De la localité logicielle à la localité matérielle sur les architectures à mémoire partagée, hétérogène et non-uniforme

Nicolas Denoyelle

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Nicolas Denoyelle**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

De la localité logicielle à la localité matérielle sur les architectures à mémoire partagée, hétérogène et non-uniforme

---

Date de soutenance : 05 Novembre 2018

Devant la commission d'examen composée de :

M. Emmanuel Jeannot	Directeur de recherche, Inria / LaBRI	Encadrant
M. Brice Goglin	Chargé de recherche, Inria / LaBRI	Directeur
M. Guillaume Papauré	Ingénieur de recherche, Atos Bull Technologies	Encadrant
M. Arnaud Legrand ..	Directeur de recherche, CNRS / LIG	Président du Jury
M. Patrick Carribault	Ingénieur de recherche, CEA DAM	Rapporteur

– 2018 –

Résumé La hiérarchie mémoire des serveurs de calcul est de plus en plus complexe. Les machines disposent de plusieurs niveaux de caches plus ou moins partagés et d'une mémoire distribuée. Plus récemment le paysage du Calcul Haute Performance (CHP) a vu apparaître des mémoires adressables embarquées dans le processeur ainsi que de nouvelles mémoires non-volatiles (périphérique mémoire sur le bus d'entrées sorties et prochainement de la mémoire non-volatile directement sur le bus mémoire). Cette hiérarchie est nécessaire pour espérer obtenir de bonnes performances de calcul, au prix d'une gestion minutieuse du placement des données et des tâches de calcul. Là où la gestion des caches était entièrement matérielle et masquée au développeur, le choix du placement des données dans telle ou telle zone de mémoire, plus ou moins rapide, volatile ou non, volumineuse ou non, est maintentant paramétrable logiciellement. Cette nouvelle flexibilité donne une grande liberté aux développeurs mais elle complexifie surtout leur travail quand il s'agit de choisir les stratégies d'allocation, de communication, de placement, *etc.* En effet, les caractéristiques des nombreux niveaux de hiérarchie impliqués varient significativement en vitesse, taille et fonctionnalités.

Dans cette thèse, co-encadrée entre Atos Bull Technologies et Inria Bordeaux – Sud-Ouest, nous détaillons la structure des plates-formes contemporaines et caractérisons la performance des accès à la mémoire selon plusieurs scénarios de localité des tâches de calcul et des données accédées. Nous expliquons comment la sémantique du langage de programmation impacte la localité des données dans la machine et donc la performance des applications. En collaboration avec le laboratoire INESC-ID de Lisbonne, nous proposons une extension au célèbre modèle Roofline pour exposer de manière intelligible les compromis de performance et de localité aux développeurs d'applications. Nous proposons par ailleurs un outil de synthèse de métriques de localité mettant en lien les évènements de performance de l'application et de la machine avec la topologie de cette dernière. Enfin, nous proposons une approche statistique pour sélectionner automatiquement la meilleure politique de placement des tâches de calcul sur les cœurs de la machine et des données sur les mémoires.

Mots-clés Calcul haute performance, modèles de plate-formes parallèles, modèles d'applications, apprentissage statistique, modèle de performance, mémoire partagée, mémoire hétérogène, in-package memory, roofline model, mesure de performance, compteurs matériels, instrumentation de code

Laboratoire d'accueil / Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Title From Software Locality to Hardware Locality in Shared Memory Systems with NUMA and Heterogenous Memory.

Abstract Through years, the complexity of High Performance Computing (HPC) systems' memory hierarchy has increased. Nowadays, large scale machines typically embed several levels of caches and a distributed memory. Recently, on-chip memories and non-volatile PCIe based flash have entered the HPC landscape. This memory architecture is a necessary pain to obtain high performance, but at the cost of a thorough task and data placement. Hardware managed caches used to hide the tedious locality optimizations. Now, data locality, in local or remote memories, in fast or slow memory, in volatile or non-volatile memory, with small or wide capacity, is entirely software manageable. This extra flexibility grants more freedom to application designers but with the drawback of making their work more complex and expensive. Indeed, when managing tasks and data placement, one has to account for several complex trade-offs between memory performance, size and features.

This thesis has been supervised between Atos Bull Technologies and Inria Bordeaux – Sud-Ouest. In the hereby document, we detail contemporary HPC systems and characterize machines performance for several locality scenarios. We explain how the programming language semantics affects data locality in the hardware, and thus applications performance. Through a joint work with the INESC-ID laboratory in Lisbon, we propose an insightful extension to the famous Roofline performance model in order to provide locality hints and improve applications performance. We also present a modeling framework to map platform and application performance events to the hardware topology, in order to extract synthetic locality metrics. Finally, we propose an automatic locality policy selector, on top of machine learning algorithms, to easily improve applications tasks and data placement.

Keywords High Performance Computing, parallel platforms modeling, applications modeling, machine learning, performance modeling, shared memory, heterogeneous memory, in-package memory, roofline model, performance monitoring, hardware counters, binary instrumentation

Laboratoire d'accueil / Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Remerciements

L'orientation est une succession de choix parmi plusieurs possibilités. Il est rare de pouvoir comparer à posteriori la qualité d'un choix car on ne peut faire l'expérience que d'un seul. En revanche, ce dont je suis sûr, c'est de l'épanouissement dont j'ai bénéficié pendant ce doctorat, et des opportunités qui me sont offertes après celle-ci. Cette expérience, je la doit à toute les personnes bienveillantes qui ont gravité dans ma sphère durant ces trois années et plus.

Mon premier remerciement va à Emmanuel Jeannot, directeur de recherche dans l'équipe qui m'a accueillie. Je lui doit toute mon orientation post-diplôme d'ingénieur de mon stage de master, à mon doctorat, et jusqu'à mon orientation post-doctorat, grâce à son réseau dont j'ai bénéficié. Je souhaite remercier de manière équivalente Brice Goglin et Guillaume Papauré, qui m'ont également encadrés durant ces trois dernières années. De ces trois personnes, j'ai apprécié l'excellence tant scientifique que technique, qui a conféré sa valeur à ma formation et à ma production. J'ai également apprécié la justesse de leur encadrement, m'allouant la liberté nécessaire pour exprimer ma créativité, tout en étant présents aux bons moments pour la canaliser. D'autre part, ne faisant pas de partition stricte entre mon travail et ma vie privée, j'ai apprécié la relation extra professionnelle que j'ai pu avoir avec chacun. J'ai la conviction que cette dernière a eu pour effet une multitude de retombées positives sur mon travail tout au long de ces années.

Mes seconds remerciements vont à mes collègues thésards, ingénieurs, post-doc et chercheurs pour le partage de leur expérience et pour leur soutien. En particulier je remercie Marc Sergent qui m'a hébergé et assisté lors de plusieurs de mes déménagements ; Guillaume Lepoutère pour mon état des lieux de sorties Grenoblois ; Piotr Lesznicki, Mario Dagrada et Emmanuel Brelle pour leur aide précieuse à mes déménagements et les sorties ski ; François Tessier qui m'a précédé en doctorat et post-doctorat, en me transférant son expérience.

Il y a une contribution indirecte, néanmoins sensible, de l'entourage dans une réussite. Ainsi, mes troisièmes remerciements vont à mes amis les plus proches et à ma famille. Alban Demartin et Jérôme Lebot, mes camarades de promotion, qui m'ont aussi hébergé à Bordeaux ; Gary Peverelli ami sportif et mes parents. Grâce à eux et à leur soutien inconditionnel, ces années ponctués pour chacun de bon moments comme de difficultés, se sont déroulées avec fluidité et sereinité.

