas, cependant dans la partie suivante (3.2), nous cherchons à étendre cette limite en complétant l'approche spatiale avec une approche temporelle permettant d'affiner la prédiction des coefficients de la fonction objective.

3.1.4 Conclusion

Dans cette partie, nous avons introduit la notion de temps d'exécution de l'application en fonction d'une stratégie d'allocations. Nous avons décrit une formulation non linéaire permettant de résoudre le problème d'allocation de données avec une approche proactive et statique. Nous avons montré que le problème d'allocation de données sur une AMH peut être formulé linéairement sous l'hypothèse d'indépendance des performances de traitement des accès mémoires simultanés sur les différents objets de données alloués sur une même mémoire. Cette hypothèse implique certaines limites à la résolution du problème initial avec cette formulation linéaire, mais nous verrons dans le chapitre 5 que malgré ces limites il est possible d'obtenir des résultats encourageants en pratique.

3.2 Analyse temporelle

Dans cette partie, le but est d'affiner le modèle pour réduire les limites dues aux hypothèses faites dans la construction de la formulation linéaire introduite dans la partie précédente. Pour cela, nous développons une analyse temporelle afin de compléter la formulation précédente du problème linéaire en nombres entiers, et d'évaluer les coefficients de la fonction objective. Ensuite, nous détaillons le premier pas de notre heuristique : sélectionner les régions du code (régions temporelles) qui sont pertinentes dans le choix de la stratégie d'allocation de données.

3.2.1 Intérêt d'une analyse temporelle

Sur un programme s'exécutant pendant 7 seconde, une bande passante de 4GB/s peut-être mesurée sur la totalité de l'exécution alors qu'une portion d'une seconde du programme utilise une bande passante de 16GB/s et que le reste du programme (6 secondes) ne transfère qu'à un taux de 2GB/s. C'est pourquoi il faut découper le programme en région temporelle avec une granularité suffisamment fine pour capturer ces changements de bande passante utilisée, repérer quelles sont les régions du code utilisant assez de bande passante pour que les performances du code y soient sensible, et extraire les données

accédées dans ces régions afin de les placer sur la mémoire du système fournissant le plus de bande passante. Finalement, mesurer la bande passante utilisée d'un programme à partir de son temps d'exécution total ne donne pas suffisamment d'information pour analyser correctement la sensibilité à la bande passante des objets de données.

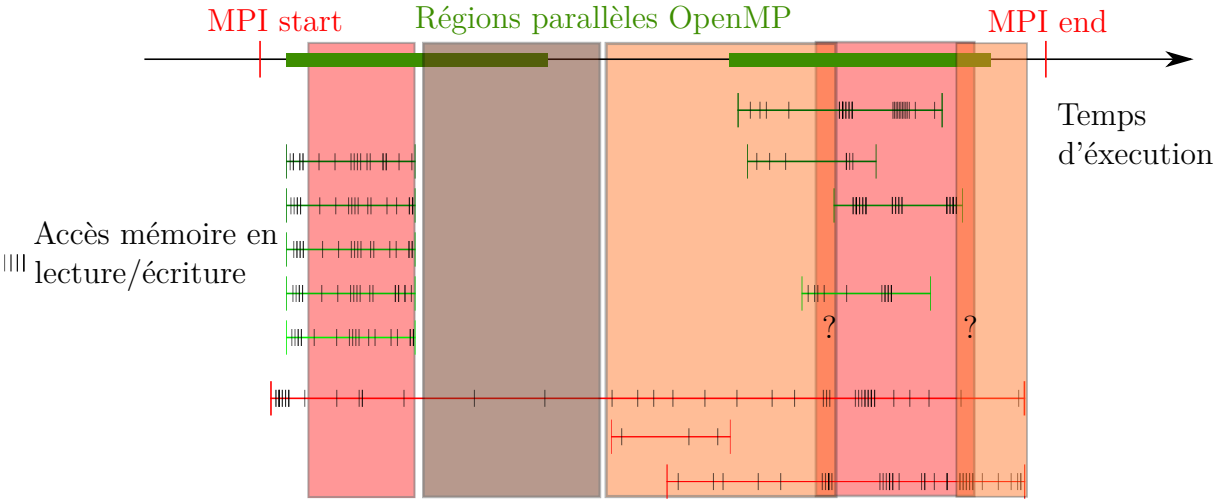


FIGURE 3.2.1 – Schéma d'accès à la mémoire sur une application MPI+OpenMP

La figure 3.2.1 représente l'exécution d'une application scientifique utilisant les bibliothèques MPI pour l'envoi de message et OpenMP pour le calcul parallèle en mémoire partagée. Le temps s'écoule de gauche à droite, chaque segment représente le temps de vie d'une donnée générée par l'exécution et les petits traits notent l'emplacement temporel (l'instant) d'un accès à la mémoire en lecture ou en écriture. Ici, l'aspect spatial n'a pas d'importance puisqu'on considère déjà que chacun de ces accès représente un accès à la mémoire principale. La congestion temporelle et spatiale du système mémoire est due en partie aux canaux et aux bancs mémoire, mais cette congestion est rapidement limitée par celle essentiellement temporelle qui est due aux listes globales du contrôleur mémoire [Eklov et al., 2012].

Ce code est constitué de différentes parties avec des patterns temporels d'accès à la mémoire très différents et ne pouvant pas être distinguées en considérant uniquement les accès à la mémoire par objet de données sur l'ensemble de l'application. En effet on observe que ce sont les accès conjugués de tous les objets de données qui influencent simultanément la nature de la sensibilité d'une région temporelle. Les zones rouges, oranges et marrons représentent respectivement des zones temporelles sensibles à la bande passante, à la latence, et au calcul lorsque la zone n'est sensible à aucune caractéristique mémoire.

La frontière entre une zone sensible à la bande passante et une zone sensible au calcul est simple à délimiter. Du point de vue du Roofline model (cf partie 1.1.3, page 8) cela correspond à comparer l'intensité arithmétique de la région du code avec l'intensité arithmétique qui correspond au point d'équilibre de la machine. La frontière entre une zone sensible à la bande passante et une sensible à la latence mémoire est plus complexe à déterminer. Comme nous l'avons vu dans le chapitre 2, les travaux effectués jusqu'ici

considèrent le nombre d'accès à la mémoire par objet de données, uniquement de manière individuelle. Cette métrique est efficace lorsque les accès mémoires ne viennent pas trop congestionner le système de traitement des requêtes mémoires. C'est-à-dire lorsque l'application est sensible aux performances de latence mémoire et non de bande passante mémoire. C'est une métrique qui est donc adaptée pour le placement de données sur une architecture faites de mémoires avec des latences très différentes, mais pas sur une architecture contenant une mémoire à forte bande passante.

3.2.2 Extension de la formulation pour une analyse temporelle

Dans certaines AMH la bande passante mémoire est une caractéristique importante pour différencier les différentes mémoires. Afin de mesurer l'influence relative de chacune des données accédées par l'application sur ses besoins en bande passante, et parce que la bande passante est une caractéristique définie par la fréquence des accès mémoire par unité de temps, il est nécessaire de développer une ou plusieurs métriques temporelles.

Nous appelons r une région temporelle obtenue lors du découpage de l'exécution du programme. Chaque région correspond à une partie séquentielle du programme, ou bien une partie parallèle encadrée par des synchronisations. Le temps d'exécution du programme est donc $t = \sum_{r \in \mathcal{R}} t^r$. Les régions ne sont pas découpées de manières égales, et chaque région possède un poids différent sur le temps d'exécution total.

On peut écrire la fonction objective en tenant compte des régions temporelles (cf équation 3.14).

$$\min_{x_{[ij]}} \quad \sum_{j=1}^{M} \sum_{i=1}^{N} (x_{[ij]} \times \sum_{r \in \mathcal{R}} t^r_{[ij]}) \tag{3.14}$$

Quelles que soient les caractéristiques de latence et de bande passante de la mémoire étudiée, les coefficients de la fonction objective peuvent être évalués en plaçant une à une chaque donnée sur la mémoire : on teste ainsi toutes les combinaisons possibles. Mais cette méthode n'est pas envisageable à cause de la quantité d'exécution à effectuer au cours du profilage. On peut tenter d'avoir une information moins précise avec uniquement deux exécutions. Pour une taille de problème dont l'usage mémoire induit ne dépasse pas la capacité mémoire de chacune des mémoires, on peut exécuter l'application et mesurer ses différentes régions temporelles avec toutes les données sur la mémoire étudiée. En faisant autant d'exécution qu'il y a de mémoires dans l'architecture, on peut calculer les $t^r_{[1j][2j]...[Nj]}$ pour tout $j \in [\![1; M]\!]$. On mesure l'impact sur les performances de l'application de l'allocation de toutes les données de l'application sur une mémoire en particulier, uniquement pendant le temps d'une région temporelle. Après quoi, il faut faire correspondre ces régions avec les données qui y sont accédées.

3.2.3 Sélection de régions de code

Nous avons décidé de ne pas étudier toutes les régions du code d'une application afin de simplifier son analyse. La sélection des régions du code se fait en fonction des métriques choisies. Nous utilisons une métrique classique, celle utilisée pour le profilage temporel mettant en évidence les régions temporelles "chaudes" du code, celles dont le temps dépasse

un temps minimal dépendant d'un seuil, obtenu par un ratio arbitraire (< 1) du temps total d'exécution. On appelle $\mathcal{H} \subset \mathcal{R}$ l'ensemble de ces régions considérées comme non négligeable dans le temps d'exécution total de l'application. À partir de cette décision, la méthode que nous développons devient une heuristique, et selon le seuil d'influence globale choisi pour ignorer certaines régions, la solution trouvée peut varier.

Dans notre méthode nous couplons aussi cette métrique avec un profilage temporel centré sur la caractéristique de bande passante mémoire. Cette caractéristique est définie par la fréquence des accès, c'est dont avec des mesures locales qu'on peut obtenir un profilage de sensibilité à la bande passante précis de l'application. Une fois ce profilage obtenu, les régions temporelles sensibles à la bande passante, $\mathcal{B} \subset \mathcal{R}$, dépendant d'un seuil de sensibilité qui dépend des caractéristiques du système mémoire, sont sélectionnées afin d'étudier uniquement les données qui y sont accédées. On peut alors écrire l'équation 3.14 en tenant compte de cette réduction des régions considérées (équation 3.15).

$$\min_{x_{[ij]}} \quad \sum_{j=1}^{M} \sum_{i=1}^{N} (x_{[ij]} \times \sum_{r \in \mathcal{H} \cap \mathcal{B}} t^r_{[ij]}) \tag{3.15}$$

3.3 Résolution du problème d'optimisation

Afin d'attribuer une valeur à chaque objet ($t_{[ij]}$), on s'appuie sur une analyse temporelle sélectionnant des régions et calculant les $t^r_{[1j]\ldots[Nj]}$ correspondant, complétée par une analyse spatiale. La sélection des régions se fait en considérant que le système mémoire est composé d'une mémoire principale avec une capacité mémoire supérieure à l'usage mémoire total, une meilleure latence et une bande passante moins bonne, et une autre mémoire avec une capacité mémoire inférieure à l'usage mémoire total une meilleure bande passante mais une latence plus lente. Ainsi t correspond au temps d'exécution total avec toutes les données allouées sur la mémoire principale, à faible bande passante. Les régions sélectionnées, sensibles à la bande passante, s'exécuteraient plus rapidement si les données accédées dans ces régions étaient placées sur l'autre mémoire. Dans cette partie, nous détaillons les problèmes posés par une approche basée sur une analyse temporelle puis spatiale, et les solutions que nous proposons.

Si N est l'ensemble des données, alors il est égal à l'union de l'ensemble W des données accédées dans les régions de code sélectionnées, de l'ensemble U des données accédées uniquement en dehors de ces régions et de l'ensemble V des données accédées dans les deux types de régions (équation 3.16).

$$N = W \cup U \cup V \tag{3.16}$$

Les données exclusivement accédées dans ou hors des régions de code sélectionnées par l'analyse temporelle sont les plus faciles à gérer. Celles qui sont accédées uniquement en dehors ne sont simplement pas considérées comme des données à être allouées sur une mémoire avec de meilleures caractéristiques que la mémoire principale. En effet pour chacune de ces données, soit le gain de performance serait négligeable (région froide du code) ou bien il n'y aurait aucun effet, voire un ralentissement du temps d'exécution

parce que la région dans laquelle la donnée est accédée n'est pas sensible à la bande passante. Finalement U est exclu de l'ensemble des données à considérer pour résoudre le problème linéaire en nombres entiers, toutes les données de cet ensemble seront allouées dans la mémoire principale dont la capacité est suffisante pour l'usage mémoire maximal de l'application. De plus, l'ensemble $\mathcal{D}$ définissant la contrainte de capacité mémoire est aussi réduit, puisque ces données ne seront pas considérées dans la construction des groupes des données dont les temps de vie interfèrent. Au contraire, les données qui sont uniquement accédées dans les zones sélectionnées (W) sont évaluées comme significative dans la résolution du problème.

Les données accédées à la fois dans des zones sélectionnées et dans des régions non-sélectionnées posent plus de difficultés. Une heuristique conservatrice consiste à enlever ces données de l'espace de recherche d'une stratégie, ce qui signifie qu'elles seront aussi laissées sur la mémoire principale. Puisque ces données sont accédées dans des régions du code qui ne sont pas sélectionnées, ces accès peuvent entraîner un ralentissement du temps d'exécution. Idéalement, on voudrait avoir la donnée qui reste cohérente sur les deux mémoires, avec les accès dans les régions sélectionnées sur une mémoire et les autres accès sur l'autre mémoire. En ne considérant plus ces données, le nouvel ensemble de données N_s considérée pour la résolution du problème est uniquement composé des données accédées exclusivement dans les régions de code sélectionnées, c'est-à-dire que $N_s = W$. Cependant, cela peut amener à ne pas considérer des données pouvant apporter une accélération à l'application si elles sont allouées sur la mémoire à forte bande passante.

Pour éviter cela, on considère le nouvel ensemble composé de toutes les données accédées au moins une fois dans une région du code sélectionnée, c'est l'ensemble $N_s = W \cup V$, et on va essayer d'enlever de cet ensemble les données pour lesquels une allocation hors de la mémoire principale n'apporterait pas d'accélération à l'application. Concentrons nous sur une de ces données, la donnée i. L'exécution du programme peut alors être découpée en trois parties. Le temps x_i passé dans l'union des régions du code sélectionnées dans lesquelles i est accédée, le temps y_i passé dans l'union des régions du code non sélectionnées dans lesquelles i est accédées et enfin z_i le temps passé dans les régions du code (sélectionnées ou non) dans lesquelles i n'est pas accédée. On peut alors exprimer le temps d'exécution total de l'application comme : $t_i = x_i + y_i + z_i$. Considérant une des mémoires de notre AMH différente de la mémoire principale, on définit S comme l'accélération maximale atteignable (le meilleur) et s comme le ralentissement minimal atteignable (le pire). D'une part, les accès mémoires dans les régions non sélectionnées ralentissent l'application par un facteur s au pire. D'autre part, ceux effectués dans les régions non sélectionnées accélère l'application par un facteur S au mieux. En allouant la donnée i sur une de ces mémoires, le temps d'exécution du programme peut s'exprimer comme une fonction de x_i, y_i, z_i, s et S (équation 3.17).

$$T_i = x_i/S + y_i/s + z_i \tag{3.17}$$

Donc, pour garder cette donnée parmi l'ensemble des données à considérer dans la résolution du problème, T_i doit être plus petit que t_i, ce qui mène au seuil suivant $\frac{1/s-1}{1-1/S} \times y_i$

avec x_i dépendant de y_i, s et S (inéquations 3.18).

$$\begin{aligned} T_i &< t_i \\ x_i/S + y_i/s + z_i &< x_i + y_i + z_i \\ x_i/S + y_i/s &< x_i + y_i \\ y_i.(1/s - 1) &< x_i.(1 - 1/S) \\ x_i &> \frac{1/s - 1}{1 - 1/S}.y_i \end{aligned} \tag{3.18}$$

Si le seuil est respecté, les accès mémoires dans les régions du code non-sélectionnées peuvent être ignorés lors de l'évaluation des coefficients de la fonction objectif propres à la donnée i. C'est-à-dire que la donnée i est considérée comme étant accédée uniquement dans les régions du code sélectionnées, elle fait donc partie de l'ensemble des données à considérer pour la résolution du problème. Au contraire, si le seuil n'est pas respecté, la donnée i est considérée comme étant uniquement accédée dans des régions du code non sélectionnées, et elle est retirée de cet ensemble. En considérant les notations utilisées jusque-là, cela signifie que pour toute donnée i tel que $x_i > \frac{1/s-1}{1-1/S}.y_i$, $\forall j \in [\![1; M]\!], x_{[ij]}$ ne sont plus des variables du problème.

Jusqu'ici on s'est concentré sur la sélection de donnée à partir d'une analyse temporelle. Cependant, plusieurs données peuvent être accédées dans la même région du code, et comme nous l'avons vu dans l'état de l'art, une analyse spatiale est nécessaire pour évaluer et comparer les différentes données entre elles. Intuitivement, bien qu'une région ait une importante influence globale sur le temps d'exécution du code, si une donnée n'y est accédée qu'une fois, l'allocation de celle-ci sur une mémoire différente n'aura pas d'influence sur le temps d'exécution de cette région, alors qu'une autre donnée accédée 100K fois dans la région aura nécessairement plus de poids.

Pour être capable d'allouer chaque donnée indépendamment, des poids spécifiques à chaque donnée doivent être évalués, il n'est pas suffisant d'évaluer le poids des régions du code. Nous avons vu dans l'état de l'art qu'il existe un certain nombre d'outils qui permettent de faire une analyse orientée donnée. Les analyses reposant sur ces outils ne pouvaient pas mesurer correctement la bande passante puisqu'elles ne prennent pas en compte les accès concurrents à plusieurs données au sein d'une même région, ainsi que le temps passé à faire ces accès (la bande passante étant une quantité d'accès par unité de temps). Il est néanmoins envisageable, une fois que les zones sensibles à la bande passante sont mises en évidences de donner un poids différent à chaque donnée en fonction de sa participation à la congestion du système mémoire. On peut donc réutiliser ici les méthodes de l'état de l'art, qui seront alors augmentées d'une analyse réelle de la sensibilité à la bande passante ainsi que d'une réduction du nombre de données à considérer dans la résolution du problème d'optimisation.

Dans la partie évaluation (chapitre 5, partie 5.2.2) nous considérerons que le seuil introduit dans les inéquations 3.18 est toujours vrai, ce qui se justifie avec les caractéristiques des mémoires du système étudié. Finalement, la nouvelle formulation peut s'écrire :

(3.19)

$$\begin{aligned} \min_{x_{[ij]}} \quad & \sum_{j=1}^{M}\sum_{i=1}^{N}(x_{[ij]} \times \sum_{r\in\mathcal{H}\cap\mathcal{B}} t^{r}_{[ij]}) \\ \text{sujet à} \quad & \forall j \in [\![1;M]\!], \forall l \in \mathcal{D}, \sum_{i=1}^{N} q_{i,l} \times x_{[ij]} \times m_i < \mathcal{C}_j, \\ & \forall j \in [\![1;M]\!], \sum_{i=1}^{N} x_{[ij]} = 1 \end{aligned} \tag{3.20}$$

$\mathcal{H} \subset \mathcal{R}$ est l'ensemble des régions considérées comme non négligeable dans le temps d'exécution total de l'application, obtenu par profilage. $\mathcal{B} \subset \mathcal{R}$ est l'ensemble des régions temporelles sensibles à la bande passante obtenu par profilage. Les $x_{[ij]}$ sont les inconnues, variables du système. Les coefficients $t_{[ij]}$ sont mesurés à partir du profilage (analyse temporelle puis spatiale). Les coefficients $q_{i,l}$ représentent l'appartenance des données aux groupes d'interférences afin de respecter les contraintes de capacité de chacune des mémoires du système, ils sont obtenus avec un profilage dynamique aussi. Enfin, les coefficients m_i et $\mathcal{C}_j$ représentent respectivement l'usage mémoire de la donnée i (profilage) et la capacité mémoire de la mémoire j (profilage, ou constructeur).

3.4 Discussion

Comme toute méthode basée sur un profilage de l'application, notre méthode fait intervenir l'utilisateur final. Les utilisateurs du Calcul Haute Performance sont néanmoins habitués à devoir profiler leur application pour optimiser aux mieux les performances de leur application. De plus, notons qu'il est possible d'effectuer un profilage en ligne [Wu et al., 2017a], mais cela interdit d'appliquer une approche proactive sur les données allouées hors de la boucle principale.

Chapitre 4

Implémentation de la méthode de résolution

Dans le chapitre précédent, nous avons détaillé notre méthode de sélection d'une stratégie d'allocation de données. Notre méthode nécessite la résolution d'un problème linéaire en nombres entiers, et nous proposons de déterminer les coefficients de la fonction objective, et des contraintes, par une analyse temporelle, puis spatiale, de l'application cible. Dans ce chapitre, nous commençons par détailler l'implémentation de l'analyse temporelle, et le développement de notre Bandit de bande passante, permettant une évaluation partielle de ces coefficients. Dans une deuxième partie, nous développons l'analyse spatiale complétant l'évaluation des coefficients. La troisième partie est consacrée aux détails du développement de la bibliothèque de capture des allocations pour leur redirection sur la mémoire sélectionnée au cours de l'exécution de l'application. Finalement, nous discutons nos différents choix d'implémentation en les comparant à des travaux connexes.

4.1 Implémentation de l'analyse temporelle

Dans cette partie, nous développons notre analyse temporelle. Pour cela nous commençons par passer en revue les différentes métriques utilisées dans l'état de l'art dans l'application d'une analyse orientée donnée (partie 4.1.1). Puis, nous développons notre métrique à partir d'un concept de la littérature, le Bandit de bande passante, dont nous

détaillons le fonctionnement (partie 4.1.2), et notre implémentation (partie 4.1.3). Enfin, nous développons un outil de profilage temporel à partir de cette métrique (partie 4.1.4) et nous discutons d'autres possibilités d'implémentation et des spécificités de notre outil de profilage.

4.1.1 Étude des métriques existantes

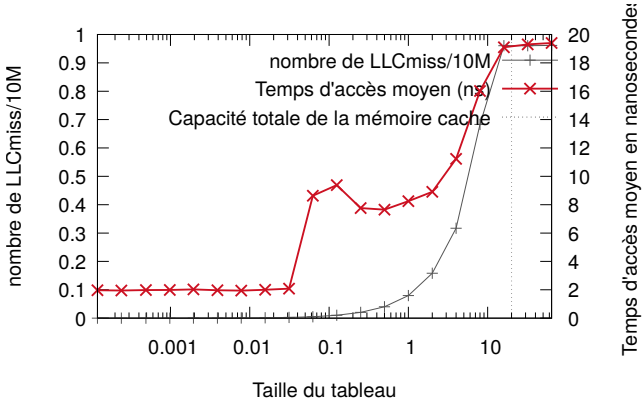
Que ce soit pour résoudre le problème d'allocation ou bien le problème de placement de données, il est nécessaire d'utiliser une ou plusieurs métriques afin d'évaluer les données du programme pour les répartir sur les différentes mémoires du système. Même lorsque le problème n'est pas formulé comme un problème d'optimisation linéaire [Peng et al., 2017], les auteurs mesurent la sensibilité d'une donnée à la latence ou à la bande passante à l'aide de métriques sélectionnées et validées sur des micro-benchmarks.