Je souhaite également remercier les chercheurs avec qui j'ai collaboré et qui ont façonné mes recherches vers des contributions fructueuses. Je pense en particulier à Aleksandar Ilic et Leonel Sousa à l'université INESC-ID de Lisbonne et Thomas Ropars aujourd'hui au LIG de Grenoble.

Enfin, je ne peux m'empêcher d'avoir au fond du coeur une pensée pour celle qui m'a accompagnée une majeure partie de mon doctorat.

Table des matières

Table des figures

Liste des tableaux

—

Chapitre 1

Introduction, contexte et motivations

1.1 Le calcul intensif au service de la science et de l'industrie

À la date d'écriture de ce manuscrit, le supercalculateur français Tera-1000-2, construit par le groupe Atos est actuellement 14ème [114] au classement Top500 [113] des calculateur les plus puissants au monde. Cette machine composée de 561 408 cœurs de calcul a enregistré un puissance de calcul de 11,97 PFlops/s (1 PFlops/s = 10^{15} opérations flottantes par secondes), pour une consommation de 3 178 kW (soit environ 2 millièmes de la production d'énergie d'une centrale nucléaire). Le banc d'essai [37] utilisé pour ce classement résout des problèmes mathématiques d'algèbre linéaire. Les briques de base de cette bibliothèque sont exploitées sur des machines de calcul similaires à Tera-1000-2 pour modéliser le comportements des fluides [72] et optimiser des structures et profils dans l'aéronautique, l'aérospatial, l'automobile, *etc.* De même ces machines sont utilisées pour simuler les interactions moléculaires [50] entre autres à des fin de recherches de traitements médicaux [91] ; pour l'étude des ondes électromagnétiques [116] pour optimiser la furtivité des avions ; à des fins de prévisions météorologiques et des risques naturels [96], *etc.*

Les applications de calcul sont fondées sur un modèle commun...

Le modèle fondateur d'architecture de Von Neumann [118] est à la base des architectures de micro-processeurs contemporains, et de leur jeu d'instructions. Les langages de programmation utilisés en Calcul Haute Performance dérivent de la manière de commander le matériel et impliquent que les problèmes résolus dans les supercalculateurs font essentiellement de la lecture et de l'écriture

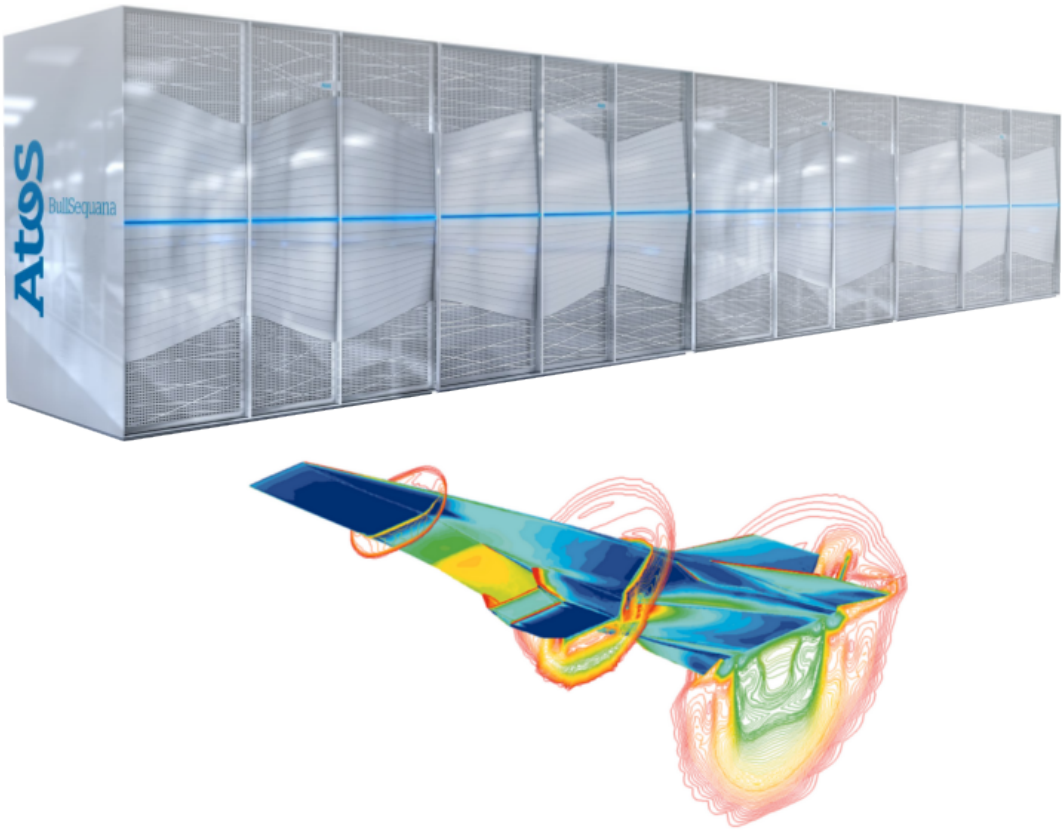


FIGURE 1.1 – Visuel d'un système de calcul haute performance et d'une application de dynamique des fluides computationelle (CFD).
Sources : Bull Sequana x1000, Hyper-X Mach 7

de données et des calculs, ce qui fait de ces derniers les principaux goulets d'étranglement des applications en général [121, 123, 66]. Cependant, évaluer la performance des systèmes de calcul pour la résolution de problèmes variés est une tâche difficile car les goulets d'étranglements se manifestent à une granularité beaucoup plus fine, dans des sous-systèmes nombreux et complexes.

...mais ont des contraintes propres

On peut actuellement identifier plusieurs classes applications, pour évaluer ou modéliser les machine de CHP. Plusieurs projets maintiennent des codes de mini-applications représentatives d'une partie de l'écosystème des applications utilisées dans le CHP. Parmi celles-ci on trouve : Simulation Numérique Aéronautique (*Numerical Aerodynamic Simulation*) (NAS) [5], *Princeton Application Repository for Shared-Memory Computers* (PARSEC) applications [8], SpecCPU *benchmarks* [51], les Dwarfs [2], Coral *benchmarks* [19]. Ces applications diffèrent par leurs objectifs, mais également par les briques logicielles

qu'elles exploitent, selon qu'elles manipulent des structures de données denses ou creuses, par les techniques numériques utilisées, le type de parallélisme mis en œuvre, et les limitations qu'elles exhibent. Certains de ces groupements d'applications sont à destination des concepteurs de systèmes de CHP car elles sont représentatives des problèmes calculatoires qui intéressent les clients de ce genre de plate-forme, et car elles imposent des contraintes variées qui challengent les constructeurs à fabriquer des systèmes performants sur une grande variété de problèmes. Parmi les bancs d'essais de systèmes de calcul, on distingue un particulier le *benchmark*[1] LINPACK [37] qui est utilisé pour classer les plates-formes au classement Top500 des calculateurs les plus puissants du monde. Ce banc d'essai est composé de briques de base de solutions à des problèmes d'algèbre linéaire dense. Cependant, ce modèle de classement est contesté, en particulier car les applications du banc de tests ne sont pas représentatives de toutes les contraintes généralement imposées à ces plates-formes. On voit donc émerger d'autres classements comme le HPCG [56] ou le Graph 500 [84] qui proposent de classer les systèmes de calcul haute performance selon d'autres critères.

Les systèmes de calcul haute performance sont généralistes

Il est rare qu'une application justifie seule, financièrement, le développement de matériel spécialisé (*e.g.* accélérateur pour l'apprentissage statistique, ASIC de minage de crypto-jetons, ou le corrélateur du Grand réseau d'antennes millimétrique/submillimétrique de l'Atacama). Ainsi, la plupart des applications et calculateurs utilisent du matériel généraliste, et c'est la pile logicielle qui adapte l'utilisation du matériel à l'application. Les machines de calcul pour l'exascale intègreront des millions d'unités de calcul et seront capable d'effectuer de l'ordre de 10^{18} opérations arithmétiques par seconde. Un tel investissement mutualisera des efforts de développement pour adapter l'utilisation d'une machine puissante et généraliste mais complexe, à des cas particuliers.

1.2 La performance et la généralité au prix de la complexité

Les supercalculateurs sont de plus en plus performants

La croissance exponentielle de la puissance de calcul des supercalculateurs (*cf.* Figure 1.2), permet de traiter des problèmes de plus grande ampleur (*e.g.* modélisation d'une portion plus grande de la coupe d'aile d'un avion) ou de plus grande précision (*e.g.* découpage plus fin de l'espace autour du fuselage d'un avion). Cette capacité de calcul est obtenue par l'aggrégation hiérarchique

1. banc-d'essai

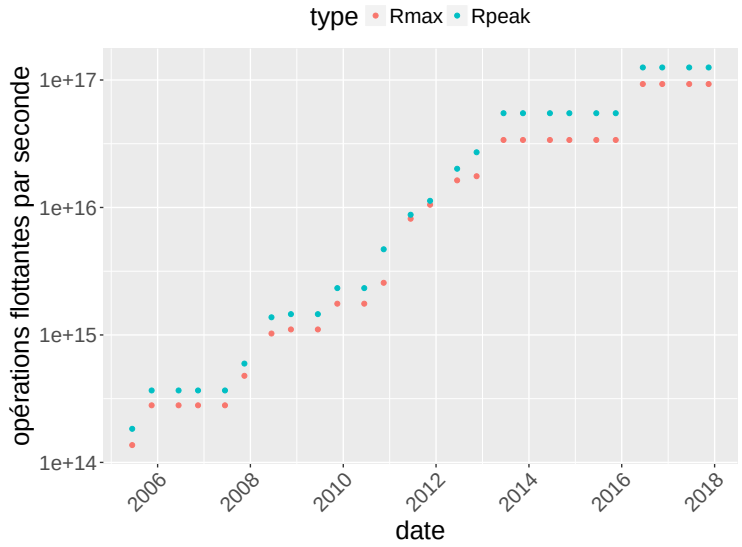


FIGURE 1.2 – Évolution de la puissance de calcul théorique (Rpeak) et mesurée (Rmax) sur un banc d'essai d'algèbre linéaire (linpack [37]), de la machines la plus puissante du classement Top500 [113] par date de parution du classement.

de ressources de calcul, depuis l'intégration de plusieurs unités de calcul dans les cœurs de calcul, de plusieurs cœurs sur la puce des processeurs, de plusieurs processeurs sur une carte mère (ou nœud de calcul), de plusieurs cartes mères dans des cabinets (ou armoires) et la mise en réseau de ces armoires en supercalculateur (illustré en Figure 1.3). Afin de soutenir cette tendance, il faut pallier à diverses barrières technologiques dans certains composants du système (le mur de la mémoire [123], limitation de la fréquence d'horloge [66]), et pallier la voracité énergétique qui croît aussi de manière exponentielle [117]. Par ailleurs, un rapport sur l'exascale [105] pointe les limites à la croissance actuelle des systèmes de calcul qu'il faudra nécessairement relever pour espérer atteindre l'échelle exascale, qui n'est pas accessible avec le matériel actuel. En particulier, la performance des accès à la mémoire et au stockage devra combler son retard sur la consommation/production des données au sein des unités de calcul.

La croissance de la taille des systèmes de calcul implique des problèmes de localité

La limite de fréquence des processeurs étant atteinte, l'augmentation de la puissance de calcul est désormais sujette à l'augmentation de la densité d'unités de calcul dans les machines et donc de leur parallélisme selon la loi de Moore [75]. Pour résoudre un problème de grande taille, on le découpe en sous-problèmes de plus petites taille, *i.e.* en tâches exécutables en parallèle

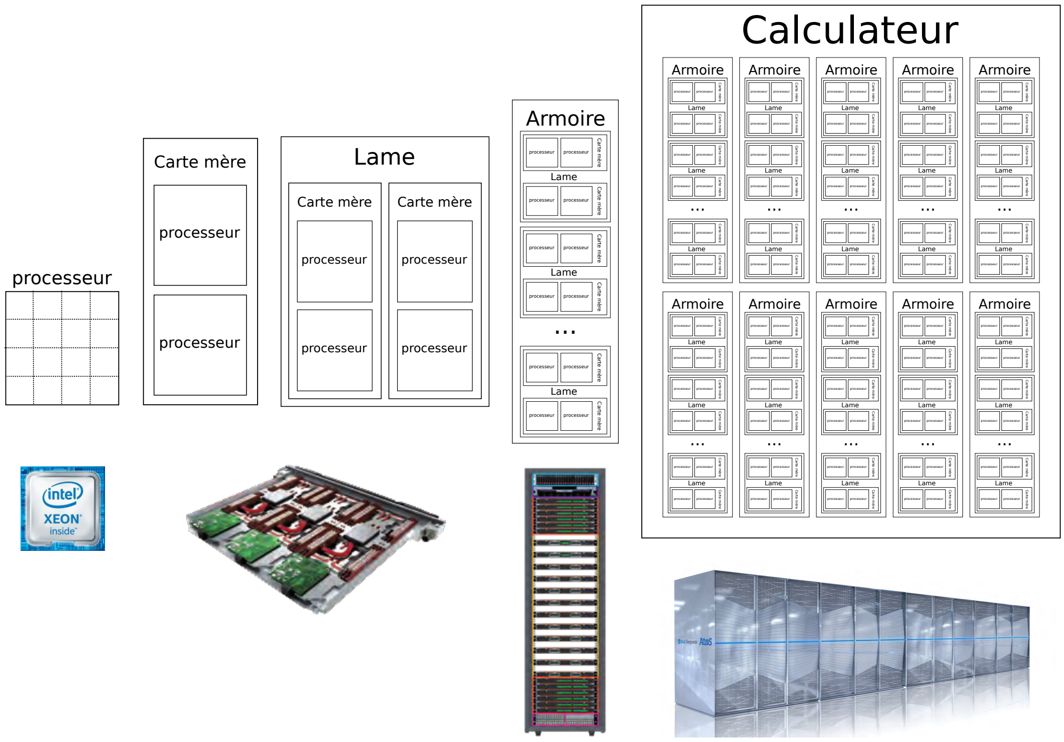


FIGURE 1.3 – Aggrégation des ressources de calcul dans un supercalculateur.

(on parallélise l'application) sur les unités de calcul. Il y a plusieurs limites au parallélisme des applications. D'abord une application n'est pas parallélisable en totalité. Elle est constituée au moins d'une partie parallèle et d'une partie séquentielle. Par conséquent, le gain de performance atteignable est borné selon la loi d'Amdahl [53]. Dans un programme découpé en tâches, la partie séquentielle peut être observée comme le chemin critique dans le graphe de dépendance entre les tâches. Les dépendances entre tâches témoignent de la nécessité d'échanger leurs résultats pour exécuter la tâche suivante. Donc, dans une exécution parallèle, il est nécessaire de déplacer des données entre les tâches. Selon la politique d'ordonnancement des tâches sur les cœurs [40], la distance entre les tâches qui échangent des données varie. Intuitivement, plus la machine est grande plus le partage des données se fait sur une grande distance et une plus grande durée. Ceci implique une notion de localité des données. Les applications sont caractérisées différemment selon la quantité et la qualité de leurs accès aux données. Les nombreuses couches de mémoires de ces systèmes tentent de répondre aux compromis qui caractérisent les applications et leur schéma d'accès et de partage des données.