Certains auteurs ont essayé de prendre des mesures à l'aide de métriques basées sur des informations statiques obtenues à la compilation [Khaldi and Chapman, 2016], mais comme expliqué dans l'état de l'art (cf partie 2.2.2, page 31), nous avons écarté cette possibilité. En effet, la complexité des analyses d'alias dans des codes C ou C++ rend impossible la correspondance entre l'analyse des patterns d'accès mémoire et les objets de données concernés avec une analyse statique. D'autres auteurs ont étudié les patterns d'accès à la mémoire à l'aide d'un profilage dynamique précis [Peng et al., 2017] pour mesurer la sensibilité d'une application aux caractéristiques mémoires. Cette méthode nécessite une instrumentation de tous les accès en lecture et en écriture avec une analyse dynamique au surcoût important. Les auteurs attribuent un score à chaque objet, pour chaque donnée, ce qui correspond à une approximation du coefficient $t_{[ij]}$ introduit dans le chapitre 3.

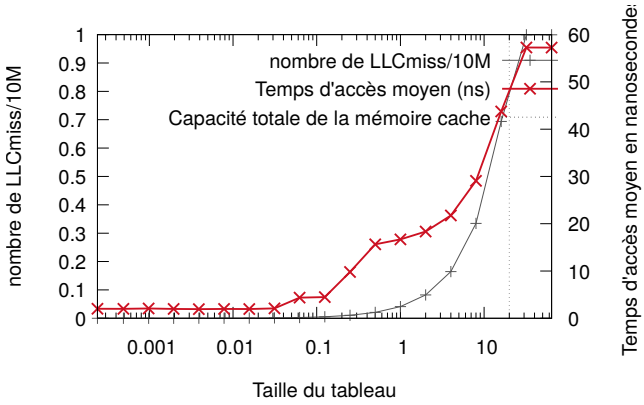
Mais la métrique qui revient le plus souvent est le compte de LLCmiss [Servat et al., 2017, Laghari and Unat, 2017, Wu et al., 2017a, Peña and Balaji, 2014b]. Cette métrique se décline sous différentes formes : le compte de LLCmiss seul [Peña and Balaji, 2014b] ou par octet pour chaque donnée [Servat et al., 2017] ou bien compte de LLCmiss corrigé avec un facteur obtenu expérimentalement [Wu et al., 2017a]. Dans cette partie nous mettons en évidence les limites de ces métriques, justifiant ainsi le choix de la métrique que nous avons fait.

Le numérateur du compte de LLCmiss par octet possède une valeur d'autant plus grande que le nombre d'accès manqués aux mémoires caches est grand, et le dénominateur induit sur le ratio une valeur d'autant plus grande que la donnée est de petite taille. Or, rien ne prête à penser que plus une donnée est petite, plus elle génère du trafic mémoire. Au contraire, cette donnée peut tenir dans le cache, et donc, si les accès à cette donnée sont suffisamment proches dans le temps, la donnée reste dans les mémoires caches, et les accès ne nécessitent pas le reste du système mémoire. Cette métrique, basée sur un ratio et fonctionnant dans le cas du problème de sac à dos classique n'est donc pas adaptée à notre problématique.

Les résultats observables sur la figure 4.1.1 mettent en évidence le fait qu'il n'y a pas de lien clair de cause à effet entre le compte de LLCmiss et le temps d'exécution d'un programme. Ces résultats ont été obtenues en exécutant sur une architecture Sandy Bridge



(a) pas de 128 octets



(b) pas de 256 octets

FIGURE 4.1.1 – Comparaison du ratio de LLCmiss avec la latence moyenne d'un accès pour 100 K accès réguliers avec des pas différents sur SandyBridge, 1 seul thread

un code dont le but est de rendre compte du temps de latence des accès selon que la donnée est dans une mémoire cache ou en mémoire principale. Il consiste à effectuer un parcours de tableau par un unique processus léger. La taille du tableau parcouru ainsi que le pas entre deux accès sont paramétrables. Ce temps de latence moyenne est directement lié à la bande passante par la formule de Little (cf partie 1.2.1, page 10). Ces figures permettent de mettre en évidence qu'un même compte de LLCmiss, selon le pattern d'accès et la taille du tableau accédé, ne correspond pas au même temps de latence moyenne. En effet, pour des ratios similaires, des tableaux accédés avec un pas de 128 ou bien de 256 octets provoquent des temps d'accès différents. Avec des accès au pas de 128 octets, non seulement la courbes de ratio n'est pas croissante, mais en plus les tailles de tableaux qui entrainent des ratios de 0.4 à 0.6 impliquent des temps d'accès entre 1 et 12 nanosecondes ce qui correspond au tiers des temps d'accès avec des pas de 256 octets. Cela signifie qu'avec un ratio plus élevé un tableau pourrait être placé sur une mémoire plus rapide par une stratégie basée sur le compte de LLCmiss alors qu'un autre ayant un ratio plus faible en aurait plus bénéficié à cause de son pattern d'accès.

Certains auteurs [Servat et al., 2017] utilisent le compte de LLCmiss comme métrique. Dans leur article, les auteurs comparent le compte de LLCmiss par objet de données pour savoir quelles données allouer sur la mémoire à forte bande passante, la MCDRAM, de l'AMH étudiée qu'est le KNL. Pour ne pas placer sur la MCDRAM des données sensibles à la latence, parce qu'elle possède une latence 10% plus lente que celle de la DDR4, les auteurs instaurent un seuil arbitraire, pour chaque objet, de 1% de LLCmiss par instruction. Finalement, le compte de LLCmiss brut ne semble pas être une métrique convaincante pour l'évaluation de la sensibilité d'une donnée aux performances mémoires.

Une autre métrique, utilisée par McCalpin pour mesurer la bande passante dans la suite de benchmarks STREAM [McCalpin, 1995], consiste à diviser la quantité de données transférées par le temps total d'exécution. Cela suppose de connaître le nombre d'accès en lecture et en écriture qui n'ont pas pu être satisfaits par le cache de dernier niveau. On peut alors mesurer "à la main" la quantité de données nécessaires par le calcul en omettant le travail du pré-lecteur matériel en simplifiant la localité temporelle et spatiale dans les caches avec l'hypothèse d'une mémoire cache de taille infinie. Mais ce travail de mesure est fastidieux, et l'hypothèse simplificatrice ne permet pas de rendre compte du trafic mémoire pour un code plus complexe qu'un benchmark.

La métrique que nous retenons pour notre outil d'analyse provient des travaux sur le concept du Bandit de bande passante [Eklov et al., 2012, Eklov et al., 2013, Casas and Bronevetsky, 2014 Casas and Bronevetsky, 2016] permettant d'obtenir le profil de sensibilité à la latence et la bande passante d'une application. En effet, le bandit de bande passante consiste à profiler l'exécution d'une application en volant de sa bande passante mémoire. Une fois ce profilage effectué il est alors possible de connaître la variation des performances de l'application en fonction de la bande passante disponible. On peut donc essayer d'extrapoler et de savoir combien pourrait gagner une application à s'exécuter sur une architecture avec une meilleure bande passante. Dans [Eklov et al., 2013], les auteurs ont aussi montré que selon le profil obtenu il est possible de différencier une application sensible à la latence d'une application sensible à la bande passante. Cela demande néanmoins une dizaine d'exécutions, parce qu'il faut exécuter le code avec un bandit dérobant des valeurs variées de bande passante. Enfin, la méthode implémentant cet outil n'est pas publiquement disponible et n'a pas été pensé pour un profilage temporel à grain fin. Nous décidons d'implémenter notre Bandit de bande passante, en réduisant le nombre d'exécution nécessaire à son minimum, c'est-à-dire deux, une avec et une sans le Bandit activé.

Enfin, le temps d'exécution d'une région, voire d'une application, est une métrique courante dans le HPC. Lorsqu'il est mesuré relativement pour chaque région de code de l'application, il permet de mettre en évidence les points chauds du code, ceux dans lequel l'exécution passe le plus de temps. Pour optimiser un code on cherche alors à améliorer les performances des parties chaudes du code avec de la parallélisation, vectorisation, réutilisation des données dans le cache, etc. Pour placer les données sur une mémoire possédant de meilleures caractéristiques, il semble donc intéressant de chercher à allouer sur cette mémoire les données accédées dans les régions chaudes du code, puisque dans les autres régions, allouer une donnée sensible à la bande passante sur une mémoire à forte bande passante n'aurait que peu d'influence sur le temps d'exécution total de l'application.

Nous cherchons donc à coupler cette métrique temporelle avec l'utilisation d'un Bandit de bande passante pour développer notre analyse temporelle.

4.1.2 Principe du Bandit de bande passante

Notre méthode de résolution nécessite l'évaluation des coefficients de la fonction objective présentée dans la partie 3.1.2, page 46. L'implémentation de notre méthode prend pour cible l'architecture du KNL qui est une AMH composée d'une DDR4 et d'une MCDRAM (architecture détaillée chapitre 5, page 76). Les caractéristiques différenciant le plus ces deux mémoires sont la bande passante et la capacité mémoire, d'où l'importance de l'analyse temporelle décrite dans le chapitre 3, différenciant les régions de code sensibles à la bande passante mémoire.

Au sein de l'analyse temporelle, nous implémentons un Bandit de bande passante qui évalue relativement les coefficients en approximant la dérivée du temps d'exécution en fonction de la bande passante disponible (cf équation 4.1). Le profilage avec notre Bandit se fait sur un système de mémoires homogènes (NUMA), et la variation de temps d'exécution de l'application sur un tel système dépend uniquement de la variation de bande passante induite par l'exécution du Bandit.

$$Bandit\_Coeff = \frac{\Delta t}{\Delta bande\_passante} \tag{4.1}$$

Cependant, la bande passante n'est pas la seule caractéristique mémoire différenciant les performances des applications exécutées sur KNL, la latence doit aussi être considérée, parce que la MCDRAM possède 4 à 5 fois plus de bande passante que la DDR4, mais 10% de latence supplémentaire. Comme cela est expliqué page 16, avec la figure 1.2.4, selon l'usage de la bande passante mémoire fait par les accès mémoires de l'application, l'allocation des données devra être fait sur l'une ou l'autre des deux mémoires.

On peut estimer la variation entre deux coefficients de la fonction objective à l'aide du Bandit : dans l'équation 4.2, l'indice 0 de t_{i0} correspond à la DDR SDRAM et l'indice 1 de t_{i1} à la MCDRAM du KNL.

$$\forall i \in Data,$$
$$t_{i0} - t_{i1} = \begin{cases} Bandit\_Coeff \times \Delta bande\_passante, \text{ Si } Bandit\_Coeff > Seuil \\ \Delta latence \times Nombre\_Acces\_memoire, \text{ Si } Bandit\_Coeff < Seuil \end{cases}. \tag{4.2}$$

Cette estimation est justifiée par le fait qu'une application sensible à la latence fait peu d'usage de la bande passante disponible, donc lorsque le Bandit influence son exécution, en déplaçant l'usage mémoire total vers la droite sur la courbe 4.1.2, on doit quand même observer une perte de performance, et un temps d'exécution qui s'allonge. Néanmoins, sur une application sensible à la bande passante, on peut passer de la zone linéaire à la zone exponentielle, ou bien être déjà dans la zone exponentielle, dans les deux cas le ralentissement observé devrait être plus important que sur un code sensible à la latence.

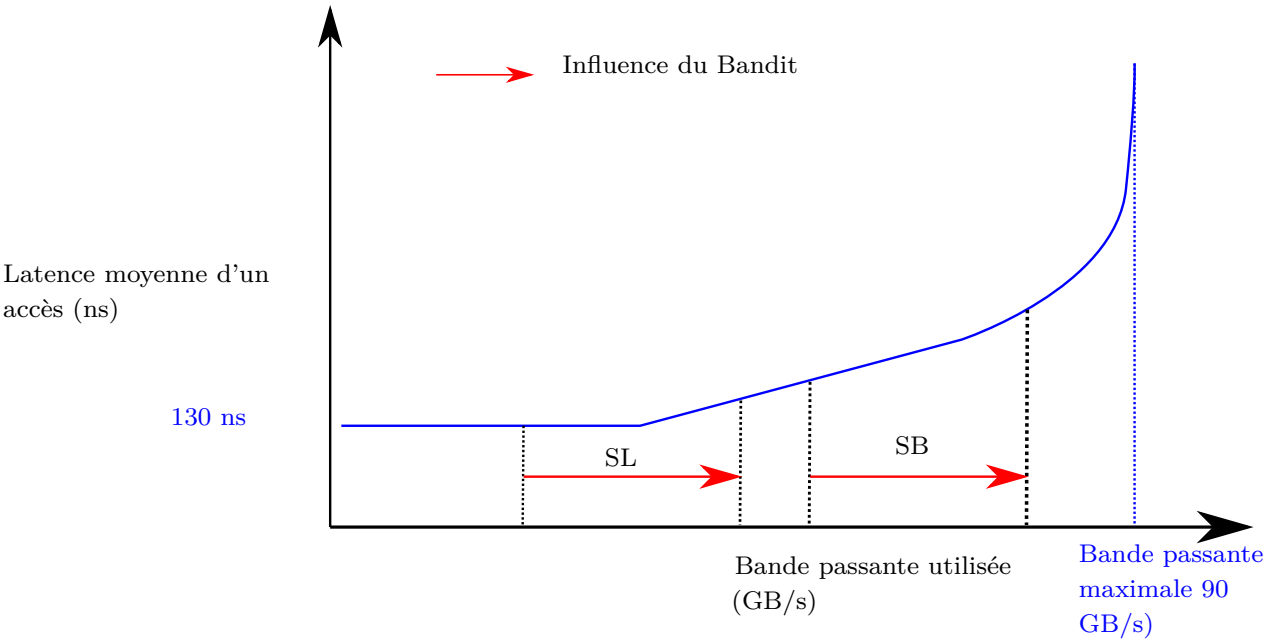


FIGURE 4.1.2 – Latence d'un accès mémoire en fonction de la bande passante mémoire utilisée par l'application, observation de l'influence du Bandit sur l'usage de bande passante et les conséquences sur la latence moyenne. SL : sensible à la latence, SB : sensible à la bande passante

4.1.3 Implémentation du Bandit de bande passante

Détails techniques

Le but d'un profilage au Bandit, respectivement de mémoire principale ou de cache, est d'évaluer l'influence d'une réduction des ressources disponibles, respectivement de bande passante mémoire ou de capacité du cache, sur les performances de l'application profilée (cf travaux d'Eklov *et al.* [Eklov et al., 2012, Eklov et al., 2013] et Casas *et al.* [Casas and Bronevetsky, 2016]). Ainsi, il est possible de prédire la perte de performance engendrée par le partage de ressource matérielle due à l'exécution simultanée de plusieurs applications sur un même nœud de calcul. Nous détournons le but du Bandit de mémoire principale, nommé Bandit de bande passante dans la littérature, pour en faire un outil de prédiction des performances d'une région temporelle pour son exécution sur une architecture possédant une meilleure bande passante.

Le Bandit est composé de processus léger exécutant un parcours de tableau pour générer des accès mémoires venant congestionner les files d'attente du contrôleur mémoire. Pour ne pas rendre notre Bandit tributaire du fonctionnement du placement de données dans les différents niveaux de mémoires caches, comme c'est le cas des autres Bandits, notre implémentation permet d'attacher ces processus légers sur un (ou plusieurs) nœud NUMA sélectionné. On fait l'hypothèse qu'en prenant soin d'exécuter l'application profilée sur un (ou plusieurs) nœud NUMA différent de ceux du Bandit, les accès aux mémoires cache du

Bandit ne viendront pas influencer les accès aux mémoires cache de l'application. Cette hypothèse sera validée dans la partie suivante (cf page 62). La bibliothèque dynamique de notre Bandit permet aussi de sélectionner les nœuds NUMA sur lesquels sont placées les données auxquelles il accède. Ainsi en prenant soin de les placer sur la mémoire locale aux processus légers de l'application ciblée, les accès mémoires effectués par le Bandit entraînent une congestion du contrôleur mémoire qui gère aussi les accès mémoire de l'application. Du point de vue de l'application cible, tout se passe comme si la bande passante disponible était artificiellement diminuée. Enfin, le fait que notre Bandit repose sur l'utilisation d'une machine NUMA rend son application en partie indépendante de la machine sur laquelle il est exécuté, au contraire de l'implémentation d'Eklov, qui nécessite de prendre en compte la taille des mémoires caches et leur degré d'associativité. Dans la partie suivante, la validation du Bandit a été menée sur 3 machines NUMA différentes : Haswell, Nehalem et Sandy Bridge.

Dans notre implémentation la bande passante volée par le Bandit est limitée par les effets NUMA. Plus précisément, elle est limitée par la bande passante QPI reliant les nœuds NUMA entre eux. Cette limitation est en partie due au fait qu'il y a moins de connexions à l'agent QPI sur le bus (anneau partagé), qu'au contrôleur mémoire [Saini et al., 2014]. La congestion occasionnée par les accès via le QPI impacte bien le débit du traitement de toutes les requêtes mémoires (anneau partagé), c'est-à-dire tant la bande passante mémoire du Bandit en accès distant que celle de l'application en accès local.

Notre bibliothèque dynamique implémentant le Bandit possède 4 méthodes à insérer dans l'application à profiler. Une première méthode permet d'initialiser les processus légers et de les attacher aux nœuds NUMA sélectionnés à l'aide d'appel à la bibliothèque hwloc [Broquedis et al., 2010]. Une deuxième méthode, devant être appelée à la fin du *main*, permet de terminer le Bandit et de récupérer toutes les informations de profilage alors écrites dans un fichier de données. Au-delà de ces deux fonctions, deux autres méthodes permettent de démarrer et d'arrêter les processus légers du Bandit pour encadrer au mieux la région (temporelle) du code à laquelle on souhaite voler de la bande passante.

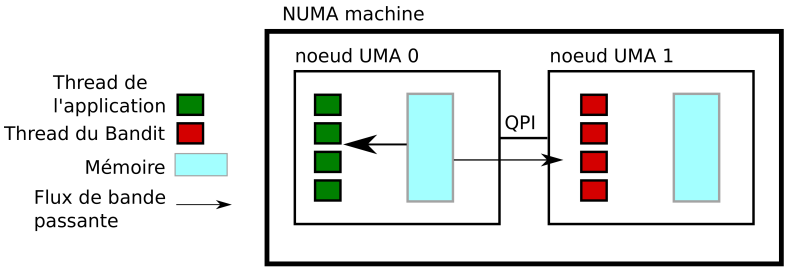


FIGURE 4.1.3 – Schéma de fonctionnement de notre Bandit de bande passante

Les processus légers de notre Bandit de bande passante sont directement lancés par le processus de l'application analysée, et l'attachement des processus légers du Bandit aux nœuds NUMA souhaités se fait avec les variables d'environnement définis dans la bibliothèque : nombre de nœuds NUMA et leur numéro logique dans la représentation hwloc. On peut donc choisir précisément sur quels nœuds NUMA le Bandit s'exécute, mais

la bibliothèque du Bandit ne permet pas d'assurer le placement des processus légers de l'application. Le fonctionnement du Bandit est illustré sur la figure 4.1.3.

Dans nos expériences avec des applications utilisant uniquement le paradigme de mémoire partagée (standard OpenMP), nous avons utilisé la variable d'environnement KMP_AFFINITY pour la version Intel, et OMP_PLACES couplé avec CPU_PROC_BIND pour la version GCC, afin d'attacher les processus légers de l'application sur un nœud NUMA particulier. Pour que les processus légers de l'application et ceux du Bandit soient attachés à des nœuds NUMA différents il faut tenir compte du fait que la numérotation utilisée par les implémentations OpenMP sont basées sur la numérotation physique de l'architecture (celle donnée par le système d'exploitation), alors que l'attachement des processus légers du Bandit utilise la numérotation logique de hwloc. Cette dernière bibliothèque permet aussi de faire la correspondance entre les deux notations.

En ce qui concerne les codes utilisant MPI, il y a deux cas de figure. Soit il n'y a qu'un processus MPI par nœud et alors on se retrouve dans le cas précédent. Soit il y a plusieurs processus MPI par nœud et alors il faut que le nombre de processus multiplié par le nombre de processus légers par processus MPI ne dépasse pas la capacité d'un nœud NUMA. Ceux-ci sont alors attachés aux même nœud NUMA, et le Bandit peut se positionner sur un autre nœud NUMA. Mais l'implémentation décrite précédemment lance le Bandit pour chaque processus, ce qui pose problème ici puisqu'on veut un seul Bandit par nœud NUMA. On a donc ajouté la possibilité de préciser les rangs des processus lançant des processus légers de Bandit. En supposant que le placement est régulier, il suffit de passer un nombre et tous les processus dont le rang est égal à zéro modulo ce nombre lancent le Bandit.

La bande passante utilisée par les processus légers de notre Bandit de bande passante est mesurée de la même manière qu'elle est mesurée par McCalpin dans la suite de benchmarks STREAM [McCalpin, 1995]. C'est-à-dire que la quantité de données traversées par le Bandit est divisée par le temps passé à s'exécuter.

Validation de notre Bandit

Dans cette partie nous validons notre Bandit en mettant en évidence l'absence d'influence de celui-ci sur les accès mémoires de l'application aux différents niveaux de mémoire cache. De plus, nous développons le calibrage de notre Bandit afin qu'il soit capable de différencier une région de code selon qu'elle est sensible à la bande passante mémoire, à la latence ou bien au calcul.

Pour valider l'hypothèse de non interférence du Bandit sur les accès aux caches, nous appliquons notre Bandit à une application de produit matriciel, DGEMM (un GEMM sur des nombres de type `double`). Cette application est optimisée pour tenir en cache (produit par bloc) et pour réutiliser le plus possible les données [Kågström et al., 1998]. Elle n'utilise donc que peu la bande passante de la mémoire principale, et si on observe une influence du Bandit, c'est que celui-ci n'a pas le comportement souhaité et détériore les performances des mémoires cache du nœud NUMA de l'application. Le fait d'observer une influence négligeable du Bandit sur les performances de l'application DGEMM validerait donc l'hypothèse.

Nous avons testé deux configurations pour notre Bandit de bande passante. Nous montrons que la première configuration ne permet pas de valider les critères de fonctionnement attendu du Bandit, alors que la deuxième configuration sera retenue pour développer notre outil.