La localité et la performance des accès à la mémoire en général sont mitigées par la complexité de l'architecture des systèmes de CHP

En Figure 1.4 est comparé le classement des meilleures machines au monde, selon qu'elles sont classées sur un banc d'essai d'algèbre linéaire dense orienté calcul (classement Top500) ou bien avec de l'algèbre linéaire creuse orientée accès mémoire (classement HPCG500). On voit la tendance d'accroissement de la performance des meilleures machines que ce soit en terme d'accès mémoire ou de performance de calcul. Cependant, il n'y a pas de correspondance entre les machines de chacun de ces groupes, donc l'augmentation de la performance de calcul n'est pas synonyme d'accélération des accès mémoire. On voit que selon les besoins, la performance d'un système change, et donc son architecture est de nature à influencer la performance. En particulier, les accès mémoire sont plus limitants que la capacité de calcul pour certaines applications, et le fossé entre le temps d'accès à la mémoire et au stockage, et la capacité de calcul tend à se creuser [105]

La mémoire est organisée hiérarchiquement pour répondre aux différents compromis de coût, de capacité et de performance requis par les applications. Cette hiérarchie mémoire n'est que partiellement administrable logiciellement bien que toutes les parties de celles-ci participent à la performance des applications. À proximité des cœurs, l'espace sur la puce du processeur est précieux et en quantité limitée tandis qu'en s'éloignant le périmètre disponible pour la mémoire s'agrandit alors que la distance rend les accès plus lents. Par conséquent, la mémoire d'un système de calcul est constituée de peu de mémoire Static Random Access Memory (RAM) (SRAM) chère et rapide sur la puce du processeur tandis que la carte mère est équipé de mémoire Dynamic RAM (DRAM) pouvant parfois s'étendre jusque dans les commutateurs du réseau, puis la périphérie du système est équipée de Solid Slate Drive (SSD) localement en tant que périphérique d'un nœud de calcul ou à distance dans des agrégateurs d'ES, avant de faire place à la technologie la moins coûteuse en terme de d'euro par octet, les Disque Durs (DDs) qui servent à stocker les informations qui doivent persister à l'extinction de la machine. Les disques durs sont présents sur les nœuds de calcul, et dans des baies de stockage dédiées au système de calcul complet, plus loin encore des calculs. Il existe même des entrepôts de stockage sur bandes pour l'archivage, où les bandes sont déplacées par des machines pour leur maintenance et la gestion physique de la localité des données.

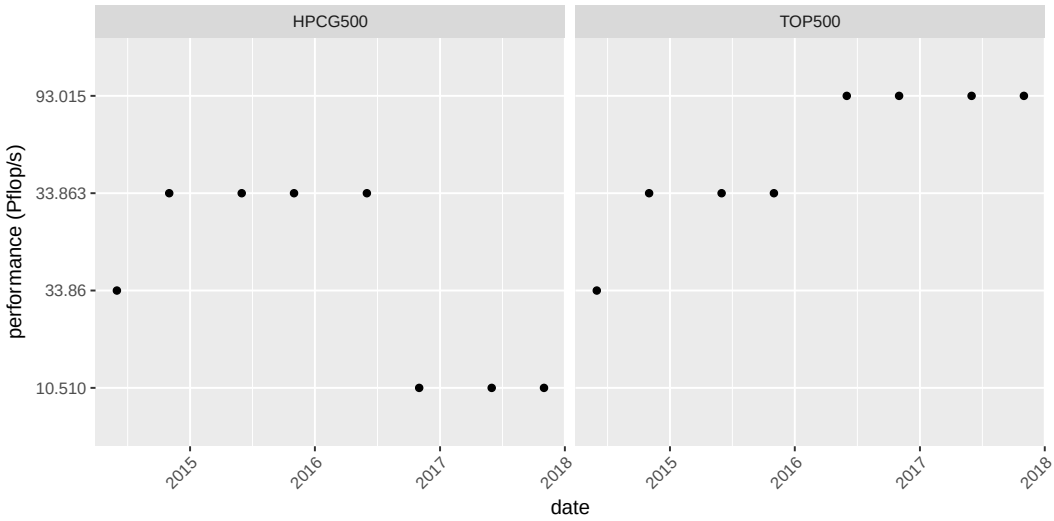
Il peut être nécessaire d'optimiser la gestion de la mémoire au niveau de l'application pour toutes les couches de mémoire. Il devient alors très vite intéressant de mutualiser ces efforts d'optimisation et de les abstraire/cacher au maximum pour réduire le coût d'optimisation des applications. Cette complexité est largement prise en charge dans différentes couches logicielles : lan-

gage, compilateur, support d'exécution, système d'exploitation, bibliothèque de communication, *etc.* Cependant, il reste encore beaucoup de défis d'optimisation dans l'utilisation de la hiérarchie mémoire et l'abstraction de la complexité des systèmes de calcul.

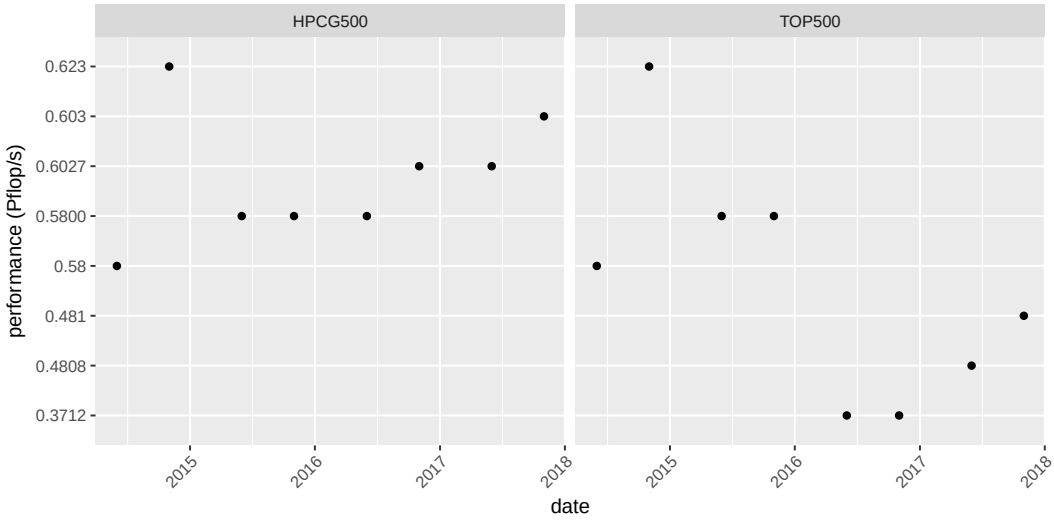
1.3 Organisation du document

Améliorer la localité des données consiste à mettre en place des mécanismes qui placent les données dans les mémoires proches de là où elles vont être utilisés où les calculs plus près des données au moment opportun. La distance entre la donnée et son utilisateur ainsi réduite permet également de diminuer les temps d'accès et d'améliorer les performances. Face à la croissance de la complexité des machines, la localité s'impose comme une nécessité pour espérer exploiter au mieux les systèmes de calcul intensifs. Dès lors les questions suivantes se posent : Comment améliore-t-on la localité ? Comment modélise-t-on la performance ? Comment exposons-nous les mécanismes à l'utilisateur ? Ce document traite ces questions de manière détaillée, et s'articule de la manière suivante :

- Le Chapitre 2 détaille la structure du matériel et de la mémoire dans les nœuds de calcul haute performance, ainsi que les compromis de performance en leur sain en fonction de la localité des données.
- Le Chapitre 3 décrit l'expression de la localité dans le logiciel et ses conséquences sur la position des données dans le matériel. De plus, ce chapitre dresse l'état l'art des solutions pour optimiser la localité des données, et énonce les problèmes ouverts traités dans cette thèse.
- Le Chapitre 4 expose plusieurs contributions pour modéliser les performances des machines et des applications par rapport à la localité des données. En particulier, nous étendons un modèle de représentation des bornes supérieurs (*roofline*) de performance de la machine, et des conséquences sur la performance des applications.
- Le Chapitre 5 présente une méthodologie et un outil associé pour construire des modèles de localité mettant en relation les évènements matériels et logiciels avec la structure du système de calcul.
- Le Chapitre 6 expose une étude de l'impact du placement des tâches des applications et de leur données, sur la localité, et propose une méthode statistique pour sélectionner automatiquement la meilleur politique de placement.



(a) Meilleur machine du HPCG500 (à gauche) et Top500 (à droite) au banc d'essai Linpack



(b) Meilleur machine du HPCG500 (à gauche) et Top500 (à droite) au banc d'essai HPCG

FIGURE 1.4 – Comparaison de la meilleur machine par période de parution du classement, selon le classement HPCG500 [56] (partie gauche) ou du classement Top500 [38] (partie droite) avec le même banc d'essai.

Chapitre 2

Hiérarchie mémoire dans les machines parallèles

Les systèmes de CHP sont constitués de plusieurs niveaux de mémoire, des registres du processeur à la mémoire distribuée. Les systèmes de calcul sont composés de plusieurs nœuds de calcul chacun avec une mémoire locale accessible par les autres nœuds. L'espace mémoire global de la machine est accessible depuis n'importe quel cœur et est présenté comme un espace d'adresses plat. Cependant, celui-ci est hétérogène en capacité, performance, accessibilité, volatilité, et sa structure impacte la performance des applications. Dans ce chapitre nous passons en revue les différentes mémoires des systèmes de calcul présents et futurs, de l'échelle la plus fine à l'échelle la plus large. Enfin nous caractérisons les différents compromis de performance des accès à la mémoire qui opèrent au sein du système.

L'optimisation des applications pour les systèmes de calcul haute performance nécessite une connaissance profonde de la machine, à toutes les couches mémoire, depuis les registres dans les cœurs de calcul jusqu'aux baies de stockage non-volatiles. Certains détails de fonctionnement et de performance de la mémoire peuvent être trouvés dans la littérature [110, 39, 106]. Certains sont repris dans ce chapitre, et étendus aux machines modernes, en perspective avec le problème de la localité.

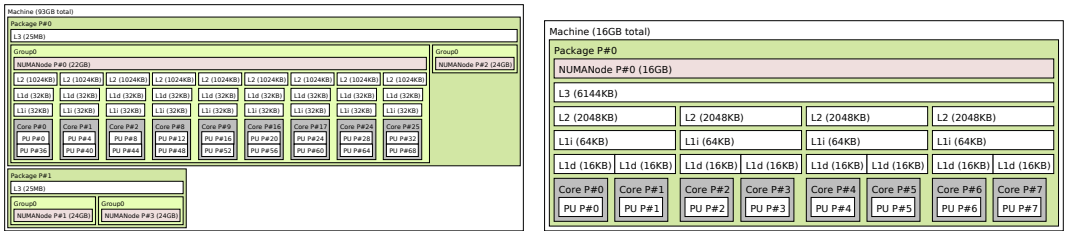
Le terme « mémoire » est utilisé conjointement pour désigner la mémoire adressable par octet (*i.e.* la mémoire que l'on peut explicitement allouer, libérer, lire et écrire, par opposition à la mémoire « cache » gérée de manière transparente par le matériel), ou bien le lieu d'existence de la donnée (*i.e.* un cache, un registre, une mémoire adressable, une mémoire non-volatile, *etc.*).

2.1 Composants de la hiérarchie mémoire

2.1.1 Registres du processeur

Le code d'une application est traduit en langage machine (assembleur, micro-code) dont les instructions sont encodées en une suite binaire par un compilateur, avant d'être exécutées par le processeur. Elles sont commandées par les cœurs et sont empilées dans un pipeline. Les instructions sont successivement chargées depuis le cache d'instructions, décodée puis exécutées. L'exécution d'une instruction donne éventuellement lieu à un chargement vers ou depuis la mémoire, l'exécution d'une opération arithmétique, le déplacement d'un registre vers un autre, *etc.* Les cœurs peuvent théoriquement effectuer toutes les étapes du pipeline simultanément lorsque ce dernier est rempli et que chaque étape peut être effectuée en un seul cycle. Dans ce cas, chaque cœur peut effectuer autant d'instructions par cycle d'horloge que de pipeline géré par celui-ci. En pratique, le pipeline d'instructions est rarement rempli à cause des dépendances avec les instructions qui mettent plus d'un cycle à être exécutées. Les trous dans le pipeline dûs au code de l'application peuvent être néanmoins mitigés par l'exécution dans le désordre des instructions, la prédiction des branchements logiques dans le micro-code, l'exécution spéculative des branchements, le préchargement de données, *etc.*

La plus petite unité de mémoire du processeur est le registre. Les processeurs d'architecture « load-store » tels qu'utilisés en CHP suivent principalement le schéma d'instructions suivant : chargement de données depuis la mémoire centrale vers les registres, opération sur les registres, stockage du résultat dans la mémoire centrale. Les registres sont extrêmement coûteux et en nombre limité. Par conséquent, il est nécessaire de veiller à optimiser leur usage pour éviter de devoir repasser par la mémoire ou le cache. Sur les architectures x86 récentes, la latence de chargement des données dans un registre est au minimum $\simeq$1 cycle (*cf.* sous-section 2.2.1) par registre, si la donnée est préchargée



(a) Intel(R) Xeon(R) Gold 6140 (b) AMD Opteron(TM) Processor 6272

FIGURE 2.1 – Structure hiérarchique du cache dans les machines réelles telle que présentée par la bibliothèque hwloc [13]. Chaque niveau de cache est noté L_n et de taille et de degré de partage croissant selon n. Les caches suffixés par la lettre i sont des caches d'instructions, tandis que ceux suffixés par la lettre d sont des caches de données.

depuis le cache le plus proche du cœur. En revanche, elle peut aller jusqu'à plusieurs centaines de cycles et donc ralentir sévèrement le pipeline d'instructions, de sorte que même les mécanismes de mitigation internes au processeur ne suffisent pas à compenser l'attente. Lorsque les pipelines sont relativement pleins et que le parallélisme et le débit d'instructions sont quasi-optimaux, la vectorisation permet d'atteindre un niveau supérieur de parallélisme et de débit d'opérations. Les registres de vecteurs permettent de stocker plusieurs éléments. Les instructions vectorielles (SIMD) permettent d'effectuer la même opération sur tous les éléments de ces registres en une seule instruction. Sur les architectures x86 les opérations SIMD sont implémentées à travers les technologies Streaming SIMD Extension (SSE), Advanced Vector eXtensions (AVX), AVX2, AVX-512, permettant de stocker jusqu'à 512 bits d'information ou 8 éléments flottants en double précision dans un registre SIMD. L'utilisation de ces registres et instructions associées permet d'accroître le débit de données parfois au coût d'une latence légèrement supérieure. En revanche, si la donnée se trouve dans une mémoire lointaine, le débit est limité non pas par le débit d'instructions et de données du cœur mais par celui des composants intermédiaire que traverse la donnée, comme par exemple les niveaux de cache supérieurs.