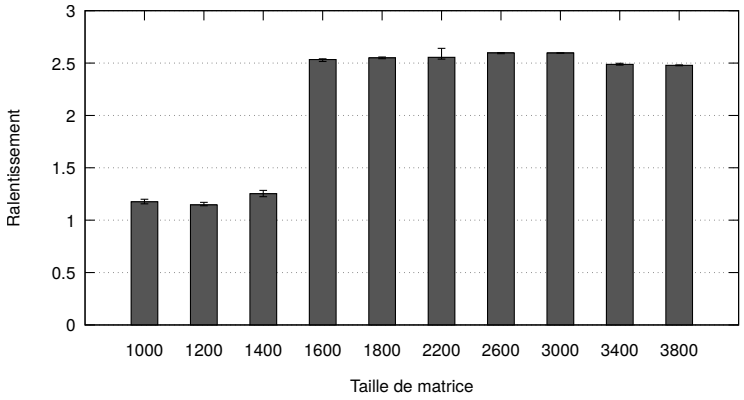


FIGURE 4.1.4 – DGEMM volée : données distantes

Le graphe de la figure 4.1.4 représente l'influence de notre Bandit de bande passante sur DGEMM, sur Haswell, avec les données de l'application allouées en distant, c'est-à-dire qu'elles sont situées sur même nœud NUMA que les processus légers du Bandit. On peut observer une influence d'un facteur $\times 2.5$ du Bandit sur DGEMM à partir d'une certaine taille de matrice qui correspond à un usage mémoire qui dépasse la taille des caches privés. Cette influence, intervenant dès que l'usage mémoire implique une utilisation intensive du cache partagé, vient invalider l'utilisation du Bandit pour voler uniquement de la bande passante mémoire lorsque les données de l'application sont allouées de manière distante sur la mémoire du nœud NUMA sur lequel les processus légers du Bandit s'exécutent.

La figure 4.1.5 montre les résultats obtenus lorsqu'on exécute l'application DGEMM en cherchant à voler sa bande passante disponible avec notre Bandit sur un processeur Sandy Bridge. On peut voir que pour des matrices de tailles suffisantes, il n'y a presque pas d'influence du Bandit sur les performances de l'application. Cela signifie donc que cette application, comme on pouvait s'y attendre par sa nature, n'est pas sensible à la bande passante disponible sur la machine. Les expériences de validation ont aussi été menées sur Nehalem et montre aussi une influence négligeable du Bandit sur DGEMM.

On peut observer l'influence du Bandit sur un benchmark de Pointer Chasing avec la figure 4.1.7. Le processus de ce benchmark, issue de la suite lmbench [McVoy and Staelin, 1996], effectue des accès mémoire espacés d'un certains pas, dont l'adresse suivante est contenue dans la case mémoire de l'accès précédent, mais régulier. Ainsi pour des pas suffisamment réduits, le système de pré-lecture matérielle est efficace, les performances des accès mémoires du benchmark deviennent plutôt sensibles à la bande passante mémoire et c'est pourquoi on peut observer une sensibilité au Bandit supérieure à 15%. Ce seuil est mesuré à 10% sur la machine à deux sockets Nehalem sur laquelle nous avons aussi validé et calibré

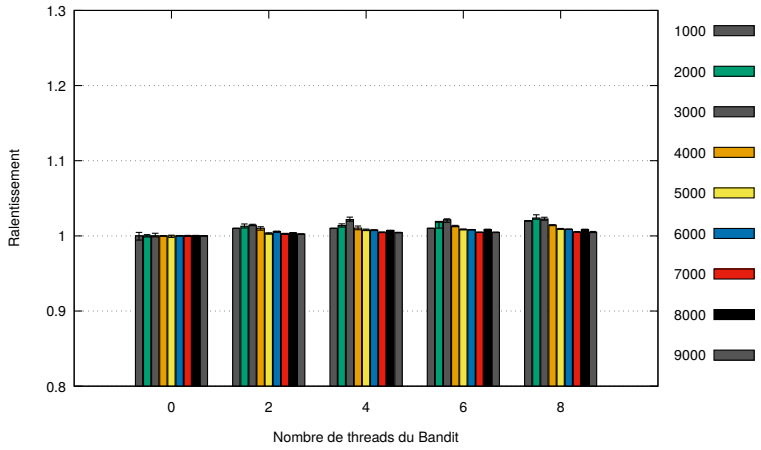


FIGURE 4.1.5 – Ralentissement induit par notre Bandit sur l'application DGEMM sur Sandy Bridge

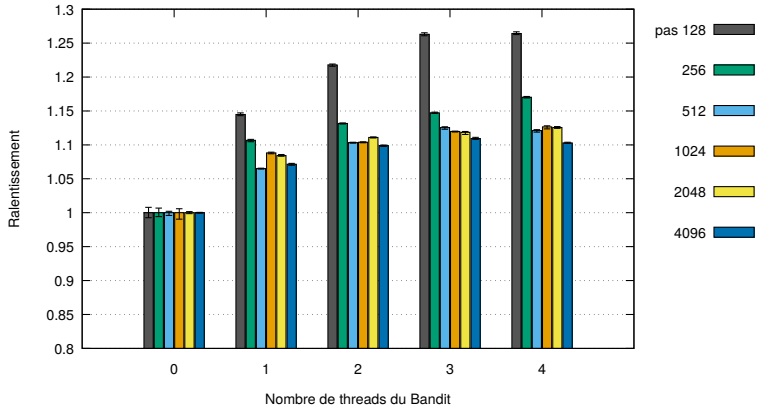


FIGURE 4.1.6 – Tableau de taille de 30 MB

FIGURE 4.1.7 – Influence du Bandit une application de Pointer Chasing pour une taille supérieure à la capacité de la mémoire cache (20MB) et différentes tailles de pas

notre Bandit. Dans la suite du manuscrit, on nommera sensibilité au Bandit de bande passante (SBB), l'impact du Bandit sur une région de code, c'est donc le ralentissement induit par l'exécution du Bandit. Si SBB dépasse le seuil de sensibilité à la bande passante calibré (15% ici), la région est considérée comme sensible aux performances de bande passante mémoire.

La figure 4.1.8 montre l'influence du Bandit sur le benchmark STREAM sur Haswell, les données de STREAM sont locales, puisque le cas avec les données distantes a été invalidé. En ordonnée on lit la quantité de bande passante en GB/s mesurée sur l'application STREAM en fonction de la bande passante (en GB/s) prise par le Bandit de bande passante. La courbe en pointillés qui relie les points suit l'ordre croissant du nombre

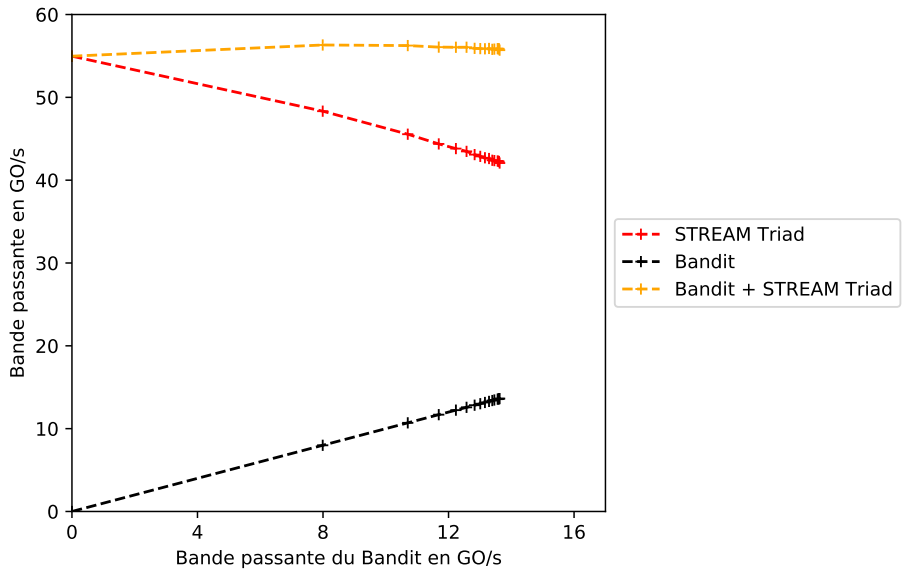
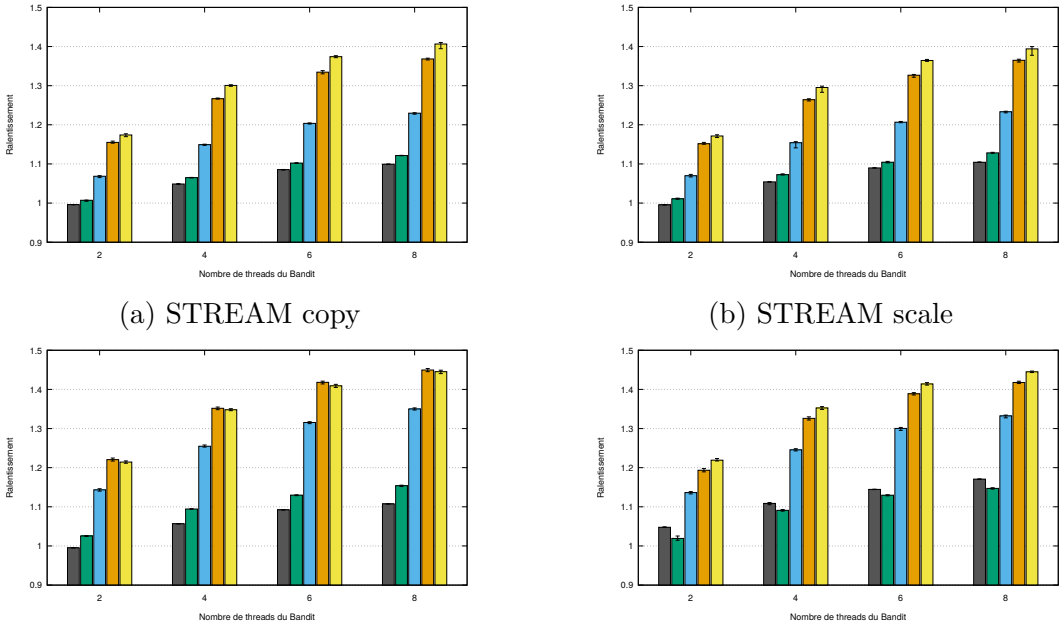


FIGURE 4.1.8 – Performance STREAM volé, exécution sur Haswell (4 nœuds NUMA)

de processus légers du bandit s'exécutant. Alors que la somme des bandes passantes cumulées du Bandit et de l'application reste relativement constante, on peut observer que l'augmentation de la bande passante prise par le Bandit correspond à une baisse de la bande passante mesurée sur STREAM. À mesure qu'on augmente le nombre de processus légers exécutant le code du Bandit de bande passante sur le nœud NUMA distant, la fréquence des accès mémoire du Bandit augmente, le contrôleur mémoire du nœud NUMA sur lequel s'exécute les processus légers de STREAM doit donc gérer des requêtes supplémentaires. L'équilibre entre la bande passante prise par le Bandit, limitée par la file de requêtes provenant du QPI, et celle prise par STREAM (requêtes locales) correspond à un ratio d'environ $\frac{1}{4}/\frac{3}{4}$. Ce ratio est proche de 0.2, le ratio de la taille de la file QPI sur la taille totale des files au sein du contrôleur mémoire [Thomadakis, 2011], en admettant que les architectures Nehalem et Sandy soient similaires sur ce point. On observe une légère augmentation de la bande passante totale, correspondant au fait que le contrôleur mémoire n'est pas au maximum de sa capacité de traitement parallèle des requêtes mémoires lorsque la file de requête provenant du QPI n'est pas utilisée.

La figure 4.1.9 permet d'observer le ralentissement induit sur chacun des 4 benchmarks STREAM (copy, scale, add et triad) par notre Bandit de bande passante en fonction du nombre de processus légers de Bandit avec une exécution sur Sandy Bridge. Le ralentissement maximal est à chaque fois atteint avec le nombre maximal de processus légers (8) qu'il est possible d'exécuter simultanément sur un nœud NUMA de cette architecture. C'est donc cette configuration qu'on utilise lors du profilage des mini-applications dans le chapitre 5.

Finalement, notre Bandit, parce qu'il est attaché aux cœurs de calcul du nœud NUMA distant, n'a aucune influence sur le cache partagé utilisé par l'application, car il ne l'utilise pas. En cela il diffère des Bandits existants (Eklov *et al.* et Casas *et al.*) dont les processus



(a) STREAM copy

(b) STREAM scale

(c) STREAM add

(d) STREAM triad

FIGURE 4.1.9 – Influence du Bandit sur les 4 versions de STREAM

légers s'exécutent sur des cœurs appartenant au même nœud NUMA.

Notre Bandit a un potentiel d'impact inférieur à celui d'Eklov *et al.* à cause des limitations de bande passante dues aux accès NUMA, mais il est plus simple à mettre en œuvre et peut s'adapter facilement à de nouvelles architectures NUMA, alors que leur Bandit, parce qu'il est spécifiquement adapté pour que moins de 4% de la totalité du cache partagé soit impactée, est lié à l'architecture matériel du Nehalem. Comparé à celui de Casas *et al.*, l'impact de notre Bandit est équivalente, car ces derniers limitent le nombre de processus légers exécutés par leur Bandit, pour que l'influence sur les mémoires caches soit "acceptable", c'est-à-dire qu'elle ne dépasse pas un seuil qu'ils fixent arbitrairement.

4.1.4 D'une métrique vers un outil de profilage temporelle

À partir du Bandit nous développons un outil de profilage temporel qui attribue à chaque région du code étudié une valeur représentant sa sensibilité au caractéristique mémoire de bande passante, en plus de la valeur représentant son influence globale. Pour cela il est nécessaire de mesurer le temps passé au sein de chaque région et de pouvoir comparer ce temps au temps total d'exécution de l'application. Se pose alors certaines questions : quelle granularité pour les régions à étudier ? Comment mesurer précisément le temps d'exécution sans interférer avec l'exécution elle-même ?

La plupart des outils de profilage possèdent leur propre façon de mesurer le temps d'exécution d'une région, et son influence sur le temps d'exécution total de l'application. Certains outils pratiquent l'échantillonnage sur un appel système, comme gprof. D'autres établissent leur mesure à partir de compteur matériel, toujours par échantillonnage :

VTune, Vampir, Tau, OProfile. Cependant Vampir et VTune sont propriétaires, il n'est pas possible d'accéder à la trace pour automatiser l'étude.

L'utilisation de l'outil Oprofile ne permet pas de connaître le temps passé dans les boucles parallèles OpenMP. L'utilisation de profilage avec Tau est tout à fait envisageable et même souhaitable pour une extension plus robuste de notre outil. Notre outil repose sur un plugin GCC permettant d'instrumenter chaque appel de fonction et chaque boucle parallèle OpenMP, parce que nous avons trouvé expérimentalement qu'elles représentent une granularité nécessaire pour l'extraction d'information pertinentes sur la sensibilité à la bande passante mémoire. Notre plugin permet d'insérer des fonctions mesurant le temps d'exécution dans chaque fragment ainsi instrumenté. On récupère aussi le nombre d'appels à chaque fonction. Le plugin est aussi utilisé pour obtenir le graphe d'appel de fonction du programme. À partir de toutes ces informations et de deux exécutions de l'application cible (avec et sans Bandit) on peut calculer l'influence du Bandit sur chaque fragment de code. On en déduit ainsi un profilage temporel de la sensibilité de l'application à la bande passante. Remarquons qu'une partie de ce plugin pourrait être remplacée par l'utilisation de l'option `-finstrument-functions` de GCC, et l'utilisation de l'interface `cyg_profile_func_enter/exit` permettant d'instrumenter chaque appel de fonction d'un programme, mais pas les boucles parallèles OpenMP.

4.1.5 Discussion

Étant donné que la plupart des codes sont écrits en MPI et OpenMP, il semble pertinent d'utiliser les interfaces de profilage de ces Application Programming Interface (API). PMPI[Karrels and Lusk, 1994, Vetter and Chambreau, 2005] pour MPI, et OMPT [Eichenberger et al., 2013] pour les régions OpenMP. Puisque la granularité qui nous intéresse se situe au niveau des processus MPI ou des processus légers OpenMP, et des boucles parallèles, une telle instrumentation permettrait de capturer toutes les informations nécessaires avec moins de développement logiciel et sans la nécessité surtout de recompiler le code source.

Pour des tailles de problèmes tenant en MCDRAM la méthode d'analyse temporelle décrite dans le chapitre 3 peut-être directement appliquée sur le KNL. Pour des tailles de problèmes supérieures, la méthode du Bandit peut en théorie s'appliquer à l'architecture du KNL, et même si notre implémentation ne le permet pas parce que les nœuds NUMA du KNL partagent le cache L2, l'implémentation d'Eklov *et al.* peut en théorie s'adapter à cette architecture.

Finalement, il faut retenir que le profilage à l'aide du Bandit de bande passante ne nécessite pas la machine AMH sur laquelle on veut réellement exécuter l'application. On peut donc prévoir en partie les performances d'applications sur de futures architectures, toute proportion gardée puisque d'une architecture à l'autre bien d'autre paramètres que la bande passante mémoire sont modifiés.

4.2 Implémentation de l'analyse spatiale

À partir de l'attribution de deux valeurs à chaque région du code, son influence globale et sa sensibilité au Bandit de bande passante, il est possible d'évaluer les coefficients de la fonction objective pour chaque groupe d'objets de données de l'application (cf partie 3.2.2, page 49) accédés dans une même région temporelle. Pour cela on établit une correspondance entre les données de l'application et les régions du code, sachant qu'une donnée appartient à un fragment de code si elle est accédée au cours de l'exécution de celui-ci.

4.2.1 Objets de données

Pour établir la correspondance entre les accès mémoires et les objets de données, il faut d'abord être en mesure d'identifier ces derniers. L'adresse virtuelle du premier élément d'un objet de données (tableau à une ou plusieurs dimensions, tableau de structure, structure de tableaux, etc.) permet de le définir précisément. Le problème de cette identification, c'est qu'à chaque nouvelle exécution du programme, l'adresse virtuelle d'un objet de données alloué à la même ligne de code, à partir du même chemin d'exécution pourra quand même être différente. Afin de permettre la réutilisation de l'identité d'un objet de données pour pouvoir apporter des modifications au code, et comprendre son implication sur les performances de l'application, il faut une identification plus pratique pour le programmeur.

L'identification proposée par une des premières analyses dynamiques orientées données [Martonosi et al., 1992, Martonosi et al., 1995] (cf chapitre 2, partie 2.2.3, page 32) consiste à identifier une donnée à partir de la pile d'appels de fonctions complète menant à son allocation dynamique. Ne gérant pas les allocations statiques (pile ou globale) les auteurs proposent aux programmeurs d'instrumenter les données de ce type qu'ils souhaitent analyser. C'est aussi la méthode utilisée dans [Servat et al., 2017], et la méthode que nous utilisons.

Pour ce qui est de nommer l'objet de données, les auteurs décident de concaténer les noms des fonctions de la pile d'appels avec le compteur du programme pour localiser l'appel, ainsi que le nom de la variable (et son type) dans laquelle l'objet a été alloué. Cette dernière information nécessite une instrumentation du code. Contrairement à ces auteurs, nous n'instrumentons pas le code pour cette étape, donc les informations dont nous disposons sont uniquement la pile d'appels et l'adresse virtuelle, cela suffit à identifier un objet même si c'est une information moins facile à interpréter pour le programmeur. Il serait néanmoins possible d'ajouter le nom de la variable à ces informations en instrumentant automatiquement le code avec une passe de compilation.

4.2.2 Extraction

À partir du fichier de configuration généré lors de l'analyse temporelle, le code de l'application est compilé pour instrumenter, dans les fonctions sélectionnées, tous les accès en lecture et en écriture à des tableaux alloués dynamiquement. Le plugin GCC lit le fichier de configuration et compare le nom des fonctions analysées avec le nom des fonctions dans la liste du fichier. S'il trouve le nom de la fonction il effectue sa passe GIMPLE consistant à instrumenter les accès mémoires, sinon il ne fait rien. L'instrumentation d'un

accès consiste en l'ajout d'un appel à une méthode prenant en argument l'adresse accédée, le numéro de ligne, le nom du fichier de code et un booléen pour préciser la nature de l'accès (lecture ou écriture). En instrumentant uniquement les accès effectués dans les régions sélectionnées lors de l'analyse temporelle, on **extrait** uniquement les données de l'application responsables de la sensibilité de l'application aux performances de bande passante mémoire (cf analyse temporelle détaillée partie 3.2.3, page 49). C'est ce qu'on nomme **l'extraction naïve**, parce qu'il n'y a pas de différenciation entre les objets de données accédés dans une même région temporelle.

```
1 DynAllocId:84;DynAllocAddr:0x2b824df1d010;CodeLoc:40ff32,41118b,41122f
    ,411289,412428,40163f,40177d;InterferenceSets:7;Marked:1;MemUsage
    :8000000;Load:1000000;Store:0;
```

Listing 4.1 – Trace d'objets de données

Le plugin a été écrit pour la version 6.3.0 de GCC. Il est en parti inspiré de la passe de `ThreadSanitizer` dans laquelle tous les accès en lecture et en écriture à des tableaux alloués dynamiquement sont instrumentés afin de vérifier que les accès ne provoquent pas d'erreur de segmentation. Avant chaque accès mémoire, le code est instrumenté afin d'ajouter un appel à une méthode de notre bibliothèque dynamique. Cette méthode prend en argument l'adresse virtuelle accédée, ainsi qu'un booléen pour définir le caractère de l'accès (lecture ou écriture).

Notre bibliothèque dynamique implémente la méthode ajoutée lors de l'instrumentation ainsi que les captures des allocations dynamiques de la libc (malloc, calloc et realloc). Lors de la capture de l'allocation, la taille du tableau ainsi que sa pile d'appels, et son groupe d'interférence sont récupérés. Ces groupes d'interférence correspondent à la construction de l'ensemble $\mathcal{D}$ dans la partie 3.1.2, page 43. Ces groupes portent le nom d'`InterferenceSets` dans le listing 4.1, celui-ci est décrit dans le paragraphe suivant. Lors de l'exécution du code, pour chaque accès instrumenté, l'adresse virtuelle est comparée aux plages d'adresses des différents tableaux dont les allocations dynamiques ont été capturées. Une structure de données est ainsi tenue à jour afin de savoir quel tableau a été accédé et combien de fois.

Toutes les informations sur les allocations accédées dans les régions du code sélectionnées sont écrites dans un fichier de trace. Il contient un numéro identifiant l'allocation dynamique, son adresse virtuelle, les adresses dans le code source des différents appels de sa pile d'appels, extraits avec la fonction `backtrace_symbols` de binutils, un booléen qui est à vrai si l'allocation est accédée dans au moins une des régions instrumentées, et un compte du nombre d'accès en lecture et en écriture. Les adresses dans le code source peuvent être converties en nom de fichier, nom de fonction et numéro de ligne si l'application a été compilée avec l'option debug (généralement -g) grâce aux informations contenues suivant le standard DWARF [Committee,].

Les accès sont récupérés dans des structures privées à chaque processus léger, et il est possible de choisir le degré d'échantillonnage. Le profilage n'entraîne donc pas de verrou pendant l'exécution, cependant il faut quand même compter sur un ralentissement d'un facteur 20 à 30. Enfin, un verrou est nécessaire autour de l'appel aux fonctions des binutils `backtrace` et `backtrace_symbols` car elles font un appel à malloc qu'on ne souhaite pas capturer.

Une fois toutes ces informations récupérées, la prochaine étape consiste à effectivement évaluer les coefficients de la fonction objective détaillée dans le chapitre 3 La résolution du problème d'optimisation est implémenté avec des scripts python. L'heuristique codée consiste à trier les objets en fonction de leur valeur et de sélectionner, à partir de ceux qui ont la plus haute valeur, les objets à mettre en mémoire rapide, en respectant la contrainte de capacité mémoire. Cette partie est modulaire, et il est donc possible de coder une heuristique différente de celle actuelle pour résoudre le problème.