2.1.2 Hiérarchie de caches

À l'intérieur du processeur, au plus près des cœurs de calcul se loge la mémoire cache. Le cache est une mémoire très rapide, qui n'est pas explicitement accessible par la couche logicielle, et généralement organisée en hiérarchie de petits caches comme illustré en Figure 2.1. Plus un niveau est loin des cœurs, plus sa capacité est élevée et sa performance est faible.

La nature hiérarchique du cache fait que les données peuvent exister et être modifiées de manière concurrente à plusieurs endroits et nécessitent une

coordination globale pour conserver les propriétés de cohérence d'une mémoire monolithique. Cette cohérence est maintenue au travers d'un protocole (Modified Exclusive Shared Invalid (MESI), Modified Owned Exclusive Shared Invalid (MOESI), Modified Exclusive Shared Invalid Forward (MESIF), *etc.*), qui associe à chaque ligne de cache un état qui permettra, selon que les accès suivants sont en lecture ou en écriture, de décider de la nécessité de propager l'information dans la hiérarchie mémoire et de la portée de cette propagation. Ce protocole impacte également la performance de manière complexe et nécessite d'être modélisé [94, 42] pour adapter les applications au matériel. Les caches partagés peuvent être inclusifs, *i.e.* ils contiennent une copie du contenu des caches fils, exclusifs, *i.e.* leur contenu est strictement disjoint des caches fils, ou des variantes, *i.e.* ils peuvent contenir une copie de certaines données des caches fils. L'inclusivité des caches influe sur le protocole de cohérence, et donc sur les performances.

Le cache implémente une politique de remplacement qui permet sous contrainte de taille, d'avoir les données nécessaires dans les niveaux les plus proches des cœurs qui y accèdent. Les politiques implémentées dans les processeurs dérivent généralement de *Least Recently Used* (remplacement du plus anciens) (LRU) [41] et de *Least Frequently Used* (remplacement du moins utilisé) (LFU) [69]. Le cache implémente également un mécanisme spéculatif de préchargement des données [109], dans le but d'avoir la donnée au plus proche des cœurs lorsqu'elles sont effectivement utilisées. En pratique il peut y avoir plusieurs préchargeurs différents par mémoire cache de la hiérarchie de cache [93]. En conclusion, une donnée a plus de chances de se trouver dans le cache près des cœurs si elle est souvent utilisée ou bien si elle est utilisée de manière prévisible par le matériel. Ces politiques sont implémentées dans le matériel et impactent largement la performance, sans toutefois être publiques. D'où l'intérêt de les modéliser [7] pour les exploiter à bon escient.

La façon dont sont organisés les accès mémoire d'une application a un impact direct sur l'utilisation du cache et donc sur la performance [90]. Par exemple, on peut organiser l'accès aux données des applications en blocs hiérarchiques [101] (comme l'est le cache) de taille correspondant aux différents niveaux de ce sous-système de manière à ordonner par fréquence l'accès aux données et de se conformer à la politique du cache. On peut également s'assurer que les données sont accédées par paquets de la taille d'une ligne de cache, pour minimiser le déplacement de données dans le cache, *etc.*

2.1.3 Mémoire volatile adressable

La mémoire vive principale est une mémoire volatile (c'est-à-dire qui se vide en l'absence d'alimentation électrique), adressable par octet, située généralement à l'extérieur de la puce du processeur, plus loin que les caches, sur la carte mère d'un nœud de calcul. La mémoire volatile est découpée logiciellement en

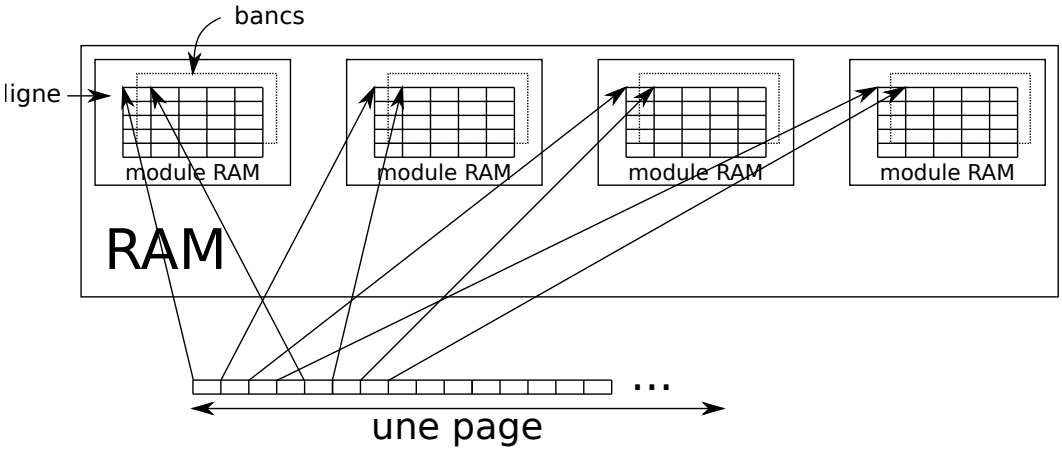


FIGURE 2.2 – Organisation d'un nœud NUMA.

pages (de 4096 octets sur x86). Les pages sont stockées par banc dans les différents modules de mémoire physique. Avec l'aide du système d'exploitation, les pages d'adresses virtuelles contiguës sont réparties en ronde, module mémoire par module mémoire, banc par banc, pour exploiter le parallélisme à l'intérieur de la mémoire [98] comme décrit en Figure 2.2. La manière d'accéder aux données impacte la performance des accès mémoire selon qu'elle exploite ou non (de manière implicite) les mécanismes d'optimisation dans le matériel : parallélisme entre les modules, cache interne à la mémoire, réutilisation des lignes déjà activées, *etc.* Celle-ci est généralement optimale lorsque les données sont accédées selon un schéma d'adresses très prévisible, *e.g.* des adresses contiguës ou régulièrement espacées. L'exploration d'une plage d'adresses virtuelles est sujette à la construction d'une table des pages hiérarchique, qui est en partie contenue dans un Cache de traduction d'adresses (*Translation Lookaside Buffer*) (TLB), qui permet de faire la traduction entre les adresses virtuelles vues par l'application et les adresses physiques propres au matériel. Le TLB étant d'une taille limitée, la performance des accès mémoire est aussi limitée à ce niveau par l'exploitation de cette structure.

Actuellement dans les processeurs généralistes des nœuds de CHP sont équipés de jusqu'à deux types de mémoires volatiles adressables (par octet), *i.e.* deux nœuds NUMA (Accès Non-Uniform à la Mémoire) : l'une en dehors de la puce de calcul accessible via les canaux mémoire traditionnels et l'autre, moins courante, directement sur la puce du processeur. Ces deux mémoires diffèrent en pratique par leur performance et leur capacité, et le choix de l'allocation dans l'une ou de l'autre est actuellement laissé à l'utilisateur. Il existe plusieurs travaux sur les compromis et l'optimisation de l'utilisation de ce genre de mémoire hétérogène [104, 18, 44], mais il n'y a pas de solution à la fois standardisée et performante pour automatiser le choix de la mémoire.