4.3 Développement d'un outil d'allocation automatique

Une fois le problème d'optimisation résolu, et les objets de données à placer en MCDRAM identifiés, il faut encore allouer ceux-ci sur la bonne mémoire lors de l'exécution. Les informations contenues dans les traces de profilage permettent d'identifier la localité dans le code de chacune des fonctions de la pile d'appels de l'allocation de chacun des objets à allouer en MCDRAM. Pour cela il faut identifier l'appel de cette pile identifiant au mieux l'allocation sélectionnée, c'est-à-dire l'appel contenu dans le moins d'autres pile d'appels possible, et on encadre cet appel avec les méthodes de sélection de notre bibliothèque dynamique *switchtender*. Cette bibliothèque dynamique capture les fonctions d'allocations et redirige celles identifiées à l'étape précédente. À l'exécution, notre bibliothèque fait appel à l'allocateur spécifique à la mémoire sélectionnée : l'allocateur de la bibliothèque memkind pour les données à allouer sur MCDRAM, ou bien l'allocateur de la libc pour les données à allouer sur DDR4. Pour savoir quel allocateur utiliser, la bibliothèque tient compte de la valeur d'un booléen mis à jour par les méthodes de sélection.

Le code représenté dans le listing 4.2 met en jeu une fonction, `my_malloc`, englobant l'allocateur de la libc. Cette fonction est appelée dans deux fonctions `f` et `g`, elles-mêmes appelées dans le `main`.

Lors de l'exécution de ce programme, toutes les données allouées dynamiquement sont identifiables par leur adresse virtuelle retournée par la fonction `malloc`, mais cette adresse n'est pas identique d'une exécution à l'autre. Pour l'allocation exécutée par `f`, la pile d'appels (`main`, `f`, `my_malloc` et `malloc`) suffit à identifier la donnée A. La pile d'appels des allocations de B ne suffisent pas à différentier les différents B entre eux, mais on peut agglomérer ces objets en un seul objet en supposant qu'ils auront le même comportement [Martonosi et al., 1992]. Ainsi l'ensemble des B alloués par la pile d'appels (`main`, `g`, `my_malloc` et `malloc`) constitue un seul et même objet de données analysé.

```
1  void * my_malloc(size_t size){
2      return malloc(size);
3  }
4  int f(){
5      A = my_malloc(18);
6      [... A memory accesses]
7      free(A);
8      return 0;
9  }
10 int g(){
11     B = my_malloc(42);
12     [... B memory accesses]
13     free(B);
14     return 0;
15 }
16 int main(int ac, char * av[]){
17     f();
18     for(int i=0;i<3;i++){ g() ;}
19     return 0;
20 }
```

Listing 4.2 – Pile d'appels et identification d'objet de données

Bien que sa pile d'appels permette d'identifier précisément un objet de données, la comparaison de chacun des appels à `malloc/calloc/realloc` avec les piles d'appels des objets de données à placer spécifiquement lors de l'exécution coûte cher en temps d'exécution. Nous avons choisi de remplacer ce coût par une précision plus grossière dans l'identification des données sélectionnées en instrumentant avant l'exécution un unique appel dans la pile d'appels qui définit l'allocation dynamique sélectionnée. Dans l'exemple présenté, si `f` est sensible à la bande passante et qu'il est donc intéressant de stocker A en MCDRAM, on préfèrera instrumenter le code de l'appel `my_malloc` à la ligne 6 pour qu'à l'exécution un simple booléen définisse si une donnée doit être placée en MCDRAM.

Cette instrumentation du code avant l'exécution évite un parcours de base de données à chaque nouvelle allocation qui est la solution retenue dans ce papier [Servat et al., 2017]. Cependant, notre sélection est moins précise puisqu'une ligne de code n'identifie pas de manière aussi précise qu'une pile d'appels complète. On réduit néanmoins cette imprécision en sélectionnant l'appel de la pile identifiant au mieux la donnée, c'est-à-dire l'appel se retrouvant dans le moins de piles d'appels différentes. En pratique, cette méthode doit réduire le surcoût à l'exécution, et l'instrumentation peut-être automatisée avec une passe de compilation.

Pour que les données allouées avec l'allocateur de la bibliothèque memkind soit placée en MCDRAM on met la variable environnement MEMKIND_HBW_NODES à 1 lors de l'exécution, parce que la MCDRAM correspond au nœud NUMA 1 du KNL. Afin de généraliser cette pratique à un système contenant plusieurs mémoires, il serait nécessaire d'avoir une bibliothèque pour laquelle on peut préciser à chaque appel d'un allocateur sur quel nœud NUMA allouer les données. Il est déjà en partie possible de faire cela avec l'appel hwloc_alloc de la bibliothèque hwloc, il reste cependant à implémenter des

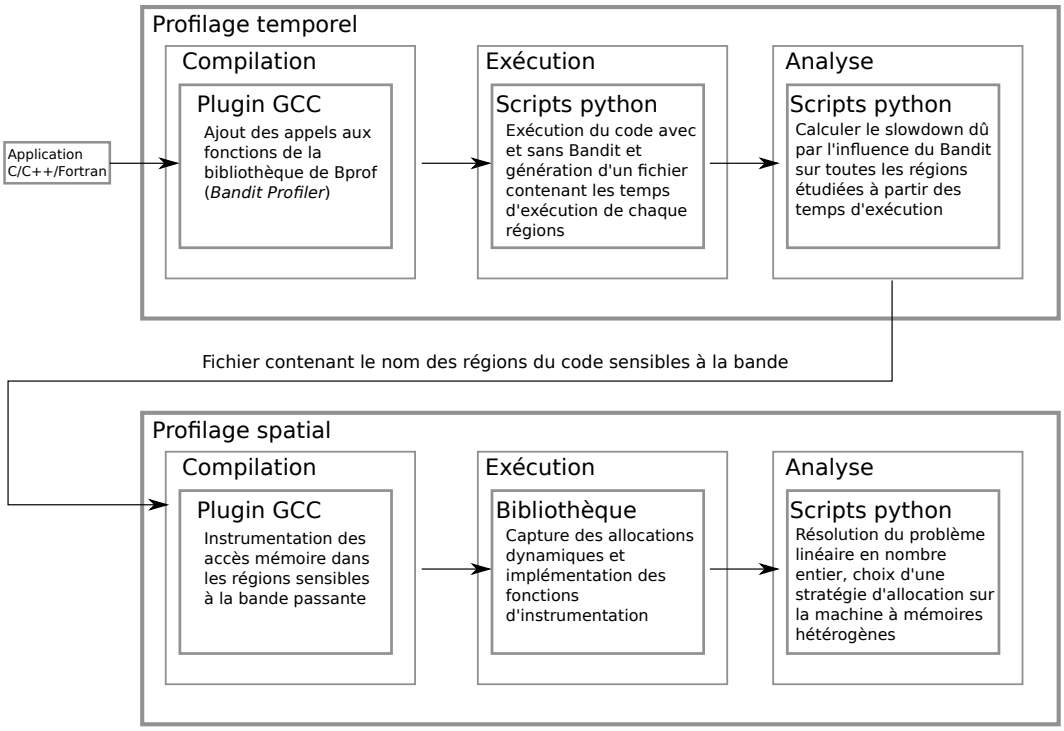


FIGURE 4.3.1 – Schéma de l'ensemble de la boîte à outils

fonctions similaires pour les allocations de type realloc et calloc.

4.4 Vision d'ensemble de la boîte à outils

La figure 4.3.1, représente une vue d'ensemble de notre boîte à outils de profilage et d'allocation des données sur KNL. L'analyse temporelle est implémentée avec un plugin de compilation pour instrumenter le code, une bibliothèque dynamique pour exécuter le Bandit et des scripts d'analyse des résultats pour sélectionner les régions temporelles ; L'analyse spatiale est implémentée avec un autre plugin de compilation afin d'instrumenter les accès en lectures et en écritures des régions temporelles sélectionnées, une bibliothèque dynamique pour capturer les appels d'allocation dynamique (malloc, realloc, calloc) et des scripts d'exécution et d'analyse de la trace obtenue. L'exécution du Bandit dépend de quelques paramètres permettant de sélectionner les détails de son exécution (cf partie 4.1.3, page 60), et notre boîte à outils contient des scripts python permettant de faciliter la sélection des valeurs pour ces paramètres.

4.5 Travaux Connexes

Dans notre méthode, l'insertion de sondes oblige le programmeur à modifier son code après avoir profilé son application, mais ces modifications sont mineures, et cette tâche est automatisable avec une passe de compilation. Cependant, comme nous l'avons vu, cette instrumentation par sonde ne permet pas d'identifier parfaitement un objet de données.

Il y a donc un compromis à trouver entre le surcoût dû à l'analyse de la pile d'appels entière et la précision de la sélection. En effet, la résolution du problème d'optimisation est basée sur une identité pour chaque donnée caractérisée par sa pile d'appels et son adresse virtuelle. Lorsqu'une paire de sondes est insérée pour signifier à la bibliothèque de placer la donnée sur une mémoire, l'information de la pile d'appels est réduite à un unique appel, il est donc possible que certaines données qui ne sont pas sélectionnées pour être allouées sur la MCDRAM le soit finalement lors de l'exécution.

Dans [Wu et al., 2017a] les auteurs proposent un support d'exécution, *Unimem*, qui choisit à la volée les données à mettre sur la DRAM en supposant que par défaut toutes les données sont sur la NVM. C'est un problème analogue à celui de placer les données entre la MCDRAM et la DDR4 et notre formulation inclus aussi ce cas de figure d'AMH. Ils développent ainsi une approche proactive et dynamique en répondant à deux questions :

— Évaluation statique : Comment capturer et caractériser les patterns d'accès à la mémoire associée aux objets de données ?
— Approche dynamique : Comment trouver le bon équilibre entre déplacer les données sur la bonne mémoire en fonction des patterns d'accès qui évolue au cours de l'exécution et ne pas payer trop cher le coût de déplacement des données ?

Comme dans notre approche, ils découpent l'exécution de l'application en phase temporelle, ils utilisent la bibliothèque PMPI ce qui permet un profilage ne nécessitant pas de compilation spécifique contrairement à notre outil, cependant leur profilage est limité au découpage MPI, alors que notre boîte à outil permet n'importe quel découpage, notamment la prise en compte des boucles parallèles OpenMP. Le découpage MPI se compose des phases de calculs entre deux opérations MPI, des phases de communication (collectives ou point à point) et des phases de synchronisation. Là où notre approche se veut générique, eux ciblent particulièrement les applications contenant une boucle itérative, comme le benchmark de Gradients Conjugués de la suite de benchmarks NAS [Bailey et al., 1991], ce qui leur permet d'intégrer la phase de profilage dynamique à l'exécution réelle de l'application et d'éviter une exécution préalable lors d'un profilage hors ligne.

$$BW(data) = \frac{\#LLCmissesSampled(data) \times CacheLineSize}{\frac{\#LoadStoreSampled(data)}{\#TotalSamples} \times phase\_execution\_time} \tag{4.3}$$

Les auteurs utilisent le compte de LLCmiss pour étudier la sensibilité d'une région MPI aux caractéristiques de latence ou de bande passante mémoire. Cependant, ils ont conscience que des événements ne provoquant pas de LLCmiss, sont aussi responsables du trafic mémoire total : éviction de ligne de cache ou opération de pré-lecture. Les auteurs cherchent donc à résoudre ce problème en utilisant des coefficients correcteurs calibrés par benchmarks. L'équation 4.3 permet de faire le lien entre les accès mémoires mesurés avec les compteurs matériels et la sensibilité d'un objet aux caractéristiques de latence et de bande passante. Elle représente le calcul de la bande passante utilisée par un objet de donnée. Si la bande passante mesurée sur NVM est en dessous de 10% du pic de bande passante, l'objet est dit sensible à la latence. Si c'est supérieur à 80%, c'est à la bande passante, sinon la sensibilité n'est pas connue. Dans le cas du calcul du gain potentiel, si

la sensibilité est inconnue, ils supposent que l'objet est sensible à la caractéristique qui maximise le gain de performance.

Bien que les auteurs ne semblent pas prendre en compte la congestion mémoire due à des accès mémoires simultanés sur différents objets de données, et que l'utilisateur doive instrumenter les allocations dynamiques qu'il souhaite analyser avec *Unimem*, leur approche avec un profilage en ligne et une découpe à partir des régions MPI est très prometteuse.

4.6 Discussions

Dans ce chapitre nous avons décrit notre implémentation du Bandit de bande passante, sa validation et son adaptation pour en faire un outil de profilage temporel. L'outil fait partie d'un ensemble permettant de trouver les données de l'application sensibles à la bande passante, d'instrumenter le code en conséquence, et d'exécuter l'application avec une surcharge des allocateurs dynamiques conventionnels afin que les données sensibles à la bande passante soient allouées sur la mémoire délivrant le plus de bande passante maximale.

En l'état, notre implémentation ne permet pas de tenir compte du problème relevé dans la partie 3.3 du chapitre précédent. En effet, notre profilage dynamique attribue une même valeur (coefficient de la fonction objectif) à tous les objets de données accédées dans une même région, alors qu'il serait nécessaire de pouvoir les différencier lors de la résolution du problème d'optimisation. Dans ce but nous pourrions intégrer une métrique généralement employée pour caractériser les besoins mémoires (LLCmiss), car même si celle-ci ne rend pas exactement compte du trafic mémoire, nous pensons que le couplage avec notre métrique temporelle peut donner de bons résultats.

Chapitre 5

Évaluation de la méthode sur une Architecture à Mémoires Hétérogènes

Dans le chapitre précédent nous avons développé la manière dont a été implémenté notre méthode d'allocation de donnée sur une architecture à mémoire hétérogène. Afin de valider notre méthode, nous l'avons mise en œuvre sur une architecture réelle : le KNL. Dans ce chapitre, nous expliquons l'impact de notre méthode sur la taille du problème à traiter. De plus, les différences de performances obtenues entre notre méthode et une allocation naïve sont détaillées. Pour cela nous avons sélectionné trois mini-applications représentatives d'applications scientifiques réelles et nous les avons exécutées sur KNL.

5.1 Cadre d'expérimentation

Parmi les trois mini-applications sélectionnées, deux sont issues du projet MANTEVO, ce sont les benchmarks CORAL MiniFE et LULESH, l'autre est une des mini-applications développées au CEA, HydroMM. MiniFE est représentative des applications implémentant une méthode de décomposition en éléments finis, alors que LULESH est représentative des applications de simulations hydrodynamiques. HydroMM est une mini-application du CEA représentative des applications de simulations hydrodynamiques multi-matériaux.

5.1.1 Architectures

Le KNL [Sodani, 2015] est une architecture Many Integrated Core (MIC), c'est la seconde génération des produits Intel Xeon Phi, et le premier CPU à intégrer la technologie HMC[Silicon, b, Pawlowski, 2011] dans une mémoire sur puce (MCDRAM) ; celle-ci vient augmenter les performances de bande passante mémoire du processeur. L'architecture du KNL sur laquelle les expériences ont été menées est composée de 34 tuiles contenant chacune 2 cœurs, eux-mêmes pouvant exécuter jusqu'à 4 processus légers utilisant la technologie SMT (hyperthreads), pour un total de 272 threads s'exécutant en parallèle sur la machine (schéma hwloc en appendice, cf figure .0.3, page 102). Chaque cœur possède un cache L1 qui n'est pas partagé et chaque tuile possède un cache L2, partagé au sein de chaque tuile du KNL par les deux cœurs la composant.

Le KNL peut être utilisé dans différents modes définissant : la manière dont le matériel gère les accès NUMA, la vision logicielle de la structure NUMA et la manière (logicielle et/ou matérielle) dont est gérée la mémoire sur puce MCDRAM. En ce qui concerne la MCDRAM, le mode **Cache** permet une totale transparence de la mémoire du point de vue des couches logicielles. C'est le matériel qui gère la mémoire comme une mémoire cache de niveau 3 (dernier niveau). Le mode **Flat** dévoile explicitement la mémoire, les couches logicielles la voient alors comme une mémoire d'un autre nœud UMA sur lequel il n'y a pas d'unité de calcul. Pour que des données soient allouées sur la MCDRAM, il faut alors qu'un acteur des couches logicielles choisissent explicitement ces données. Le dernier mode se situe entre les deux, c'est le mode **Hybrid** : il divise la mémoire en deux parties de même capacité (8GB), une gérée en mode **Cache** et l'autre en mode **Flat**. Lorsque la MCDRAM est utilisée en mode **Flat**, elle est dévoilée à la pile logicielle sous la forme d'un second nœud UMA qui ne contient aucune unité de calcul mais une mémoire de 16GB. Cela permet d'allouer les données sur le KNL avec les options de la commande Linux *numactl* :

- numactl -m 0 alloue toutes les données de l'application sur la DDR4, c'est ce mode d'exécution qui donne les performances de référence auxquelles on compare toutes les autres stratégies de placement de données ;
- numactl -m 1 alloue toutes les données de l'application sur la MCDRAM, la DDR4 n'est pas utilisée, même si la capacité mémoire de la MCDRAM ne suffit pas à stocker toutes les données de l'application ;
- numactl -p 1 alloue les données de l'application sur la MCDRAM dès que possible, s'il n'y a plus de place, les données sont allouées sur la DDR4. Si de la place se libère par la suite, d'autres données peuvent alors être allouées sur MCDRAM, mais il est à noter qu'une fois une donnée allouée celle-ci ne sera pas déplacée.

Ainsi, utiliser la commande linux *numactl -m 0* avec le mode **Hybrid** agit comme si la mémoire cache de dernier niveau était divisée par deux, par rapport au mode **Cache**.

En plus des modes pour gérer la MCDRAM, il est possible de choisir la manière de gérer les grappes de mémoire cache, et de qui se charge de s'assurer de la localité des accès : le programmeur, ou bien le matériel [Sodani, 2015]. Les différents modes sont le mode *quadrant*, qui permet une gestion entièrement matérielle de la localité des accès à

la mémoire, DDR4 comme MCDRAM. Les modes *snc2* ou *snc4* permettent aux couches logicielles de visualiser les différents nœuds de l'architecture NUMA du KNL, et donc la localité des accès mémoires est sous la responsabilité des couches logicielles. Toutes nos expériences sont faites avec le processeur démarré en mode *quadrant*, ainsi c'est le matériel qui garantit les performances des accès aux caches L2 et nous ne devons pas travailler logiciellement sur la localité des accès.

Le processeur Sandy Bridge est composé de 8 cœurs, 3 niveaux de mémoires caches, un privé et deux partagés. Les nœuds utilisés pour le profilage temporel avec notre Bandit sont des nœuds contenant deux sockets de ce processeur, reliées par un lien QPI. Ces machines sont donc NUMA, et elles valident les besoins que nous avons pour l'exécution de notre Bandit de bande passante comme cela a été montré dans la partie 4.1.3 du chapitre 4.

Le processeur Nehalem est composé de 4 cœurs, 3 niveaux de mémoires caches, un privé et deux partagés. Comme avec le Sandy Bridge, les machines utilisées pour le profilage au Bandit sont des nœuds contenant deux sockets de ce processeur reliées par un lien QPI.

5.1.2 Benchmarks

LULESH

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) est un code issu de la suite de benchmarks CORAL, dont le but est de représenter la complexité des accès mémoire sur une application de simulation de chocs hydrodynamiques (stencil). Il existe plusieurs versions disponibles et c'est la version 2.0 [Karlin et al., 2013] qui est exécutée dans les expériences de ce chapitre. Ce code est composé d'une boucle principale sur un pas de temps calculé à chaque itération. Les itérations s'arrêtent soit lorsque le nombre limite d'itérations est atteint (infinie par défaut), soit dès que le temps de simulation à calculer est atteint (0.01 secondes par défaut).

LULESH approxime le problème de simulation de chocs hydrodynamiques à l'aide de la méthode des volumes finis sur une grille cartésienne. Cependant, les structures de données sont volontairement complexes pour représenter les accès mémoire de toute simulation s'appliquant sur des géométries irrégulières et non structurées, et ne doivent pas être optimisées à la main (notamment pour bénéficier de la régularité de la grille cartésienne). En particulier, ce benchmark stresse la vectorisation du compilateur, le surcoût des supports exécutif d'OpenMP, et du parallélisme sur nœud. LULESH doit être lancé avec un nombre de processus MPI égal au cube d'un entier (1, 8, 27, 64, 125, 512 ...), cela est dû au caractère tri-dimensionnel de la répartition des processus sur le problème traité. Le nombre de threads OpenMP lancé pour chaque processus MPI n'a pas de limitation à priori, bien qu'avec un nombre de processus inférieur au nombre de cœurs disponibles, les performances sont généralement meilleures, car les problèmes de partage de ressource sont ainsi minimisés. En pratique, il n'est pas trivial de lancer un nombre de threads différent par processeur MPI, cela demande un effort de programmation supplémentaire ; de même, attacher les processus à des cœurs spécifiques de la machine demande aussi des efforts de programmation ; finalement, mise à part les puissances de 2, la contrainte d'un nombre cubique de processus MPI rend difficile une utilisation optimale

des machines. Nos tests sont faits avec un seul processus MPI et se concentre sur les effets induits par le parallélisme de mémoire partagé OpenMP.

En ce qui concerne la taille des problèmes, il est conseillé d'instancier entre 25 000 et 30 000 éléments par GB de mémoire [cor,]. Pour un unique processus MPI et une machine de 64GB cela correspond environ à une taille de problème de 124 mailles par dimension, parce que le paramètre d'entrée donne une des trois dimensions et $1600000 < 124^3 = 1906624 < 1920000$. Cependant, dans nos expériences nous irons des tailles 50^3 à 400^3 mailles par dimension comme d'autres auteurs qui se sont intéressés au placement de données sur KNL [Peng et al., 2017]. Cette décision s'explique par le fait qu'en pratique il faut moins de 16GB pour stocker les données avec une taille en entrée inférieure à 250^3 et nos expériences sont faites pour stresser l'utilisation de la MCDRAM qui a une capacité de 16GB.

Pour vérifier le résultat de la simulation, il faut regarder les valeurs écrites en fin d'exécution. Pour un problème avec une même configuration, les énergies à l'origine finales doivent être les même (pour les chiffres affichés). Le nombre d'itérations effectuées doit aussi être identique, et les valeurs symétriques pour l'énergie à l'origine finale doit être inférieure à 10^{-8} pour des petits problèmes de taille $taille \times MPIRanks^{1/3} < 100$. Nous avons donc pu vérifier la cohérence des résultats obtenus après chaque exécution.

MiniFE

MiniFE est une mini application implémentant des noyaux représentatifs des applications développant la méthode implicite des éléments finis. La méthode développée assemble un système linéaire creux depuis l'état d'équilibre des équations de conduction sur un problème découpé en briques de 8 éléments. Le système linéaire est ensuite résolu avec un gradient conjugué préconditionné simplement.

Ce code contient les noyaux suivant :

— Calcul des opérateurs d'élément (matrice de diffusion, vecteur source) ;
— Assemblage (éclatement des opérateurs d'éléments en matrice creuse et vecteur) ;
— Produit d'une matrice creuse et d'un vecteur (Résolution du Gradient Conjugué) ;
— Opération sur les vecteurs (BLAS de niveau 1 : axpy, dot, norm).

En fin d'exécution, MiniFE compare la solution calculée avec la solution analytique pour un état d'équilibre de température dans un cube.

La taille d'un problème se définit sur trois dimensions, la taille selon chaque dimension n'est pas nécessairement constante contrairement à LULESH dont la taille ne peut être modifiée qu'uniformément selon les trois dimensions. La documentation pour MiniFE propose quatre tailles de problèmes : petite (16 processus MPI, environ 250 mailles par dimension), moyenne (moins de 1000 nœuds, environ 10 000 processus MPI et environ 2000 par dimension), grand (environ 4000 nœuds, 65 000 processus, 4000 par dimension) utilisé comme référence CORAL pour tester le supercalculateur IBM BG/Q et enfin les problèmes de taille *classe CORAL* environ deux fois supérieur au précédent (double le nombre de nœuds et de processus MPI selon la deuxième dimension). Dans notre démarche, uniquement les problèmes de petites tailles testant le parallélisme OpenMP ont été lancés.

HydroMM

HydroMM est un benchmark représentatif des codes de simulation de chocs hydrodynamique multi-matériaux, il résout un problème similaire à celui résolu par Hydro2D [Sewall and Colin de Verdière, 2015]. La différence principale consiste à l'aspect multi-matériaux de HydroMM. C'est un code hautement parallélisé avec les pragmas OpenMP.

Il est composé d'une boucle reposant sur le calcul d'un pas de temps à la manière de LULESH. Les données accédées par l'application sont regroupées dans trois allocations dynamiques : qInterf, qMean, et un allocateur interne à l'application qui regroupe les allocations de tuiles sous une seule méthode. Cette allocation de tuiles rend difficile l'identification des données.

5.2 Profilage des applications

La démarche consiste à faire une analyse temporelle des applications en suivant la méthode décrite dans la partie 3.2, puis d'effectuer une analyse spatiale pour enfin résoudre le problème d'allocation de données (cf partie 3.3). Ces analyses, ou profilages, sont effectuées sur des machines différentes du KNL, un processeur Nehalem et un Sandy Bridge, en donnant les même résultats. S'ils ne sont pas faits sur KNL, c'est que l'implémentation de notre Bandit n'a pas été testée, et donc validée ou invalidée, sur cette machine.

5.2.1 Analyse temporelle

L'analyse temporelle permet de sélectionner les fonctions ou régions du code analysées ensuite par l'analyse spatiale. Le ralentissement induit par le Bandit sur chaque région du code est comparé au seuil établi lors de la validation de notre Bandit (cf partie 4.1.3, chapitre 4) qui est égal à 10%. À ce seuil nous avons arbitrairement décidé de ne considérer que les régions temporelles représentant au moins 1% du temps d'exécution total de l'application.

Les tableaux 5.2.1 et 5.2.2 représentent respectivement la sensibilité au Bandit de bande passante (SBB, définie partie 4.1.3, page 62) de LULESH et de MiniFE. Cette sensibilité est mesurée lors de l'exécution du programme, et comme tout profilage dynamique, elle dépend à priori des paramètres d'entrée du programme. Nous considérons que les paramètres d'entrée sont uniquement la taille du problème et le nombre d'itérations. Les autres paramètres d'entrées de MiniFE, tels que ceux sélectionnant tel algorithme plutôt qu'un autre et induisant des branchements conditionnels, sont fixés aux valeurs par défaut pour toutes les expériences. Les résultats des tableaux 5.2.1 et 5.2.2 montrent qu'à partir d'un certain nombre d'itérations et d'une certaine taille, il n'y a plus de variations significatives de l'influence du Bandit sur le temps d'exécution de l'application. Cela signifie que le profilage au Bandit ne dépend pas, dans une certaine mesure, des paramètres d'entrée. On peut donc profiler pour une certaine taille de problème et extrapoler lors de l'exécution d'autres taille de problème. Notons qu'alors que LULESH possède deux paramètres d'entrée, comme HydroMM, qui sont la taille et le nombre d'itérations, seule la taille est paramétrable dans MiniFE.

Taille	SBB (en %)
50	112
100	120
150	134
200	137
250	138
300	139

(a) Différentes tailles. 30 itérations.

Itérations	SBB (en %)
1	113
10	120
20	120
30	120
40	122
50	122

(b) Taille fixée (100). Différentes itérations.

Itérations	SBB (en %)
1	120
10	132
20	133
30	134
40	134
50	135

(c) Taille fixée (150). Différentes itérations.

TABLE 5.2.1 – Sensibilité au Bandit de bande passante (SBB, donné en pourcentage de ralentissement) de LULESH en fonction des paramètres d'entrée du programme

Taille	50	100	150	200	250	300	350	400
SBB	1.39	1.43	1.44	1.46	1.45	1.45	1.45	1.44

TABLE 5.2.2 – Sensibilité au Bandit de bande passante (SBB, définie partie 4.1.3, page 62) de miniFE en fonction des paramètres d'entrées, avec le nombre d'itérations fixé à 200

5.2.2 Analyse spatiale

Dans cette partie nous montrons les résultats obtenus en termes de nombre d'objets à analyser pour résoudre le problème d'allocation de données sur KNL grâce à notre analyse temporelle (Bandit). Cette réduction consiste à évaluer dès que possible certaines des variables de décision de la fonction objective décrite page 49 (cf équation 3.14). Dans le cas du KNL, $M = 2$ (MCDRAM+DDR4) et pour une donnée i si $t_{[iMCDRAM]} \geq t$ il n'y a aucun avantage à mettre cette donnée sur MCDRAM (parce que c'est la mémoire avec le moins de capacité) : on peut donc directement évaluer $x_{[iDDR4]}$ à 1 et $x_{[iMCDRAM]}$ à 0. On considère alors que toutes ces données n'ont plus besoin d'être considérées lors de la sélection des données à placer en MCDRAM. Nous détaillons aussi les trois catégories de taille de problème : *grande*, *moyenne* et *petite*. Ces termes seront réutilisés dans la suite du chapitre.

Le seuil introduit dans la partie 3.3 (inéquation 3.18, page 52) est considéré comme

étant toujours vérifié étant donné les caractéristiques du KNL. Donc toutes les données accédées au moins une fois dans une région sélectionnée comme sensible à la bande passante est considérée comme importante dans le choix des données à allouer en MCDRAM.

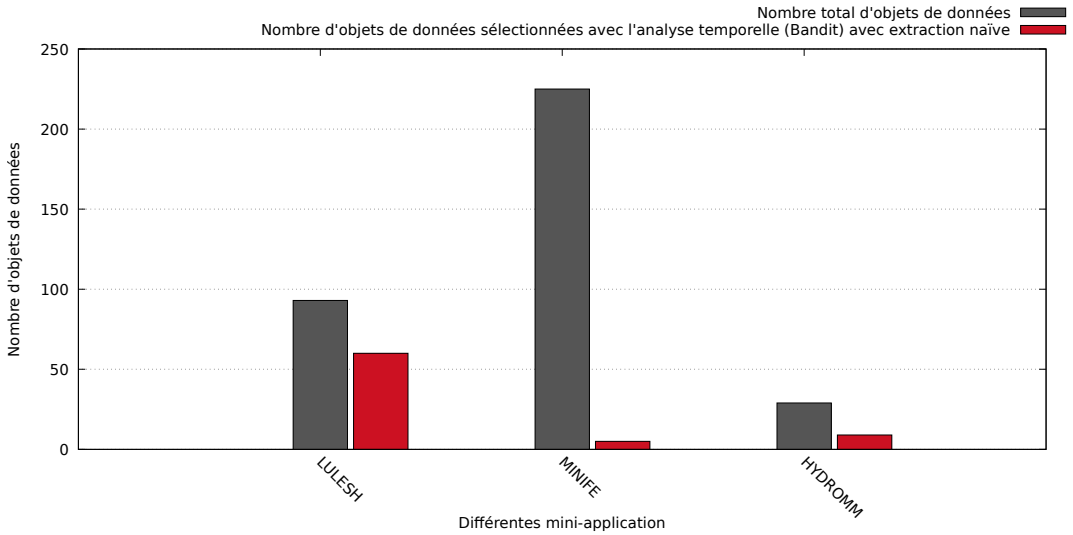
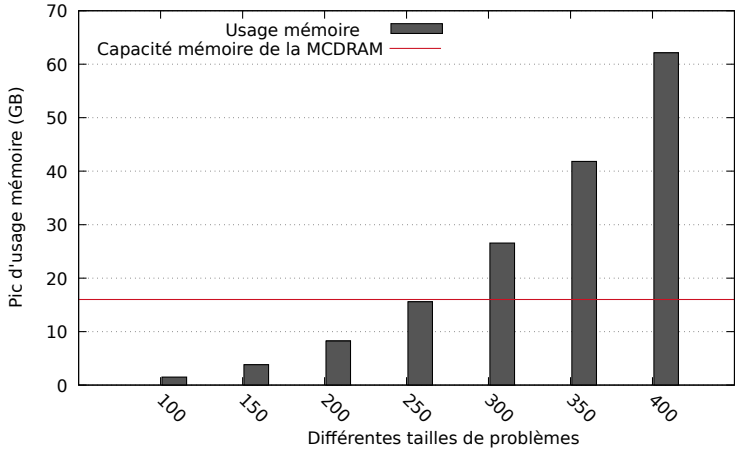


FIGURE 5.2.1 – Nombre d'objets de données alloués dynamiquement

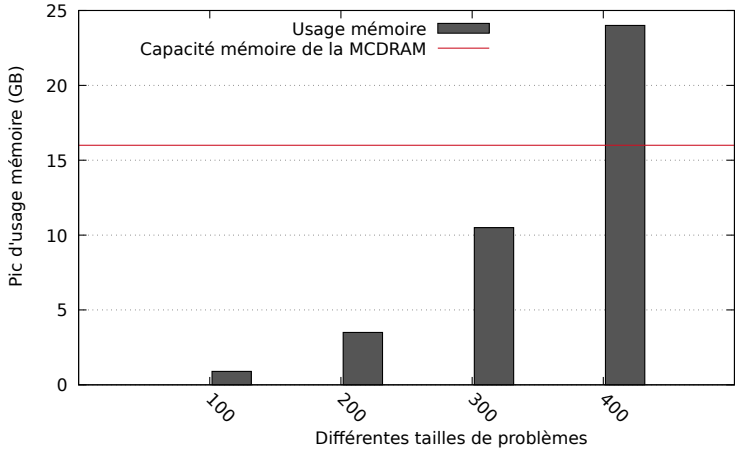
La figure 5.2.1 permet de visualiser la quantité d'objets de données marqués par notre méthode d'analyse temporelle (Bandit) avec une extraction naïve des données (cf partie 4.2.2, page 69), par rapport au nombre total d'objets alloués dynamiquement, pour chaque application. L'extraction naïve des données consiste simplement à extraire de l'application toutes les données allouées dynamiquement et accédées dans les régions temporelles marquées par l'analyse avec le Bandit. Un objet de donnée correspond à une ligne du code à partir de laquelle une allocation dynamique est effectuée (cf chapitre 4, partie 4.2.1, page 68). Sur certaines applications, comme MiniFE dans laquelle de nombreux objets ne sont instanciés que pour le post-traitement des données, le nombre d'objets de données finalement conservés pour la résolution du problème d'allocation peut être drastiquement réduit.

La figure 5.2.2 représente l'usage de mémoire physique maximale (récupéré en lisant les informations de */proc/cpuinfo*). Ce pic d'usage mémoire ne doit pas dépasser la capacité de la mémoire physique. Cette contrainte est représentée par la contrainte de capacité du problème d'optimisation défini dans le chapitre 3. Pour les problèmes de *petite* taille, l'usage mémoire total ne dépasse pas la capacité de la plus petite mémoire du système, qui est la MCDRAM dans le cas du KNL. Dans ce cas, il n'y a pas de contrainte de capacité à prendre en compte et seul les coefficients de la fonction objective de la formulation permette de sélectionner les données à placer sur telle ou telle mémoire.

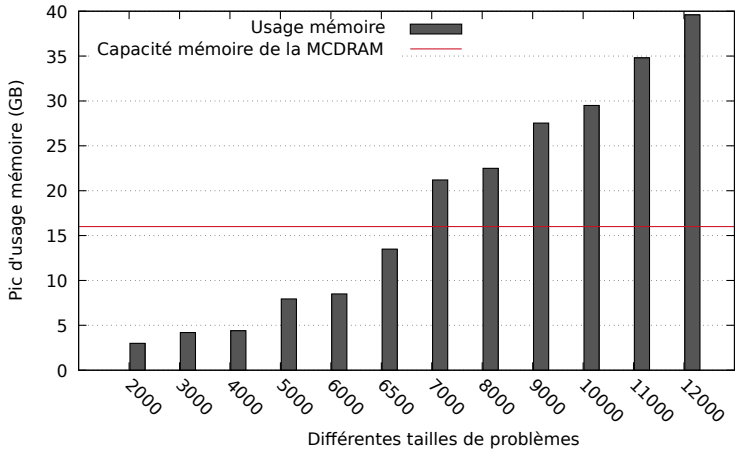
Lorsque l'usage mémoire maximal dépasse la capacité mémoire de la MCDRAM, il est nécessaire de prendre en compte la contrainte de capacité mémoire, on parle alors de problème de taille *moyenne*. Pour cela il faut connaître l'usage mémoire maximal de chacun des objets de données de manière indépendante. Notre outil de profilage spatial nous



(a) LULESH



(b) miniFE



(c) HydroMM

FIGURE 5.2.2 – Pic d'usage mémoire des applications scientifiques testées pour différentes tailles de problèmes

permet de récupérer cette information pour toutes les allocations dynamiques. Cependant, l'usage mémoire ainsi récupéré correspond à un usage de mémoire virtuelle. C'est donc une contrainte conservatrice, plus forte que nécessaire, qui est considérée lors de la résolution du problème d'allocation de données. En effet, il n'est pas certains que l'objet de données soit réellement allouées physiquement lorsque l'allocation dynamique virtuelle a été faite.

Enfin les problèmes de *grande* taille correspondent au cas où l'usage mémoire maximal dépasse de loin la capacité mémoire de la MCDRAM, il est possible que l'usage mémoire indépendant de chaque objet de donnée dépasse déjà cette capacité mémoire. Dans ce cas aucun objet de données ne peut être placé sur la MCDRAM sans un nouveau découpage au préalable.

5.2.3 Résolution

```
1 lulesh-init.cc:483|lulesh-init.cc:120|lulesh.cc:2896|
2 vector.tcc:561|lulesh.h:154|lulesh-init.cc:73|lulesh.cc:2896|
3 lulesh-init.cc:278|lulesh-init.cc:117|lulesh.cc:2896|
4 lulesh.cc:2553|lulesh.cc:2782|lulesh.cc:2906|
```

Listing 5.1 – Trace de Données solutions

Pour un problème de `petite` taille, toutes les données peuvent être placées en MCDRAM, si telle est la meilleure stratégie. Par contre, même si l'application aurait de meilleures performances avec toutes ses données en MCDRAM, un problème de taille *moyenne* ne permet pas de placer toutes les données en MCDRAM et il faut faire un choix. Par conséquent, la résolution du problème d'allocation se fait en fonction de la taille du problème à exécuter. Le code (cf listing 5.1) représente une partie de la sortie de notre outil précisant les données à allouer sur la MCDRAM pour le code LULESH. Une fois cette information connue, le programmeur instrumente (cf listing 5.2) le code en prenant l'appel de la pile identifiant au mieux chaque objet de données comme décrit dans la partie 4.2.1, page 68.

Dans le code instrumenté, ce sont les appels à notre bibliothèque *switchtender* qui vont allouer dynamiquement sur la mémoire rapide les tableaux dont les appels à malloc sont compris entre les sondes. Dans l'exemple (listing 5.2), les sondes encadrent exactement le code des allocations, parce que dans LULESH les allocations sont isolées de cette manière, mais dans d'autres applications, il peut y avoir du code entre les sondes et les allocations ciblées sans modifier le fonctionnement de notre bibliothèque : toutes les allocations dynamiques comprisent entre les sondes sont considérées comme marquées.

```
IntegrateStressForElems( Domain &domain,
        Real_t *sigxx, Real_t *sigyy, Real_t *sigzz,
        Real_t *determ, Index_t numElem, Index_t numNode)
{
    [...]

    Index_t numElem8 = numElem * 8 ;
    Real_t *fx_elem;
    Real_t *fy_elem;
    Real_t *fz_elem;

    switchtender_switch_on();

    fx_elem = Allocate<Real_t>(numElem8) ;
    fy_elem = Allocate<Real_t>(numElem8) ;
    fz_elem = Allocate<Real_t>(numElem8) ;

    switchtender_switch_off();

    [...]
}
```

Listing 5.2 – LULESH : exemple de code instrumenté

5.3 Évaluation des performances

Dans cette partie nous évaluons les performances de notre méthode en exécutant sur le processeur KNL chacune des mini-applications scientifiques sélectionnées. Les quatre modes de la MCDRAM du KNL ont été testés sur chacune de ces mini-applications : le mode cache, hybrid et flat (cf partie 5.1.1). Cela permet de mettre en regards notre méthode avec des solutions d'outils existants, simples à utiliser. Nous allons montrer que la méthode donne de bons résultats sur une partie des cas testés, et nous expliquerons pourquoi elle reste limitée pour certains cas.

5.3.1 HydroMM

Le code de HydroMM a été exécuté pour trouver la meilleure configuration du nombre de threads à exécuter sur KNL. Chaque cœur de la machine peut exécuter jusqu'à 4 hyperthreads en parallèle. Lorsque des processus légers s'exécutent sur un même cœur (hyperthreads), ce n'est pas aussi efficace que s'ils s'exécutaient sur des cœurs distincts. En effet, en plus de partager tous les niveaux de mémoire cache, ils partagent certaines unités de traitement d'instructions, et selon les instructions effectuées par les processus légers, ce parallélisme peut induire une baisse de performance par rapport à une exécution sans utilisation de la technologie des hyperthreads. La figure 5.3.1 contient les résultats de ces exécutions, on peut en déduire qu'avec 136 processus légers, et 2 processus légers par cœur l'application atteint ses performances optimales pour un problème de taille 2048^3. On utilisera cette configuration pour toutes les expériences à mener sur KNL, en faisant l'hypothèse que ce résultat reste vrai quelle que soit la taille du problème.

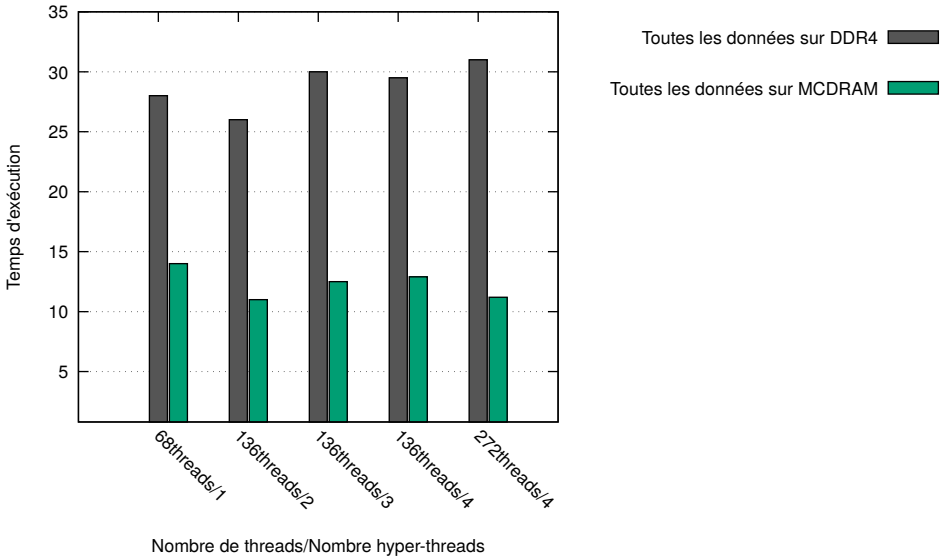


FIGURE 5.3.1 – Sélection du nombre de processus légers pour un temps d'exécution minimal sur KNL avec HydroMM (taille de problème fixée à 2048^3)

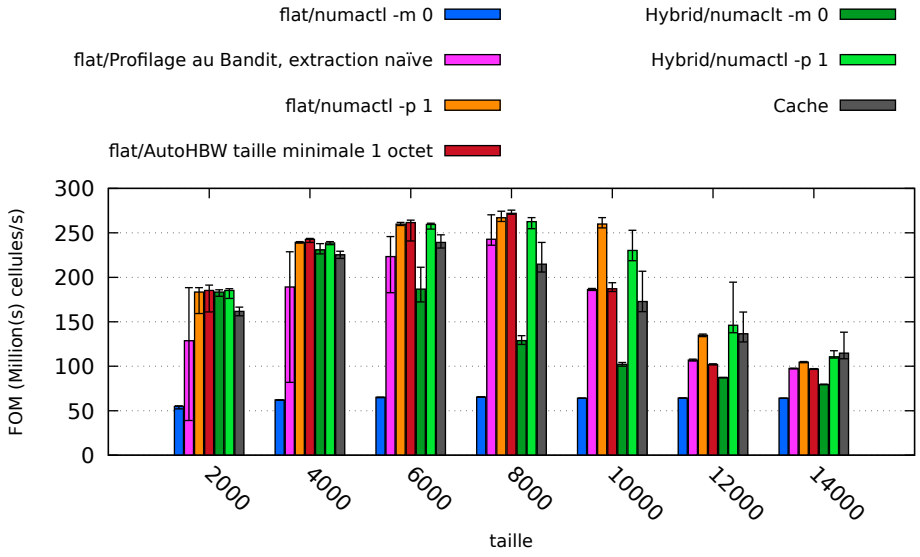


FIGURE 5.3.2 – HydroMM exécuté du KNL : Comparaison des performances entre notre méthode (rose) et les approches de l'état de l'art en fonction de la taille du problème exécuté

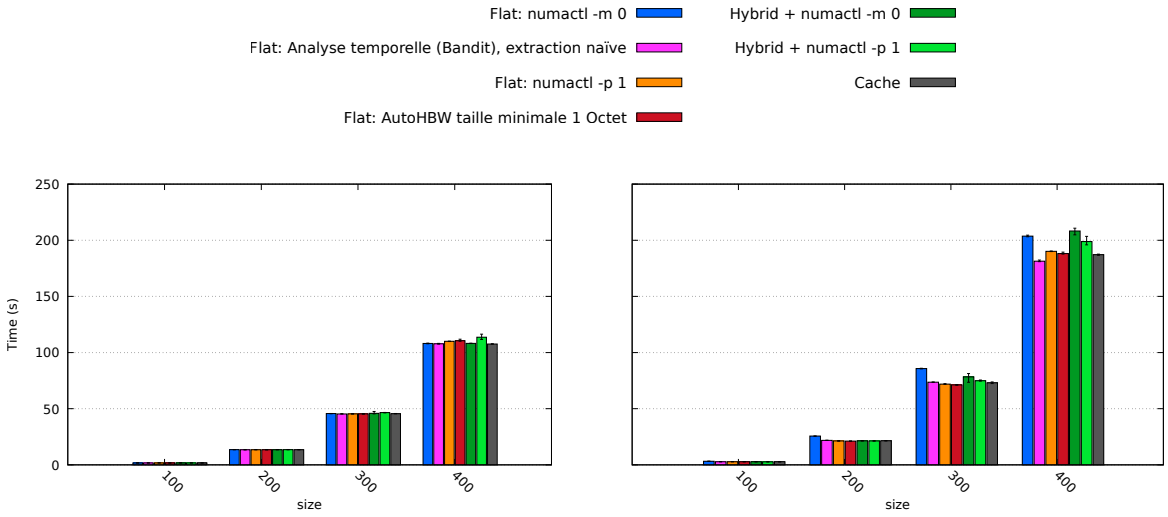
La figure 5.3.2 représente les performances sur KNL de la mini-application HydroMM. Le code de HydroMM a été optimisé pour maximiser la parallélisation des accès à la mémoire, ses performances sont donc en grande partie limitées par la bande passante mémoire. C'est pourquoi, en plaçant les données correspondantes à la majorité des accès sur

une mémoire à forte bande passante (MCDRAM), on peut observer un gain de performance significatif. C'est ce que met en évidence la comparaison des FOMs de l'exécution avec numactl -m 0 (première barre, en bleue, de chaque groupe) où toutes les données sont sur la DDR4, aux FOMs de l'exécution avec numactl -p 1 ($3^{ème}$ barre, en orange, de chaque groupe) où les données sont placées autant que possible en MCDRAM. Pour les tailles de problème inférieure à 8000 (*petites tailles*) on observe que les performances obtenues avec notre méthode (flat/Profilage au Bandit, extraction naïve) sont équivalentes aux autres méthodes de placement de données en MCDRAM. En effet lorsque l'ensemble des données de l'application peuvent tenir en MCDRAM, si l'application ne possède pas de donnée sensible à la latence comme c'est le cas avec HydroMM, toutes les stratégies de placement de données se valent. Pour des tailles petites néanmoins (2000 et 4000), le temps d'exécution est court et l'utilisation répétée par notre méthode de l'allocateur memkind induit un coup imprévisible qui n'est pas négligeable, comme le mettent en évidence les barres d'erreurs sur les performances de notre méthode (la seconde de chaque paquet, en rose).