La mémoire volatile est du point de vue de l'application présentée par le

système d'exploitation comme un espace plat et linéaire d'adresses. Cependant, même à l'échelle d'un seul processeur, elle peut être hétérogène avec des accès non uniformes (NUMA). En effet, selon la source et la destination des accès, la performance de ces derniers varie, et il est généralement préférable d'écourter le chemin des données pour améliorer la performance des accès, notamment grâce à la diminution de la latence, l'augmentation du débit et la diminution de la congestion sur le réseau. Il est possible à l'échelle d'un système d'exploitation, d'utiliser plusieurs processeurs interconnectés sur plusieurs cartes mères, avec chacune leur module de RAM volatile. Chaque module ou nœud NUMA est accessible localement par le processeur de la carte mère associée ou à distance à travers le réseau d'interconnexion, et ceci de manière totalement transparente pour l'utilisateur qui voit toujours le même espace d'adressage linéaire. On décompose de tels machines en domaines NUMA, *i.e.* un ensemble de cœurs locaux à une ou plusieurs mémoires, et interconnectés (à l'intérieur du processeur, ou entre plusieurs processeurs).

La majorité des machines NUMA sont *cache-cohérentes* et NUMA, c'est-à-dire que la cohérence et le déplacement des données entre les différentes mémoires et les différents processeurs sont gérés de manière transparente par le matériel. La cohérence impacte significativement la performance [49], ce qui motive le fait de modérer l'accès à des données partagées et leur bon placement pour tempérer ce coût. Cette interconnexion a lieu à l'intérieur d'un seul processeur ou bien entre plusieurs processeurs. Actuellement, le fabriquant Intel connecte les quatre « quadrants » (quarts de processeur : 4 groupes de cœurs et mémoires) du processeur KNL et de la gamme de processeurs Xeon par un maillage tandis que les générations précédentes utilisaient un anneau.

A l'échelle de plusieurs processeurs, le même fabriquant connecte jusqu'à 4 processeurs avec un graphe complet dans sa technologie Quick Path Interconnect (QPI), ou plus récemment UltraPath Interconnect (UPI). Bull et SGI proposent également de connecter un plus grand nombre de nœuds avec des solutions au dessus de UPI, *i.e.* respectivement Bull Coherent Switch (BCS) et NUMALink.

Contrairement à l'allocation de la mémoire qui est explicite, les mouvements de données sont implicites, et impactent la performance différemment selon l'échelle à laquelle ils ont lieu. Un schéma représentant de tels organisations est donnée en Figure 2.3. À gauche une organisation similaire à celle des processeur KNL avec quatre mémoires rapides (MCDRAM) sur la puce du processeur et quatre mémoires plus grandes en dehors, le tout connecté aux cœurs par un maillage (en carré pointillé). À droite une organisation de type UPI regroupant quatre processeurs dont les cœurs (par groupe de 10) et les deux mémoires sont interconnectés par deux anneaux. Sur de tels systèmes de calcul, la mémoire virtuelle peut par exemple être associée physiquement aux mémoires de manière entrelacée comme illustré en Figure 2.4. Selon le schéma d'accès aux adresses virtuelles, différents modules mémoire proches ou distants

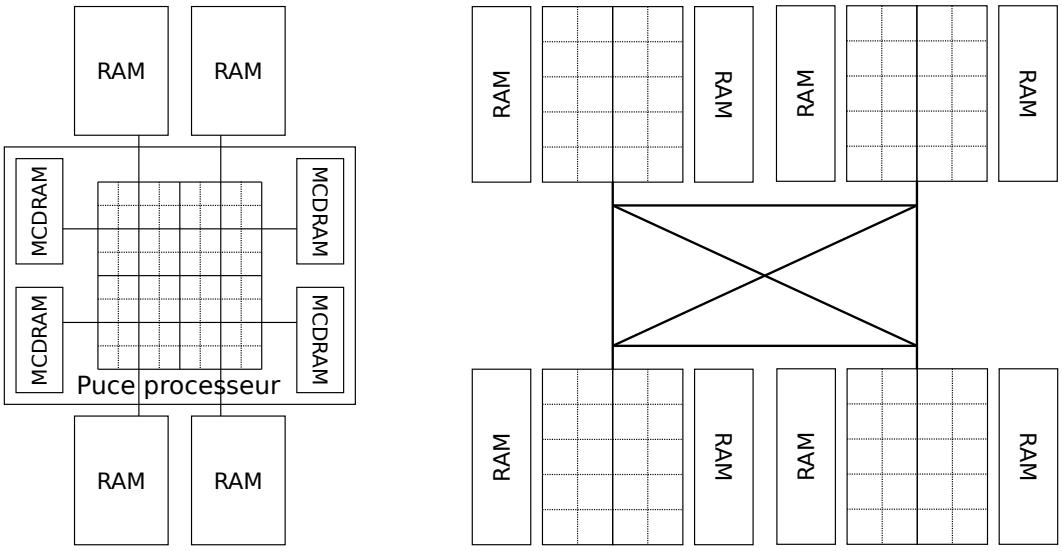


FIGURE 2.3 – Interconnexions possibles des cœurs et des mémoires dans deux machines NUMA. Interne au processeur à gauche, entre différents processeurs à droite.

sont physiquement accédés.

Les systèmes de calcul distribués incluent plusieurs nœuds de calcul à mémoire partagée avec des technologies réseau homogènes (*i.e.* une seule technologie pour le réseau) mais différentes de ces derniers (*i.e.* différent de UPI, BCS, *etc.*). Dans de tels systèmes, il est possible d'accéder à la mémoire d'autres nœuds de calcul interconnectés par un réseau haute performance (infiniband [3], omnipath [9], Bull eXascale Interconnect (BXI) [31]), par l'intermédiaire de paradigmes (*e.g.* passage de messages) qui permettent de lire et écrire sans l'intervention du nœud distant des espaces réservés de la mémoire volatile de ce dernier. De même, il existe des mémoires volatiles attachées aux commutateurs réseau qui peuvent servir d'intermédiaire de transmissions et de stockage temporaire de données dans de tels systèmes de calcul distribués (à plusieurs système d'exploitation). L'exploitation de systèmes CHP à cette échelle apporte de manière assez évidente un degré de complexité supplémentaire.

2.1.4 Mémoire non-volatile

La mémoire non-volatile est découpée logiciellement à la granularité d'un bloc qui diffère de la granularité des pages utilisée pour la gestion de la mémoire vive. Généralement, chaque nœud de calcul dispose d'un espace de stockage local accessible via l'interface Posix (`open`, `read`, `write`, `close`, *etc.*). En 2018 Atos livre également des lames de calcul bi-nœud avec chacun un stockage local mais accessible par l'autre paire de la lame en cas de panne ou d'extinction du

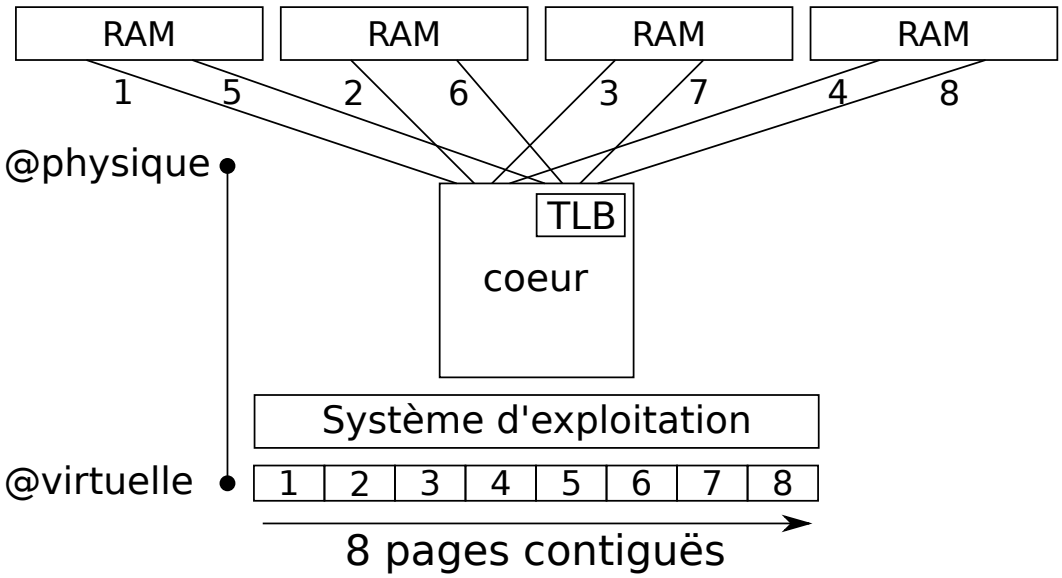


FIGURE 2.4 – Exemple d'association des pages virtuelles et physique dans la mémoire d'une machine.

nœud connecté au disque. Dans les années à venir, la mémoire RAM volatile conventionnelle sera peut-être accompagnée d'une mémoire RAM non-volatile (NVRAM) [1], gérée de manière transparente dans le matériel ou par une couche logicielle, et qui devrait grandement augmenter la capacité mémoire des machines [82] tout en offrant la persistance des données à un coût en performance beaucoup plus abordable qu'il ne l'est actuellement.