Parce que dans HydroMM de nombreuses allocations sont effectuées avec la même pile d'appels il est difficile de prendre en compte efficacement les contraintes de capacité mémoire avec une méthode agissant à la granularité des objets de données. En effet, comme nous l'avons souligné dans la partie 5.1.2, une grande partie des objets de données sont des tuiles qui sont allouées et libérées par un allocateur interne. Notre méthode restreint l'identification à un appel de la pile, mais même avec la pile d'appels complète, il n'est pas possible de différencier le placement en mémoire de ces objets de données. C'est pourquoi, pour des problèmes de *taille moyenne* (supérieures à 8000), les performances de notre méthode se dégradent autant.

La stratégie de placement de donnée implémentées dans le matériel se doit d'être générique, et nécessairement, pour certaines applications elle n'est pas optimale. HydroMM en est un exemple puisqu'on peut observer que, sur des problèmes de taille moyenne, les performances du KNL avec la MCDRAM en mode **cache** (dernière barre de chaque paquet, en gris) sont moins bonnes que celles du mode **flat** en utilisant uniquement la commande numactl -p 1. Cela est probablement dû au fait que la stratégie d'éviction de lignes de cache amène parfois à déplacer en DDR4 des données qui sont pourtant responsables d'accès à la mémoire sensibles à la bande passante, ce cas peut intervenir si le pattern d'accès aux données entraîne du *false sharing* entre les processus légers.

Enfin on observe des performances meilleures pour le mode **hybrid**, avec la stratégie performante (numactl -p 1) pour la moitiée **flat**, que pour le mode **cache** ; résultat auquel on pouvait s'attendre puisque le mode hybrid permet quand même à la moitié de la capacité de la MCDRAM en mode flat de bénéficier de la stratégie de placement du numactl -p 1. Les performances du mode hybrid avec numactl -m 0 correspondent aux performances obtenues avec une mémoire cache possédant moitié moins de capacité, donc les performances sont encore plus basses que celles du mode **cache**.



(a) MiniFE : Temps de la partie générant la matrice

(b) MiniFE : Temps total du programme incluant le temps de génération de la matrice

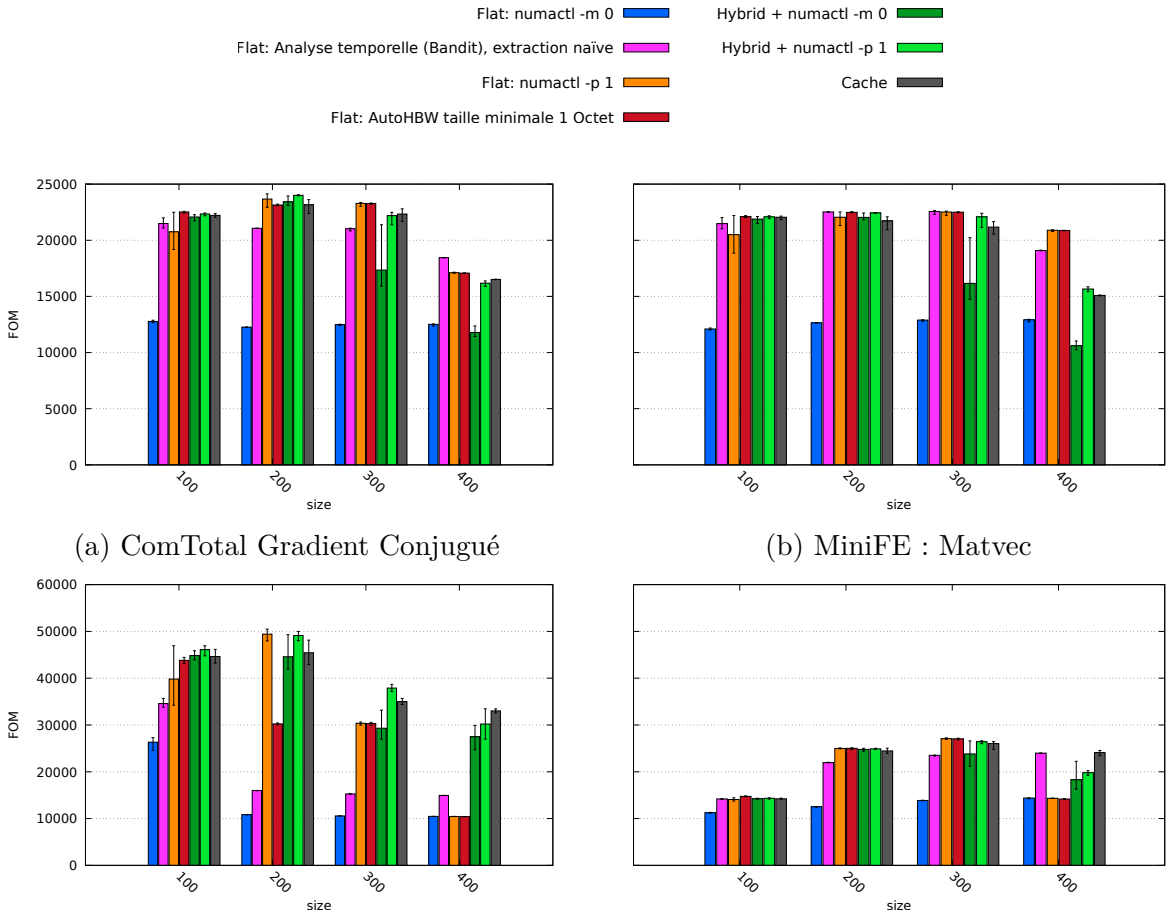
FIGURE 5.3.3 – Temps d'exécution de MiniFE en fonction de la taille du problème (racine cubique du nombre d'éléments) et comparaison de notre méthode (rose en deuxième position) avec les approches de l'état de l'art

5.3.2 MiniFE

La figure 5.3.3 représente le temps d'exécution total de miniFE (figure 5.3.3b) et le temps passé dans l'initialisation (figure 5.3.3a). On observe que la phase de génération de la matrice prend plus de la moitié du temps d'exéction. On peut constater aussi que notre méthode obtient les meilleurs résultats relativement aux autres méthodes.

Le seuil entre des problèmes de *taille petite* et des problèmes de *taille moyenne* se situe aux alentour de 350 dans les tailles de problèmes exécutés par MiniFE. C'est sur des problèmes de taille moyenne que notre méthode possède des résultats légèrement meilleurs que les autres stratégies de placement. L'analyse temporelle (cf partie 4.1, page 55) de notre méthode permet une sélection de donnée (extraction naïve) dont l'usage mémoire maximale total ne dépasse que d'un objet la capacité de la MCDRAM. Lors de l'exécution basée sur notre méthode avec extraction naïve, c'est la stratégie du premier arrivé premier servi qui prime. Lorsque cet objet tente d'être alloué sur la MCDRAM, sa taille empêche de pouvoir le placer sur la mémoire qui est déjà trop utilisée. Ce plus, cette allocation n'est pas responsable des accès mémoires effectués dans la région temporelle sélectionnée. Finalement, notre extraction naïve l'a sélectionnée, et il manque une analyse plus précise pour la différentier des autres objets de données.

La figure 5.3.4 représente le détail des performances des différents noyaux de calcul de miniFE lors de son exécution sur KNL. Ces noyaux sont MatVec, DOT, Waxpy, ils sont détaillés dans la partie 5.1.2 (page 78). La courbe 5.3.4a conglomère les résultats de tous ces noyaux en un. Sur ces courbes on observe l'influence de la sélection de données



(a) ComTotal Gradient Conjugué

(b) MiniFE : Matvec

(c) MiniFE : WAXPY

(d) MiniFE : DOT

FIGURE 5.3.4 – MiniFE exécuté sur KNL pour différentes tailles de problème et comparaison des performances de notre méthode (rose en 2^{eme} position) avec les approches de l'état de l'art. Décomposition des performances du Gradient Conjugué de MiniFE en 3 noyaux : Matvec, WAXPY et DOT.

opérées par notre analyse temporelle (Bandit) avec extraction naïve sur les différents noyaux. Ainsi, les données intervenant dans le noyau Matvec et DOT ont correctement été mises en évidence par notre analyse, alors que celles accédées dans WAXPY et sensible à la bande passante n'ont pas été retenues parce que l'influence globale de WAXPY est trop faible, elle n'appartient pas à $\mathcal{H}$ décrit dans la partie 3.2.3, page 49. Comme prévu, ce choix pour le noyau WAXPY a une influence négligeable sur les performances globales de l'application, puisque sur la courbe 5.3.3, notre méthode possède des performances semblables aux autre méthodes, voire meilleures.

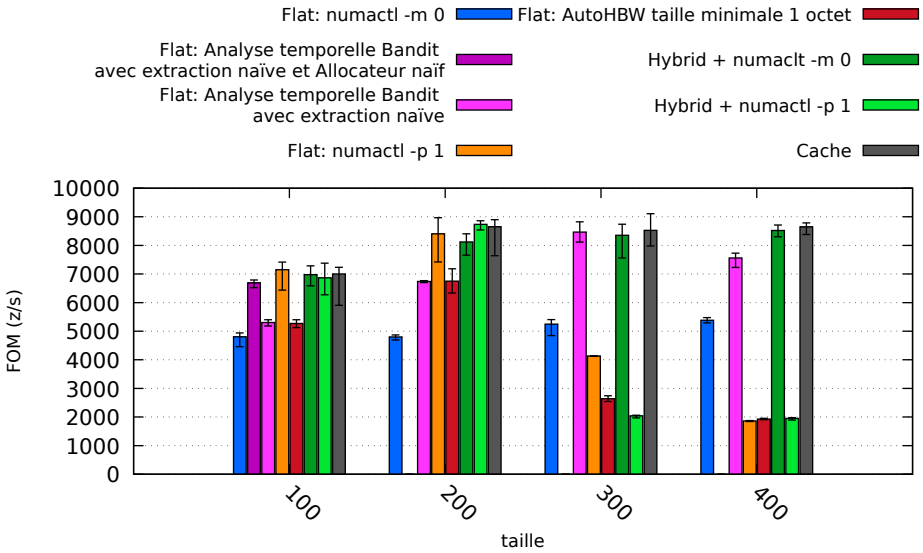


FIGURE 5.3.5 – LULESH exécuté du KNL : Comparaison des performances entre notre méthode (rose en 3^{eme} barre et violet en 2^{eme} position seulement sur taille de problème de 100^3) et les approches de l'état de l'art en fonction de la taille du problème exécuté

5.3.3 LULESH

Pour l'exécution de LULESH sur la plus petite taille de problème (100) nous avons testé un mode supplémentaire de ceux testés sur HydroMM et MiniFE, c'est le mode avec flat/ Analyse temporelle (Bandit) avec extraction naïve et **allocateur naïf**, la barre violette. Cet allocateur fait une première allocation de 16GB lors de la première allocation demandée sur MCDRAM, puis il se contente de donner des régions de cette allocation pour toutes les allocations suivantes. Cet allocateur n'est pas robuste, puisqu'il ne gère pas la libération de donnée, mais il permet d'observer l'intérêt de la méthode sur une petite taille de problème (100), là où l'utilisation de l'allocateur de la bibliothèque memkind pour chaque nouvelle allocation sur la MCDRAM induit un ralentissement non négligeable : $2^{ème}$ barre (violette), avec l'allocateur naïf, comparée à la $3^{ème}$ barre (rose), sans l'allocateur naïf.

Sur les tailles de problème supérieures, le fait que l'allocateur naïf ne gère pas les libérations de données, alors que LULESH en effectue beaucoup, ne permet pas d'exécuter l'application jusqu'à son terme. C'est pourquoi sur les tailles de problème supérieures, il n'y a pas la deuxième barre (violette) correspondant à l'allocateur naïf. Sur la taille de 200 un allocateur intelligent pourrait réduire le ralentissement non négligeable dû aux appels à l'allocateur memkind. Pour les tailles 300 et 400 les temps d'exécution sont suffisamment longs et le surcoût dû aux nombreux appels à l'allocateur de memkind est négligeable.

Finalement, sur des tailles de problème dont l'usage mémoire ne dépasse pas la capacité mémoire de la MCDRAM (100 et 200), notre méthode de sélection de données permet d'obtenir des performances équivalentes à une exécution pour laquelle on aurait placé

toutes les données sur la MCDRAM. Cela confirme le fait que notre méthode permet de sélectionner de manière exhaustive les données responsables de la sensibilité à la bande passante de l'application LULESH. De plus, notre méthode de sélection nous permet d'exécuter des problèmes en bénéficiant de la forte bande passante de la MCDRAM, alors même que l'usage mémoire ne permet pas de placer toutes les données de l'application en MCDRAM, En effet, pour les tailles de 300 et 400, on peut observer que notre méthode permet d'obtenir des performances 2 à 4 fois meilleures que les autres méthodes de placement de données pour une exécution sur KNL avec la mémoire MCDRAM en mode flat, c'est-à-dire gérée par les couches logicielles.

5.4 Conclusion

Dans ce chapitre nous avons cherché à valider notre méthode de sélection de données pour une allocation optimale sur une architecture à mémoires hétérogènes. Pour cela nous avons exécuté sur KNL différentes applications dont les performances sont sensibles à la bande passante mémoire. Ces applications ont été profilées avec notre méthode : analyse temporelle reposant sur notre bandit de bande passante et analyse spatiale reposant sur une extraction naïve des données accédées dans les régions temporelles retenues avec l'analyse temporelle.

Nous avons ensuite comparé les résultats obtenus avec notre méthode aux résultats obtenus avec des méthodes simples à utiliser par le programmeur. Pour les applications évaluées, notre méthode donne toujours des résultats proches des meilleurs résultats obtenus, toutes méthodes confondues. Alors que pour chaque autre méthode, il existe toujours une application pour laquelle les performances sont dégradées.

Notamment, notre méthode permet de mieux sélectionner les données à mettre en MCDRAM que les autres méthodes pour la mémoire **flat**, et exécuter une application dans ce mode donne le potentiel d'utiliser les 16GB de la MCDRAM pour y placer des données qui ne sont pas redondantes avec les données placées en DDR4, comme avec le mode **cache**.

Chapitre 6

Conclusion et perspectives

Les AMH font aujourd'hui partie du paysage des machines utilisées dans le HPC : GPGPU-CPU, KNL, etc. Ces architectures existent sous différentes formes, ce qui entraîne une diversité parmi les caractéristiques mémoires critiques dans la différenciation des mémoires composant une architecture (latence, bande passante, capacité, volatilité). Par conséquent, d'une architecture à l'autre, ce ne sont pas les mêmes caractéristiques mémoires qui permettent de différencier l'intérêt d'une mémoire sur une autre, pour y allouer certains objets de données dont les accès mémoires ralentissent l'exécution : parfois c'est la capacité, ou bien la bande passante, ou encore la latence ou bien même un mélange.

Notre étude prends comme exemple une AMH composée de deux mémoires principales : une mémoire à forte bande passante, avec une plus grande latence et une capacité limitée, la deuxième mémoire possède une grande capacité une bande passante bien plus faible et une latence légèrement plus rapide. Cette architecture correspond aux caractéristiques des mémoires principales du KNL. Le KNL intègre la technologie disruptive de mémoire empilée appelée MCDRAM. Cette mémoire possède 4 à 5 fois plus de bande passante que la DDR4 mais une latence 10% plus lente, et sa capacité ne permet pas de répondre aux attentes des utilisateurs du HPC. C'est pourquoi ces deux types de mémoire coexistent.

Sur de telles architectures, pour bénéficier des performances spécifiques de chacune des mémoires il faut réussir à répondre à la question : Sur quelle mémoire dois-je placer chacune des données de mon programme ? Ce problème peut être vu comme un problème de placement de données : dans ce cas une donnée une fois placée sur une mémoire peut-être déplacée vers une autre mémoire, selon l'utilisation qu'en fait l'application, ou bien comme un problème d'allocation de données : dans ce cas une fois qu'une donnée est allouée sur une mémoire, elle ne se déplace plus et elle utilise donc un certain espace mémoire jusqu'à sa libération. Nous considérons dans notre étude le problème d'allocation de données, nous réfléchirons dans les perspectives sur le moyen d'étendre notre étude au problème de placement de données.

6.1 Contributions

Dans ce mémoire nous avons cherché à formuler le problème d'allocation de données sur une AMH de sorte à faire ressortir le caractère temporel du problème. Sans ce caractère, il n'est pas possible de rendre compte de l'influence des performances de bande passante mémoire sur les performances du programme. Or, sur certaines AMH, la bande passante est une caractéristique cruciale pour différencier les mémoires et sélectionner une stratégie de placement de données performante. À notre connaissance, il n'existait pas de formulation

de ce problème d'optimisation tenant compte du temps d'exécution du programme et des régions temporelles du code.

À partir de cette formulation en un problème linéaire en nombres entiers, pour résoudre le problème d'allocation il nous faut évaluer les différents coefficients de la fonction objective. Sur une AMH possédant deux mémoires, l'évaluation de chacun des coefficients pourrait se faire en exécutant l'application autant de fois qu'elle possède d'objets de données : chaque exécution permet d'évaluer un coefficient différent en prenant soin d'allouer la donnée correspondante sur la mémoire possédant la plus petite capacité et le reste des données sur l'autre mémoire. Pour ne pas avoir à exécuter un si grand nombre de fois l'application, notre méthode, et toutes les méthodes de l'état de l'art, cherchent en fait à approximer ces coefficients. Plus les coefficients approximés ont une valeur proche de celle qui serait obtenue avec les mesures réelles, et meilleure sera la solution. Ces coefficients dépendent à priori des paramètres d'entrée de l'application et de tout ce qui constitue son environnement d'exécution. Il est possible de chercher à les approximer avec un profilage dynamique hors ligne, en ligne ou bien un profilage statique (compilation).

Nous avons décidé d'approximer ces coefficients avec un profilage dynamique hors ligne, et bien que ce ne soit pas le moyen le plus pratique pour l'utilisateur final, cette approche permet à priori une meilleure précision. Notre méthode est implémentée dans une boîte à outils codée en C,C++ et python et applicable sur des codes C et C++. Cette boîte à outils est composée :

— d'une bibliothèque dynamique, le Bandit de bande passante, exécutant un noyau de calcul dérobant le plus de bande passante mémoire possible à l'application profilée, sans influencer son utilisation de la mémoire cache. Cela permet de diminuer artificiellement la bande passante disponible afin de profiler la sensibilité des régions temporelles de l'application aux caractéristiques mémoires de latence et de bande passante ;
— d'une bibliothèque dynamique capturant les allocations dynamiques de l'application profilée et d'un plugin GCC permettant d'instrumenter les accès en lecture et en écriture des régions de code sélectionnnées afin de mettre en évidence les données accédées dans les régions sensibles à la bande passante ;
— d'un ensemble de scripts python analysant les informations extraites du profilage effectué avec les deux bibliothèques précédentes, et résolvant le problème linéaire avec une heuristique pour fournir en sortie l'ensemble des objets de données à allouer en MCDRAM ainsi que leur ligne de code caractéristique afin d'insérer des sondes informatives pour l'étape suivante ;
— d'une bibliothèque dynamique capturant les allocations dynamiques de l'application profiléeet les redirigeant sur une mémoire ou une autre en fonction des informations fournies par des sondes préalablement insérées dans le code à exécuter.

Pour fonctionner correctement, notre outil de profilage temporel doit être calibré pour la machine NUMA avec laquelle il est utilisé. Ce calibrage définit le seuil de ralentissement induit par le Bandit et différenciant une sensibilité à la latence d'une sensibilité à la bande passante pour l'application profilée. Tant que la machine utilisée pour un nouveau profilage reste identique, ce calibrage peut ne pas être renouvelé.

Une fois les coefficients évalués, plutôt que de les injecter dans la fonction objective et de chercher à réduire le problème avec toutes ses variables, on évalue directement les inconnues pour les coefficients dont on sait qu'ils ne favorisent pas une allocation sur la mémoire à faible capacité. Ainsi, on peut réduire le nombre de variables à prendre en compte lors de la résolution du problème linéaire. Cette méthode est approximative et pour obtenir une solution heuristique plus proche de l'optimale il faudrait tenir compte de tous les accès mémoires, en particulier ceux intervenant dans des régions temporelles non sélectionnées mais sur des données sélectionnées.

L'évaluation de notre méthode consiste à la comparer aux solutions publiques existantes. À travers cette évaluation, nous montrons qu'une stratégie d'allocations guidée par un profilage temporel puis spatial de l'application obtient des performances toujours proches des meilleures performances, là où les autres stratégies ne sont pas uniformément performantes sur les différentes applications testées. Nos résultats mettent aussi en évidence la limite d'une approche dont la granularité est l'objet de donnée alloué dynamiquement. En effet, sur certaines tailles de problème pour la mini-applications HydroMM notre méthode est moins performante. Pour ces tailles de problème, les objets de données alloués sont trop grands pour entrer en mémoire, et notre méthode ne permet de considérer uniquement des parties de ces objets, contrairement aux autres méthodes.

6.2 Perspectives

Ces travaux, en mettant en lumière l'importance de considérer toutes les caractéristiques mémoires avant de sélectionner une stratégie d'allocation de données, ouvrent de nouvelles perspectives à court, moyen et long termes.

L'avenir du *many-core*

Intel a annoncé au cours de SC2017 que la prochaine génération des Xeon Phi est abandonnée [Feldman,]. Bien que cette première expérience d'AMH sur CPU ne semble pas concluante, d'autres AMH existent et soulève la même problématique que celle que nous avons approfondie dans ce mémoire. De plus, les machines qui composeront les supercalculateurs exascale reposeront très certainement sur une architecture AMH [Zou et al., 2015, Vijayaraghavan et al., 2017].

Allocateur de données *AMH aware* intégré aux supports exécutifs

Il existe déjà des allocateurs *bank-aware* [Yun et al., 2014], et *NUMA-aware* [Ogasawara, 2009, Zhang et al., 2015, Williams et al., 2017, Aderholdt et al., 2018]. Par extension on pourrait imaginer ce que serait un allocateur *AMH aware*. Servat *et al.* [Alvarez et al., 2018] ont déjà développé un *memory manager* dans le support d'exécution déplaçant les données de la prochaine tâche à exécuter sur la mémoire possédant le plus de bande passante, mais cet allocateur ne prend pas en compte les différences des caractéristiques mémoires dans le choix du déplacement. K. Wu *et al.* [Wu et al., 2017a] ont développé un outil de déplacement des données entre les différentes mémoires d'une AMH mais agissant uniquement sur celles marquées par le programmeur. Leur méthode, basée sur un compte de LLCmiss mais mesuré sur des régions temporelles définies par les événements MPI

est très intéressante. Nous imaginons étendre ce concept à un support exécutif comme StarPU, avec donc la possibilité de gérer tout type d'AMH (KNL, DDR SDRAM + NVM, GPGPU + CPU) à partir d'un seul allocateur, avec une granularité du niveau des tâches définies par l'utilisateur.

StarPU [Augonnet et al., 2009] est un support exécutif unifié pour les architectures hétérogènes, composées d'unités de calcul hétérogènes et de mémoires hétérogènes. Il supporte l'odonnancement de tâches générées par l'application, et de nombreux travaux de recherche utilisant StarPU tendent à maximiser les performances en distribuant les tâches sur les ressources de calcul hétérogènes d'un supercalculateur. Certains de ces travaux cherchent à diminuer l'impact des transfers de données [Augonnet et al., 2010] entre CPUs, GPUs, et CPU-GPU : StarPU supporte les transferts de données asynchrones, il optimise la localité des données vis-à-vis des tâches y accédant, et il maximise le recouvrement des temps de transfert de données par du calcul. Ce qui manque néanmoins au support exécutif, c'est de tenir compte des caractéristiques de bande passante et latence des différentes mémoires du système, ainsi que des besoins de chacune des tâches de l'application en bande passante et latence mémoire. Pour cela, nous envisageons plusieurs pistes :

— suggestion du programmeur : sous forme de pragma le programmeur pourrait marquer certaines données comme étant sensible à la latence ou à la bande passante mémoire.
— analyse statique et dynamique : récupérer des informations par une analyse statique instrumentant le code et une analyse dynamique pour analyser la sensibilité des données de l'application aux caractéristiques mémoires.

Le support d'exécution peut alors tenir compte de ces informations dans la prise de décision des ressources mémoires à allouer à telle ou telle tâche, en quantité et en nature (DDR, MCDRAM, GDDR, NVM).

Synergie matérielle/logicielle

Comme nous l'avons introduit au début de ce mémoire, un des enjeux majeurs des acteurs du HPC est la réduction du temps d'exécution des applications scientifiques. Cela demande une grande synergie entre les trois piliers que sont le matériel, les logiciels de support exécutif et autres bibliothèques et les applications scientifiques elles-mêmes. Cette synergie se construit notamment par le développement de micro-benchmarks représentatifs du cœur des applications scientifiques. Ces micro-benchmarks sont utilisés par les fabricants pour innover de nouvelles architectures matérielles capables d'accélérer l'exécution des applications par comparaison aux architectures actuelles. La grande difficulté au niveau matériel est de développer des architectures génériques capables de satisfaire une gamme hétérogène de besoin applicatif. Une autre direction consiste, au contraire, à spécialiser autant que possible les architectures matérielles et logicielles aux besoins spécifiques de chaque application. Dans cet optique, notre méthode d'analyse pourrait être employée afin de mettre en évidence les applications nécessitant une architecture fournissant une meilleure bande passante, et les applications ne nécessitant pas de telles architectures. De plus, avec suffisamment d'exécution, le profilage au Bandit de bande passante pourrait donner une approximation des gains de performance envisageables ; toute proportion gardée, puisque d'une architecture à l'autre de nombreux paramètres, en plus de la bande

passante mémoire, varient.

Checkpoint/Restart

Le système d'une AMH se retrouve aussi sous une forme plus étendue dans les architectures des supercalculateurs proposant un mécanisme de Checkpoint/Restart automatique, qu'on nommera architecture CR. Chacun des niveaux de mémoires du mécanisme peut être considéré comme une mémoire composant une AMH. Sur ce type de système, les différences de caractéristique mémoire entre deux niveaux sont importantes, et des méthodes déjà éprouvées pour placer les données sur scratchpad pourraient être envisagées. Néanmoins, l'équivalence n'est pas évidente, parce que dans une architecture CR la capacité mémoire de chaque niveau est supérieure à celle d'une mémoire cache par plusieurs ordres de magnitude. Finalement, notre formulation du problème d'allocation de données s'applique à ce problème, mais la difficulté consiste à évaluer les coefficients de la fonction objective et ceux des contraintes sans nécessité un profilage très coûteux.

Appendices

Figures

.0HWLOC

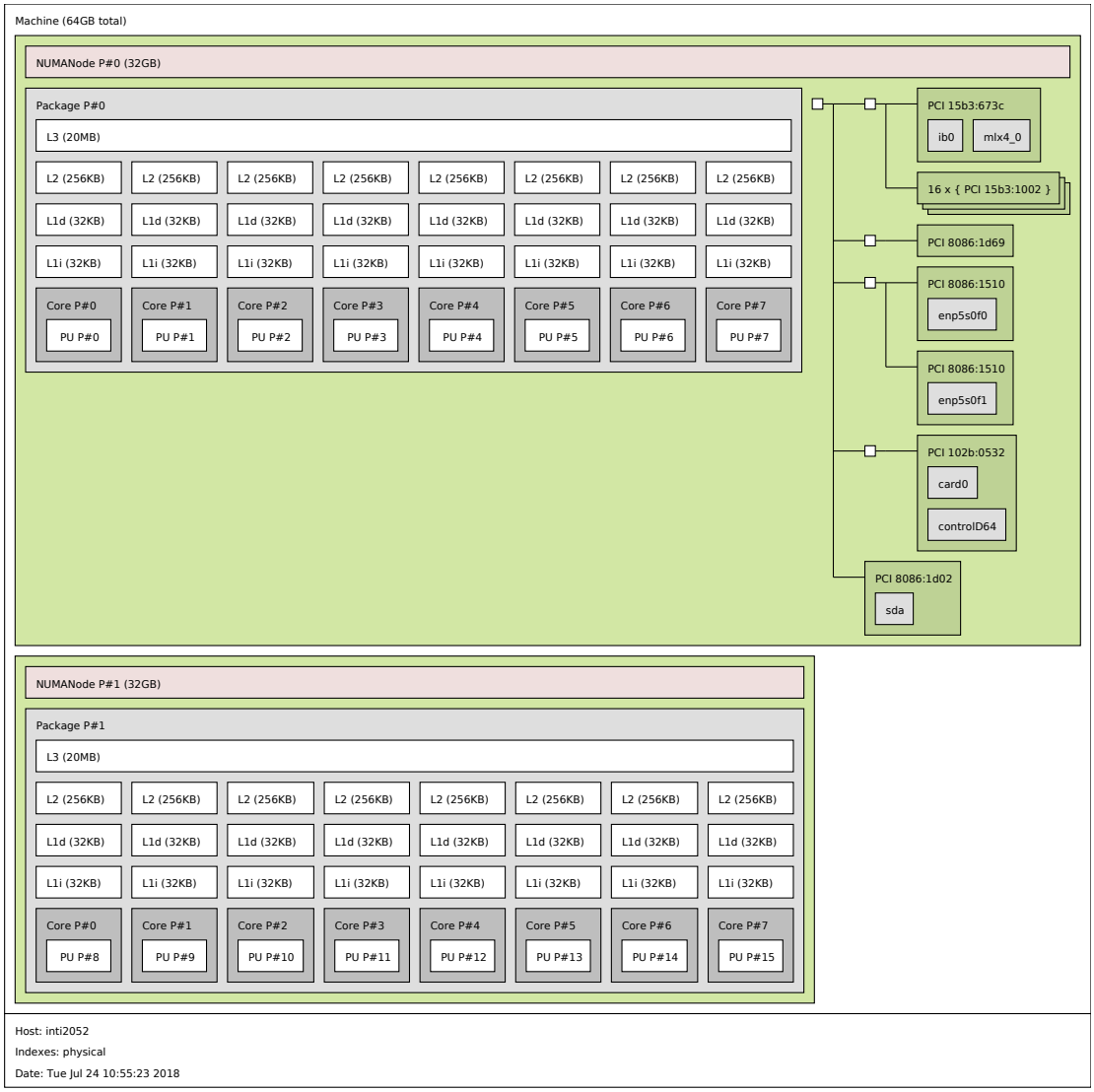


FIGURE .0.1 – lstopo de la machine Sandy avec HWLOC

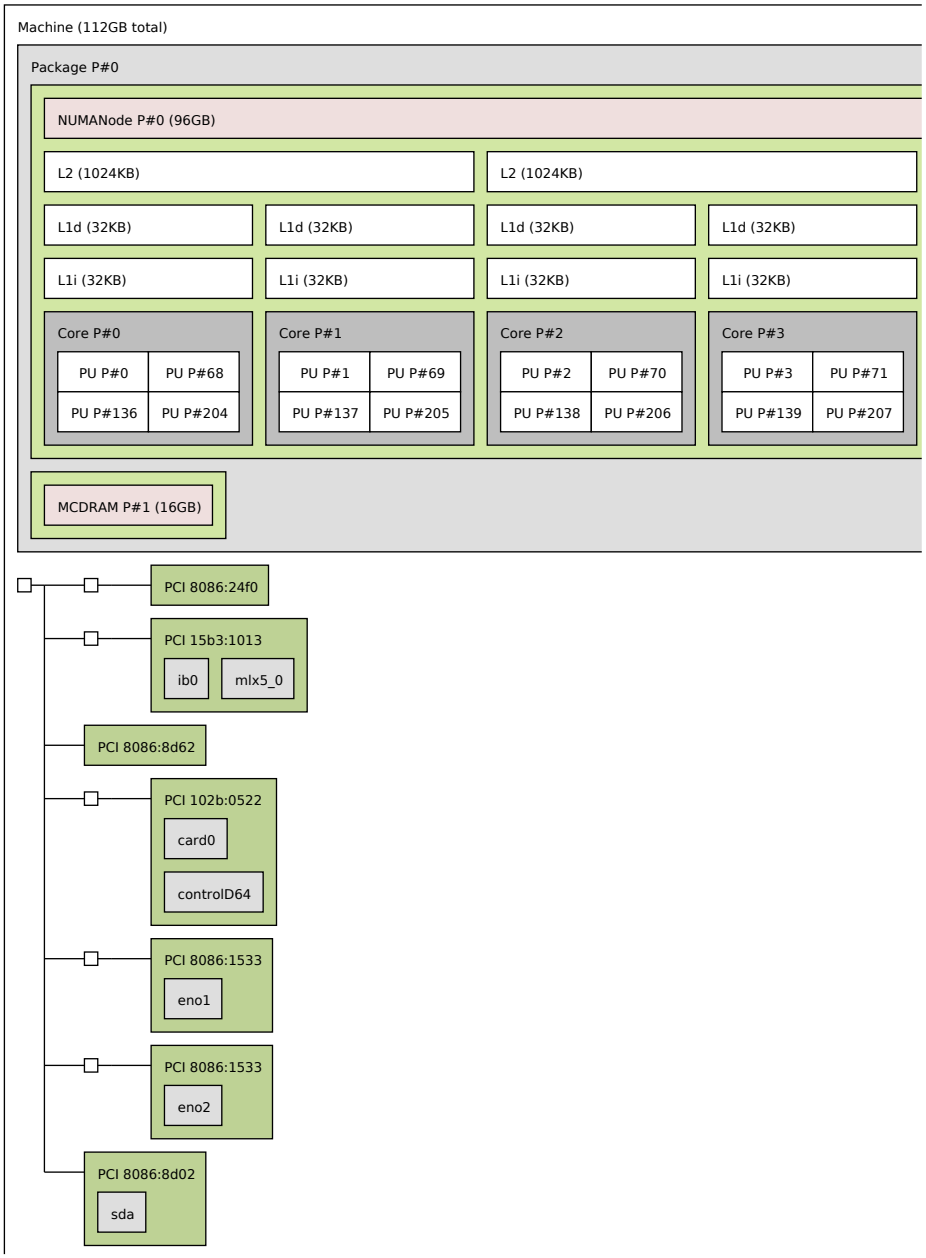


FIGURE .0.2 – KNL architecture

FIGURE .0.3 – KNL architecture

Acronymes

AMH Heterogeneous Memory Architecture : une Architecture à Mémoires Hétérogènes est une architecture matérielle composée d'un processeur, et parfois d'un GPGPU, et d'un ensemble de mémoires dont plusieurs mémoires principales possédants des caractéristiques de latence, bande passante, volatilité et capacité différentes. Ces mémoires peuvent utiliser des technologies très différentes : DDRX/DRAM classique, mémoire non-volatile (NVM), DRAM empilées, etc. 17, 20, 22–31, 34–38, 40, 41, 47, 49, 51, 58, 59, 67, 73, 91–95

API Application Programming Interface : Une Interface de Programmation est un ensemble de règles et de spécifications qu'un programme informatique peut suivre pour accéder et utiliser (faire appel) aux services et ressources fournies par un autre logiciel qui implémente l'interface. 67

APU Accelerated Processing Unit (APU) est une machine composée d'un ou plusieurs processeurs et d'un accélérateur GPGPU. Ils sont construit selon le "standard" HSA, donc la mémoire est unifiée. On peut imaginer d'autres type d'APU, ne suivant pas nécessairement le "standard" HSA, dont l'accélérateur ne serait pas un GPGPU mais un FPGA, un many-core (KNC), ou autre. 20

CPI Cycle Per Instruction (CPI) C'est le nombre de cycle par instruction. Il est calculé en mesurant le nombre de cycles total de l'application (temps * fréquence) et en divisant pas le nombre d'instructions contenues dans le code, qui correspond grossièrement au nombre de lignes de code assembleur. 11

CPU Central Processing Unit ou processeur. 1, 18–20, 103, 104

DDR SDRAM DDR SDRAM ou Double Data Rate Synchronous Dynamic Random Access Memory : Double Data Rate parce que les données sont transférées sur les front montant et descendant du signal électrique, contrairement à la version précédente de SDRAM. Les SDRAM sont des DRAM synchrones avec l'horloge du bus mémoire. C'est pourquoi la vitesse de ces mémoires est donnée en MHz et pas en nanosecondes. 18–20, 24, 26–30, 35, 36, 59, 94

DRAM Dynamic Random Access Memory (DRAM) : c'est un ensemble électronique de transistors et pico-condensateurs capable de stocker une donnée tant qu'elle est alimentée en courant électrique. Elle est asynchrone. 17, 18, 26, 27, 105

fetch C'est un déplacement de données depuis une mémoire cache ou principale vers le registre et les mémoires caches de niveau inférieurs. 11

FLOP (FLOP) Floating Point Operation : Operation flottante. 4, 6

FPGA Field-Programmable Gate Array : ce sont des circuits intégrés en silicium reprogrammables. Contrairement aux CPU, un FPGA peut être programmé pour redéfinir

le circuit intégré lui-même afin d'implémenter la fonctionnalité souhaitée. Il sera alors possible d'exécuter une application logicielle dessus. 1

GDDR GDDR ou Graphic DDR SDRAM est un type de SDRAM spécifiquement conçue pour les GPUs. 18, 19

GPGPU General Purpose Graphical Processing Unit (GPGPU) Ce sont des GPU qui sont détournés de leur fonction initiale (rendu graphique) pour être utilisé dans le calcul de noyaux d'application scientifique hautement parallèles. En effet, pour certains calculs parfaitement parallèle (comme les réseaux de neurones à convolution) ces puces permettent de réduire d'un ordre de grandeur les temps de calcul par comparaison des meilleurs CPU. 1, 20

GPU Graphical Processing Unit (GPU), Carte Graphique, est un circuit intégré assurant les fonctions de calcul de l'affichage. Un processeur graphique est généralement constitué d'une structure hautement parallèle qui le rend efficace sur des tâches graphiques qui effectuent une série d'opérations sur une grande quantité de donnée. 18, 19, 104

HBM HBM (High Bandwidth Memory) : est l'interface pour la mémoire empilée développée par AMD et Hynix. xiii, 18, 19

HMC HMC ou Hyper Memory Cube : c'est l'interface de la mémoire empilée développée par Micron Technology. xiii, 18, 19, 21, 76

HPC Calcul Haute Performance (HPC) C'est le domaine scientifique et industriel qui consiste à fabriquer et assembler des supercalculateurs et à développer des logiciels de support et des applications de simulation afin de résoudre des problèmes scientifiques complexes. 4, 7, 16, 19, 20, 23, 24, 29, 32, 43, 58, 91, 94

HSA HSA est un "standard" définit par un consortium dont AMD fait parti. Le but est de définir une norme pour l'unification de la mémoire entre GPGPU et CPU se partageant un interposeur commun et ainsi simplifié les transferts de données entre les 2. 20, 103

LLCmiss LLCmiss ou Last Level Cache miss sont les accès manqués au cache de dernier niveau mesurés par échantillonage à l'aide des compteurs matériels lorsqu'il sont disponibles. xi, 11, 27, 32, 33, 38, 56–58, 73, 74, 93

MCDRAM MCDRAM ou Multi-Channel Dynamic Random Access Memory : est un ensemble de DRAM assemblées en pile avec un très grand nombres de voies permettant d'augmenter considérablement la bande passante théorique disponible. 21, 22, 24, 26, 29, 31, 32, 58, 59, 67, 70, 71, 73, 76–78, 80, 81, 83, 84, 86, 87, 89–92

MIC (MIC) Many Integrated Core est un type d'architecture contenant de nombreux cœurs de calcul. Le nombre de cœurs a tendance à évoluer avec les technologies. 76

MLP Memory Level Parallism (MLP), c'est le niveau de parallélisme mémoire occupé par les accès mémoires de l'application. 14, 15

MRC Miss Ratio Curve (MRC) C'est la courbe représentant le nombre d'accès manqué à un niveau de cache en fonction de la taille du cache. 11

mémoire empilée Stacked-DRAM ou Mémoire empilée : c'est une mémoire issue des technologies d'empilement de DRAM qui possède une bande passante élevée grâce à la quantité importante de connecteurs. 18, 20, 21, 24, 26–30, 35, 91, 104

NUMA (NUMA) C'est une architecture composée de cœurs regroupés en groupes. Les cœurs au sein d'un même groupe sont plus proches que ceux de issus de groupes distincts. Les mémoires suivent le même principe, et un accès mémoire sur une mémoire locale est plus rapide car il ne nécessite pas l'utilisation d'un pont d'interconnexion. Les accès distants, que ce soit à la mémoire cache partagée ou principale d'un autre groupe de cœurs, doivent eux passer par ce pont d'interconnexion. Donc le temps de communication est plus long. Les performances de latence et de bande passante mémoire dépendent de la localité de la donnée accédée. xi, xiii, 7, 12, 13, 19–23, 37, 59–63, 65–67, 71, 92

NVM Non Volatile Memory (NVM) : C'est une mémoire non volatile. Par opposition à la DRAM, les données contenues dans une NVM ne sont pas perdues lorsqu'elle n'est plus alimentée en électricité. 20, 24, 28, 29, 35, 36, 73, 94, 105

OoO Out Of Order : c'est une technique matérielle qui consiste à re ordenancer les instructions au niveau des unités de calcul du processeur. 7

QPI Qui Path Interconnect (QPI) est la technologie Intel pour relier entre elles plusieurs sockets afin de constituer une machine multi-socket. 12, 61, 65, 77

RAM Random Access Memory (RAM) : c'est un ensemble électronique de transistors et pico-condensateurs capable de stocker une donnée tant qu'elle est alimentée en courant électrique. Elle est asynchrone. 17

SRAM Static Random Access Memory (SRAM) : c'est un ensemble électronique de transistors et pico-condensateurs capable de stocker une donnée tant qu'elle est alimentée en courant électrique. Elle est asynchrone. 17, 18, 26

stall stall ou "ne rien faire" : c'est lorsque le CPU ne calcul aucune instruction (ou instruction NOPE) parce qu'il est en attente d'une donnée. 33

TLB (Translation Lookaside Buffer Buffer contenant un cache des entrée de la table des pages. 29

UMA (UMA) C'est une architecture composée de cœurs de calcul et d'une mémoire partagée. Les performances de latence et de bande passante mémoire ne dépendent pas de la localité de la donnée accédée. Les perfomances sont donc uniforme sur l'ensemble de la machine. 12, 13, 21

Glossaire

exascale C'est une échelle qui correspond à un supercalculateur qui délivrerait 10^18 FLOPs par secondes. 4, 19, 24, 27, 28

Haswell C'est le nom de code de l'architecture Intel suivant le modèle Ivy Bridge. Le procédé utilise la technologie de 22 nm, son homologue est l'architecture Broadwell (14 nm). 61, 63, 64

hyperthreads hyperthread (nommenclature Intel) ou SMT (Simultaneous Multithreading) processus léger en français. Ces processus légers sont en tout point identiques à la définition d'un thread. La différence réside dans le fait que ceux là s'exécutent sur un même cœur de processeur : si le cœur possède suffisemment d'unité de calcul spécifiques et que les instructions des hyperthreads sont parallélisables, on peut observer un gain de performance ; dans le cas contraire, on observe une baisse de performance par rapport à une exécution séquentielle. 76, 84

KNL Le Knight's Landing est un processeur Xeon-Phi de type MIC Intel. Il est un exemple d'Architecture à Mémoires Hétérogènes.. xii, xiii, 20–22, 24, 26, 29–31, 35, 38, 58, 59, 67, 72, 75–81, 84–91, 94

MPI MPI ou Message Passing Interface est un standard de communication utilisée pour les envoie de message en HPC first. 93

Nehalem C'est le nom de code Intel pour la première génération d'Intel Core. C'est une architecture 45 nm. 15, 61, 63, 66, 77, 79

processus léger thread ou processus léger en français : c'est un ensemble d'instructions avec une pile qui lui est propre et un tas qui est partagé avec les autres processus légers appartenant au même processus maître. 15, 20, 22, 57, 60, 70

processus légers processus légers, pluriel de processus léger. xii, 23, 60–63, 65–67, 84–86

Sandy Bridge C'est le nom de code pour l'architecture suivant INTEL Nehalem, c'est donc la seconde génération des INTEL Core Processors. C'est une architecture 32 nm, elle sera suivie de son homologue en 22 nm (Ivy Bridge). xi, 12, 13, 16, 56, 61, 63–65, 77, 79

swap Mécanisme géré par le système d'exploitation permettant d'utiliser la mémoire d'un disque dur externe (ou carte SD ou autre) comme mémoire tampon lorsque la mémoire vive est entièrement utilisée. 28, 43

Bibliographie

[cor,] Corals. `https://asc.llnl.gov/CORAL-benchmarks/`. [Online ; accessed 26-august-2018].

[jem,] Jemalloc. `http://jemalloc.net/`. [Online ; accessed 26-june-2018].

[sta,] Le modèle standard. `https://home.cern/about/physics/standard-model`. [Online ; accessed 20-aug-2018].

[cer,] Serveur de données au cern en suisse. `https://home.cern/about/computing`. [Online ; accessed 20-aug-2018].

[spe,] Speccpu2006. `https://www.spec.org/cpu2006/`. [Online ; accessed 26-june-2018].

[Pav, 2008] (2008). *SRAM Cell Stability : Definition, Modeling and Testing*, pages 39–77. Springer Netherlands, Dordrecht.

[Aderholdt et al., 2018] Aderholdt, F., Venkata, M. G., and Parchman, Z. W. (2018). Sharp unified memory allocator : An intent-based memory allocator for extreme-scale systems. In *Euro-Par 2018 : Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 533–545.

[Advisory and Services, 2018] Advisory, I. F. and Services, E. I. B. A. (2018). *Financing the future of supercomputing : How to increase investments in high performance computing in Europe*.

[Agarwal et al., 2015a] Agarwal, N., Nellans, D., O'Connor, M., Keckler, S., and Wenisch, T. (2015a). Unlocking bandwidth for gpus in cc-numa systems. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 354–365.

[Agarwal et al., 2015b] Agarwal, N., Nellans, D., Stephenson, M., O'Connor, M., and Keckler, S. W. (2015b). Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 607–618, New York, NY, USA. ACM.

[Agawal and Gupta, 1988] Agawal, A. and Gupta, A. (1988). Memory-reference characteristics of multiprocessor applications under mach. *SIGMETRICS Perform. Eval. Rev.*, 16(1) :215–225.

[Aggarwal and Vitter, 1988] Aggarwal, A. and Vitter, Jeffrey, S. (1988). The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9) :1116–1127.

[Albericio et al., 2013] Albericio, J., Ibàñez, P., Viñals, V., and Llaberìa, J. M. (2013). The reuse cache : Downsizing the shared last-level cache. In *Proceedings of the 46th*

Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, pages 310–321, New York, NY, USA. ACM.

[Alvarez et al., 2018] Alvarez, L., Casas, M., Labarta, J., Ayguade, E., Valero, M., and Moreto, M. (2018). Runtime-guided management of stacked dram memories in task parallel programs. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 218–228, New York, NY, USA. ACM.

[(AMD),] (AMD), B. B. Die stacking is happening. MICRO 2013(02/12), keynote.

[Arge et al., 2008] Arge, L., Goodrich, M. T., Nelson, M., and Sitchinava, N. (2008). Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 197–206, New York, NY, USA. ACM.

[Augonnet et al., 2010] Augonnet, C., Clet-Ortega, J., Thibault, S., and Namyst, R. (2010). Data-Aware Task Scheduling on Multi-accelerator Based Platforms. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 291–298, Shanghai, China. IEEE.

[Augonnet et al., 2009] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2009). StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands. Springer.

[Bai and Shrivastava, 2010] Bai, K. and Shrivastava, A. (2010). Heap data management for limited local memory (llm) multi-core processors. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 317–326, New York, NY, USA. ACM.

[Bailey et al., 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The nas parallel benchmarks&mdash ;summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA. ACM.

[Bender et al., 2015] Bender, M. A., Berry, J. W., Hammond, S. D., Hemmert, K. S., McCauley, S., Moore, B., Moseley, B., Phillips, C. A., Resnick, D. S., and Rodrigues, A. (2015). Two-level main memory co-design : Multi-threaded algorithmic primitives, analysis, and simulation. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 835–846.

[Borkar, 2007] Borkar, S. (2007). Thousand core chips : A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA. ACM.

[Broquedis et al., 2010] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc : a Generic Framework for

Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia. IEEE Computer Society Press.

[Carr et al., 1994] Carr, S., McKinley, K. S., and Tseng, C.-W. (1994). *Compiler optimizations for improving data locality*, volume 29. ACM.

[Casas and Bronevetsky, 2014] Casas, M. and Bronevetsky, G. (2014). Active measurement of memory resource consumption. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 995–1004. IEEE Computer Society.

[Casas and Bronevetsky, 2016] Casas, M. and Bronevetsky, G. (2016). Evaluation of HPC applications' memory resource consumption via active measurement. *IEEE Trans. Parallel Distrib. Syst.*, 27(9) :2560–2573.

[Che et al., 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S. H., and Skadron, K. (2009). Rodinia : A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54.

[Che and Chatha, 2013] Che, W. and Chatha, K. S. (2013). Scheduling of synchronous data flow models onto scratchpad memory-based embedded processors. *ACM Trans. Embed. Comput. Syst.*, 13(1s) :30 :1–30 :25.

[Chen, 2016] Chen, A. (2016). A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125 :25 – 38. Extended papers selected from ESSDERC 2015.

[Chou et al., 2014] Chou, C., Jaleel, A., and Qureshi, M. K. (2014). Cameo : A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 1–12, Washington, DC, USA. IEEE Computer Society.

[Cicotti and Carrington, 2016] Cicotti, P. and Carrington, L. (2016). Adamant : Tools to capture, analyze, and manage data movement. *Procedia Computer Science*, 80 :450 – 460. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[Committee,] Committee, D. S. Dwarf. `http://dwarfstd.org/`. [Online ; accessed 26-october-2018].

[Corp., a] Corp., I. Auto hbw library. `https://github.com/memkind/memkind/tree/master/autohbw`. [Online ; accessed 26-june-2018].

[Corp., b] Corp., I. Memkind library. `http://memkind.github.io/memkind/`. [Online ; accessed 26-june-2018].

[Ding and Zhong, 2003] Ding, C. and Zhong, Y. (2003). Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 245–257, New York, NY, USA. ACM.

[Dongarra, 1988] Dongarra, J. (1988). The linpack benchmark : An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, UK. Springer-Verlag.

[Dongarra et al., 2015] Dongarra, J. J., Heroux, M. A., and Luszczek, P. (2015). Hpcg benchmark : a new metric for ranking high performance computing systems.

[Dongarra et al., 2003] Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The linpack benchmark : Past, present, and future. concurrency and computation : Practice and experience. *Concurrency and Computation : Practice and Experience*, 15 :2003.

[Egger et al., 2006] Egger, B., Kim, C., Jang, C., Nam, Y., Lee, J., and Min, S. L. (2006). A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 223–233, New York, NY, USA. ACM.

[Eichenberger et al., 2013] Eichenberger, A. E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., and Lorenz, D. (2013). Ompt : An openmp tools application programming interface for performance analysis. In Rendell, A. P., Chapman, B. M., and Müller, M. S., editors, *OpenMP in the Era of Low Power Devices and Accelerators*, pages 171–185, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Eklov et al., 2011] Eklov, D., Nikoleris, N., Black-Schaffer, D., and Hagersten, E. (2011). Cache pirating : Measuring the curse of the shared cache. In *2011 International Conference on Parallel Processing*, pages 165–175.

[Eklov et al., 2012] Eklov, D., Nikoleris, N., Black-Schaffer, D., and Hagersten, E. (2012). Bandwidth bandit : Understanding memory contention. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 116–117.

[Eklov et al., 2013] Eklov, D., Nikoleris, N., Black-Schaffer, D., and Hagersten, E. (2013). Bandwidth Bandit : Quantitative characterization of memory contention. In *CGO*, page 10. IEEE Computer Society.

[Feldman,] Feldman, M. Intel dumps knights hill, future of xeon phi product line uncertain. `https://www.top500.org/news/intel-dumps-knights-hill-future-of-xeon-phi-product-line-uncertain/`. [Online ; accessed 6-august-2018].

[Flynn, 1972] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9) :948–960.

[Gepner and Kowalik, 2006] Gepner, P. and Kowalik, M. F. (2006). Multi-core processors : New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13.

[Giménez et al., 2014] Giménez, A., Gamblin, T., Rountree, B., Bhatele, A., Jusufi, I., Bremer, P. T., and Hamann, B. (2014). Dissecting on-node memory access performance : A semantic approach. In *SC14 : International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 166–176.

[Goglin, 2014] Goglin, B. (2014). Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy.

[Goglin, 2016] Goglin, B. (2016). Exposing the locality of heterogeneous memory architectures to hpc applications. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 30–39, New York, NY, USA. ACM.

[Graham et al., 1982] Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof : A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA. ACM.

[Guo et al., 2011] Guo, Y., Zhuge, Q., Hu, J., Qiu, M., and Sha, E. H. M. (2011). Optimal data allocation for scratch-pad memory on embedded multi-core systems. In *2011 International Conference on Parallel Processing*, pages 464–471.

[Guo et al., 2013] Guo, Y., Zhuge, Q., Hu, J., Yi, J., Qiu, M., and Sha, E. H. M. (2013). Data placement and duplication for embedded multicore systems with scratch pad memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6) :809–817.

[Henning, 2006] Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4) :1–17.

[Heroux et al., 2009] Heroux, M. A., Doerfler, D. W., Crozier, P. S., Willenbring, J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W. (2009). Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories.

[Heroux and Dongarra, 2013] Heroux, M. A. and Dongarra, J. (2013). Toward a new metric for ranking high performance computing systems.

[Heroux et al., 2013] Heroux, M. A., Dongarra, J., and Luszczek, P. (2013). Hpcg benchmark technical specification.

[Hill,] Hill, M. D. Dinero iv trace-driven uniprocessor cache simulator. `www.cs.wisc.edu/~markhill/DineroIV`.

[Hundt, 2000] Hundt, R. (2000). Hp caliper : a framework for performance analysis tools. *IEEE Concurrency*, 8(4) :64–71.

[Idrissi Aouad and Zendra, 2007] Idrissi Aouad, M. and Zendra, O. (2007). A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy. In Zendra, O., Jul, E., and Cebulla, M., editors, *2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*, pages 31–38, Berlin, Germany. ECOOP. ICOOOLPS'2007 was co-located with the 21st European Conference on Object-Oriented Programming (ECOOP'2007).

[Ilic et al., 2014] Ilic, A., Pratas, F., and Sousa, L. (2014). Cache-aware roofline model : Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1) :21–24.

[Itzkowitz et al., 2003] Itzkowitz, M., Wylie, B. J. N., Aoki, C., and Kosche, N. (2003). Memory profiling using hardware counters. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 17–, New York, NY, USA. ACM.

[Jacob, 2009] Jacob, B. (2009). The memory system : you can't avoid it, you can't ignore it, you can't fake it. *Synthesis Lectures on Computer Architecture*, 4(1) :1–77.

[Janjusic and Kavi, 2013] Janjusic, T. and Kavi, K. (2013). Gleipnir : A memory profiling and tracing tool. *SIGARCH Comput. Archit. News*, 41(4) :8–12.

[Kågström et al., 1998] Kågström, B., Ling, P., and van Loan, C. (1998). Gemm-based level 3 blas : High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3) :268–302.

[Karlin et al., 2013] Karlin, I., Keasler, J., and Neely, R. (2013). Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973.

[Karrels and Lusk, 1994] Karrels, E. and Lusk, E. (1994). Performance analysis of mpi programs. *Environments and Tools for Parallel Scientific Computing*, pages 195–200.

[Khaldi and Chapman, 2016] Khaldi, D. and Chapman, B. M. (2016). Towards automatic HBM allocation using LLVM : A case study with knights landing. In *Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, November 14, 2016*.

[Kim et al., 2011] Kim, S., Kwon, K., Kim, C., Jang, C., Lee, J., and Min, S. L. (2011). Demand paging techniques for flash memory using compiler post-pass optimizations. *ACM Trans. Embed. Comput. Syst.*, 10(4) :40 :1–40 :29.

[Kowarschik and Weiß, 2003] Kowarschik, M. and Weiß, C. (2003). *An overview of cache optimization techniques and cache-aware numerical algorithms*. Springer.

[L. et al.,] L., A., S., B., M., F., M., K., G., M., J., M.-C., and R., T. N. Hpctoolkit : tools for performance analysis of optimized parallel programs. *Concurrency and Computation : Practice and Experience*, 22(6) :685–701.

[Laghari and Unat, 2017] Laghari, M. and Unat, D. (2017). Object placement for high bandwidth memory augmented with high capacity memory. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 129–136.

[Lam et al., 1991] Lam, M. D., Rothberg, E. E., and Wolf, M. E. (1991). The cache performance and optimizations of blocked algorithms. *SIGARCH Comput. Archit. News*, 19(2) :63–74.

[Leon and Hautreux, 2018] Leon, E. A. and Hautreux, M. (2018). Achieving transparency mapping parallel applications : A memory hierarchy affair. In *Proceedings on the 4th International Symposium on Memory Systems*.

[Levon and Elie, 2004] Levon, J. and Elie, P. (2004). Oprofile : A system profiler for linux.

[Lin and Liu, 2016] Lin, F. X. and Liu, X. (2016). Memif : Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, pages 369–383, New York, NY, USA. ACM.

[Loh, 2008] Loh, G. H. (2008). 3d-stacked memory architectures for multi-core processors. In *2008 International Symposium on Computer Architecture*, pages 453–464.

[Ludwig, 2012] Ludwig, T. (2012). The costs of hpc-based science in the exascale era. In *2012 SC Companion : High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*, pages 2120–2188. IEEE Computer Society.

[Martonosi et al., 1992] Martonosi, M., Gupta, A., and Anderson, T. (1992). Memspy : Analyzing memory system bottlenecks in programs. *SIGMETRICS Perform. Eval. Rev.*, 20(1) :1–12.

[Martonosi et al., 1995] Martonosi, M., Gupta, A., and Anderson, T. E. (1995). Tuning memory performance of sequential and parallel programs. *Computer*, 28(4) :32–40.

[McCalpin, 1995] McCalpin, J. (1995). Memory bandwidth and machine balance in high performance computers. pages 19–25.

[McCalpin, 2016] McCalpin, J. D. (2016). Memory latency on the intel xeon phi x200 "knights landing" processor. `http://sites.utexas.edu/jdm4372/2016/12/06/memory-latency-on-the-intel-xeon-phi-x200-knights-landing-processor/`. [Online ; accessed 26-june-2018].

[McVoy and Staelin, 1996] McVoy, L. and Staelin, C. (1996). Lmbench : Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA. USENIX Association.

[Meswani et al., 2015] Meswani, M., Blagodurov, S., Roberts, D., Slice, J., Ignatowski, M., and Loh, G. (2015). Heterogeneous memory architectures : A hw/sw approach for mixing die-stacked and off-package memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 126–136.

[Meswani et al., 2014] Meswani, M. R., Loh, G. H., Blagodurov, S., Roberts, D., Slice, J., and Ignatowski, M. (2014). Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 9–16, Piscataway, NJ, USA. IEEE Press.

[Mittal and Vetter, 2016a] Mittal, S. and Vetter, J. S. (2016a). A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5) :1537–1550.

[Mittal and Vetter, 2016b] Mittal, S. and Vetter, J. S. (2016b). A survey of techniques for architecting dram caches. *IEEE Trans. Parallel Distrib. Syst.*, 27(6) :1852–1863.

[Moore,] Moore, G. E. Cramming more components onto integrated circuits, electronics, vol. 38, no. 8, april 1965.

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). valgrind : a framework for heavyweight dynamic binary instrumentation. In *proceedings of the 28th acm sigplan conference on programming language design and implementation*, pldi '07, pages 89–100, new york, ny, usa. acm.

[Ogasawara, 2009] Ogasawara, T. (2009). Numa-aware memory manager with dominant-thread-based copying gc. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 377–390, New York, NY, USA. ACM.

[Pawlowski, 2011] Pawlowski, J. T. (2011). Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24.

[Pena and Balaji, 2015] Pena, A. J. and Balaji, P. (2015). Understanding data access patterns using object-differentiated memory profiling. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1143–1146.

[Peng et al., 2017] Peng, I. B., Gioiosa, R., Kestor, G., Cicotti, P., Laure, E., and Markidis, S. (2017). Rthms : A tool for data placement on hybrid memory system. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 82–91, New York, NY, USA. ACM.

[Pérache et al., 2008] Pérache, M., Jourdren, H., and Namyst, R. (2008). Mpc : A unified parallel runtime for clusters of numa machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 78–88, Berlin, Heidelberg. Springer-Verlag.

[Peña and Balaji, 2014a] Peña, A. J. and Balaji, P. (2014a). A framework for tracking memory accesses in scientific applications. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 235–244.

[Peña and Balaji, 2014b] Peña, A. J. and Balaji, P. (2014b). Toward the efficient use of multiple explicitly managed memory subsystems. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 123–131.

[Peña and Balaji, 2016] Peña, A. J. and Balaji, P. (2016). A data-oriented profiler to assist in data partitioning and distribution for heterogeneous memory in hpc. *Parallel Comput.*, 51(C) :46–55.

[Radulovic et al., 2015] Radulovic, M., Zivanovic, D., Ruiz, D., de Supinski, B. R., McKee, S. A., Rado vić, P., and Ayguadé, E. (2015). Another trip to the wall : How much will stacked dram benefit hpc ? In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 31–36, New York, NY, USA. ACM.

[Rivers et al., 1998] Rivers, J. A., Tam, E. S., Tyson, G. S., Davidson, E. S., and Farrens, M. (1998). Utilizing reuse information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 449–456, New York, NY, USA. ACM.

[Saini et al., 2014] Saini, S., Chang, J., and Jin, H. (2014). Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications.

In Jarvis, S. A., Wright, S. A., and Hammond, S. D., editors, *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, Lecture Notes in Computer Science, pages 25–51. Springer International Publishing.

[Servat et al., 2017] Servat, H., Pena, A. J., Llort, G., Mercadal, E., Hoppe, H. C., and Labarta, J. (2017). Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 126–136.

[Sewall and Colin de Verdière, 2015] Sewall, J. and Colin de Verdière, G. (2015). *From "Correct" to "Correct & Efficient"*.

[Seward and Nethercote, 2005] Seward, J. and Nethercote, N. (2005). Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA. USENIX Association.

[Shende and Malony, 2006] Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2) :287–311.

[Silicon, a] Silicon, O. Hbm. `https://www.open-silicon.com/high-bandwidth-memory-ip/`. [Online ; accessed 26-june-2018].

[Silicon, b] Silicon, O. Hmc. `https://www.open-silicon.com/open-silicon-ips/hmc/`. [Online ; accessed 26-june-2018].

[Singler et al., 2007] Singler, J., Sanders, P., and Putze, F. (2007). Mcstl : The multi-core standard template library. In Kermarrec, A.-M., Bougé, L., and Priol, T., editors, *Euro-Par 2007 Parallel Processing*, pages 682–694, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Sodani, 2015] Sodani, A. (2015). Knights landing (knl) : 2nd generation intel xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24.

[Sridharan, 1997] Sridharan, K. (1997). Vtune : Intel's visual tuning environment. In *Proceedings of USENIX-NT '97*.

[Strohmaier, 2006] Strohmaier, E. (2006). Top500 supercomputer. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA. ACM.

[Strohmaier et al., a] Strohmaier, E., Dongarra, J., Simon, H., (from 1993 until his death in 2014), M. M., and Meuer, H. green500. `https://www.top500.org/green500/`. [Online ; accessed 23-may-2018].

[Strohmaier et al., b] Strohmaier, E., Dongarra, J., Simon, H., (from 1993 until his death in 2014), M. M., and Meuer, H. top500. `https://www.top500.org/`. [Online ; accessed 23-may-2018].

[Thomadakis, 2011] Thomadakis, M. E. (2011). The architecture of the nehalem processor and nehalem-ep smp platforms.

[Udayakumaran and Barua, 2003] Udayakumaran, S. and Barua, R. (2003). Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In

Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '03, pages 276–286, New York, NY, USA. ACM.

[Udayakumaran et al., 2006] Udayakumaran, S., Dominguez, A., and Barua, R. (2006). Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2) :472–511.

[Vaumourin et al., 2016] Vaumourin, G., Guerre, A., Dombek, T., and Barthou, D. (2016). Specific read-only data management for memory system optimization. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 337–340.

[Vetter and Chambreau, 2005] Vetter, J. and Chambreau, C. (2005). mpip : Lightweight, scalable mpi profiling.

[Vijayaraghavan et al., 2017] Vijayaraghavan, T., Eckert, Y., Loh, G. H., Schulte, M. J., Ignatowski, M., Beckmann, B. M., Brantley, W. C., Greathouse, J. L., Huang, W., Karunanithi, A., Kayiran, O., Meswani, M., Paul, I., Poremba, M., Raasch, S., Reinhardt, S. K., Sadowski, G., and Sridharan, V. (2017). Design and analysis of an apu for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 85–96.

[Weidendorfer et al., 2004] Weidendorfer, J., Kowarschik, M., and Trinitis, C. (2004). a tool suite for simulation based analysis of memory access behavior. In Bubak, M., Van Albada, G. D., Sloot, P. M. A., and Dongarra, J., editors, *computational science - iccs 2004*, pages 440–447, berlin, heidelberg. springer berlin heidelberg.

[Williams et al., 2017] Williams, S., Ionkov, L., and Lang, M. (2017). Numa distance for heterogeneous memory. In *Proceedings of the Workshop on Memory Centric Programming for HPC*, MCHPC'17, pages 30–34, New York, NY, USA. ACM.

[Williams et al.,] Williams, S., Van Straalen, B., Ligocki, T., Oliker, L., Cordery, M., and Lo, L. Roofline performance model. `https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/`. [Online ; accessed 26-june-2018].

[Williams et al., 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline : An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4) :65–76.

[Wolf and Lam, 1991] Wolf, M. E. and Lam, M. S. (1991). A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA. ACM.

[Wu et al., 2017a] Wu, K., Huang, Y., and Li, D. (2017a). Unimem : Runtime data management on non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 58 :1–58 :14, New York, NY, USA. ACM.

[Wu et al., 2017b] Wu, K., Huang, Y., and Li, D. (2017b). Unimem : Runtime data management on non-volatile memory-based heterogeneous main memory. *CoRR*, abs/1705.00249.

[Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall : Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1) :20–24.

[Yun et al., 2014] Yun, H., Mancuso, R., Wu, Z. P., and Pellizzoni, R. (2014). PALLOC : DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 155–166. IEEE Computer Society.

[Zerbino and Birney, 2008] Zerbino, D. R. and Birney, E. (2008). Velvet : Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5) :821–829.

[Zhang et al., 2015] Zhang, K., Chen, R., and Chen, H. (2015). Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 183–193, New York, NY, USA. ACM.

[Zhuge et al., 2012] Zhuge, Q., Guo, Y., Hu, J., Tseng, W. C., Xue, C. J., and Sha, E. H. M. (2012). Minimizing access cost for multiple types of memory units in embedded systems through data allocation and scheduling. *IEEE Transactions on Signal Processing*, 60(6) :3253–3263.

[Ziabari et al., 2016] Ziabari, A. K., Sun, Y., Ma, Y., Schaa, D., Abellán, J. L., Ubal, R., Kim, J., Joshi, A., and Kaeli, D. (2016). UMH : A Hardware-Based Unified Memory Hierarchy for Systems with Multiple Discrete GPUs. *ACM Transactions on Architecture and Code Optimization*, 13(4) :1–25.

[Zou et al., 2015] Zou, Q., Poremba, M., He, R., Yang, W., Zhao, J., and Xie, Y. (2015). Heterogeneous architecture design with emerging 3d and non-volatile memory technologies. In *The 20th Asia and South Pacific Design Automation Conference*, pages 785–790.