Les grappes de calcul dédient généralement un sous ensemble de nœuds à la gestion d'un espace de stockage, hautement parallèle et partagé sur toute la machine. De plus, les systèmes CHP plus récents mettent en œuvre des nœuds de cache d'ES (bufferIO ou burst-buffer), plus proches des nœuds de calcul et qui permettent d'absorber en partie le trafic de lecture/écriture non-volatile et de décongestionner le stockage principal. En interne, le stockage centralisé peut également avoir une organisation complexe qui découple les nœuds de gestion de la grappe de stockage des nœuds d'ES. À la fois le stockage distribué (disques locaux à chaque nœud + caches d'ES) et centralisé sont utilisés selon les besoins en partage, en capacité, en performance, *etc.* des applications, et les contraintes de partage de la machine.

L'espace de stockage centralisé est accessible depuis chaque nœud du cluster par l'intermédiaire d'un système de fichier réseau. Celui-ci permet de cacher la caractéristique distante et distribuée du stockage physique pour effectuer les opérations d'ES de la même manière que le stockage local et vient en remplacement de ce dernier. Néanmoins l'interface classique POSIX limite grandement le parallélisme des accès aux fichiers écris sur les disques, et est remplacé dans les systèmes de calcul distribués par des gestionnaires de systèmes de fichier

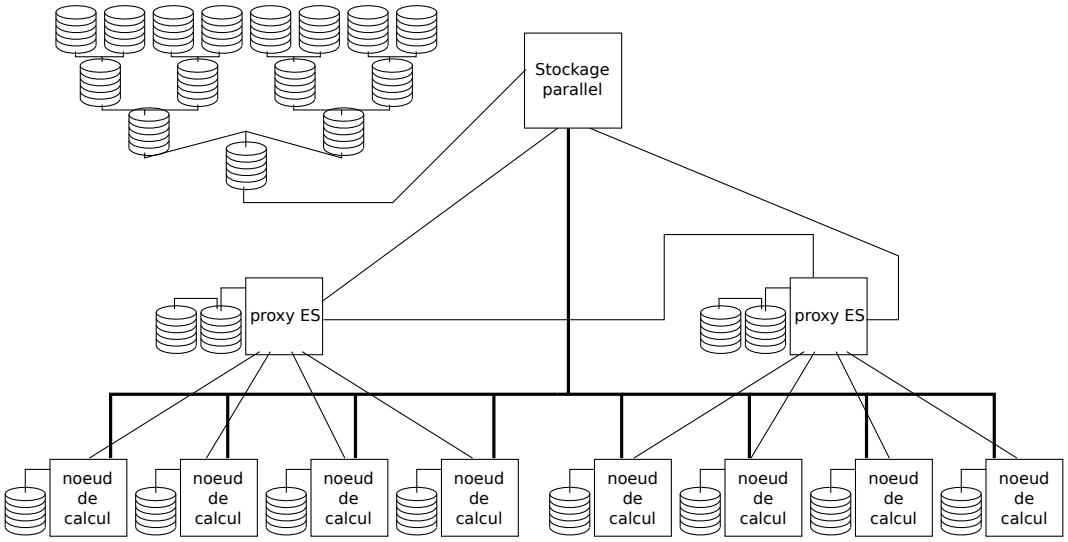


FIGURE 2.5 – Exemple d'une organisation hiérarchique du système de stockage disposant de caches d'ES (proxy ES) interconnectés et d'un stockage parallèle centralisé pour tout le système.

parallèles haute performance [103]. Par exemple, une des couches externes de tels systèmes de fichiers est l'interface Message Passing Interface (MPI)-IO [20] qui expose des primitives héritées du modèle de communication MPI [48]. Ces mécanismes permettent de masquer efficacement la complexité du système de stockage (dont un exemple simplifié est illustré en Figure 2.5), cependant il n'y a pas de connexion explicite entre la sémantique de l'application et les caractéristiques qualitatives (topologie, types de nœuds stockage), et quantitative (performance, capacité, *etc.*) des nœuds. Il faut nécessairement paramétrer les couches intermédiaires partagées sur toute la machine, *i.e.* pour toutes les applications. Donc, la performance du stockage est sujette à des optimisations hybrides spécifiques à l'application sur le stockage local, mais également à l'échelle du système de calcul entier, pour le stockage centralisé. Cette optimisation doit tenir compte de la topologie du système de stockage pour être efficace [111].

Dans ce manuscrit, nous nous intéressons à l'échelle d'un seul nœud de calcul, c'est-à-dire au paradigme à mémoire partagée, sans considérer les défis spécifiques à la mémoire non-volatile.

2.1.5 Modèle de représentation des nœuds de calcul

La bonne exploitation des architectures à mémoire partagée requiert une interface simple et générique pour appréhender leur complexité. Cela permet notamment de produire des méthodes d'optimisation portables et adaptables d'une machine à une autre. La structure des caches est toujours hiérarchique

dans les machines récentes même dans celles où le dernier niveau de cache est également configurable en mémoire adressable (par octet). La structure de cette hiérarchie est accessible sur les architectures x86 via l'instruction `cpuid`. La mémoire locale à chaque processeur est découverte par le BIOS au démarrage de la machine, et propagée à travers le réseau d'interconnexion. Elle est ensuite consultable avec un appel système (fonction de communication avec le noyau du Système d'Exploitation (SE)) via l'interface `numaif.h` ou via la lecture dans le système de fichiers virtuels `/sys/` sous Linux. L'agencement des mémoires et caches ainsi découverts est varié et la modélisation de leur structure dans leur généralité correspond à un modèle en graphe dont les sommets sont des composants de la topologie (*e.g.* mémoires, caches, cœur, thread matériel,*etc.*) et les arêtes des connexions entre les sommets.

Une première approche consiste à considérer uniquement les cœurs et les mémoires comme sommets et la totalité des liens entre eux [17] dans un graphe complet pondéré par des métriques de performance des liens. Cette technique très haut niveau ne tiens pas compte des phénomènes complexes liés à des composants intermédiaires partagés à travers la machine (tels que les caches) et qui impactent la latence des échanges, *i.e.* le poids des arêtes, selon si la mesure tient compte de cette structure ou non.

À ce jour, la bibliothèque hwloc [13] est l'outil qui fourni l'information qualitative la plus complète sur les composants de la topologie des nœuds de calcul. hwloc fournit une topologie statique issue du système d'exploitation, des pilotes (graphiques, réseau, *etc.*), et du processeur. hwloc propose de structurer les éléments de la topologie matérielle sous forme entièrement hiérarchique. Cette structure est globalement assez fidèle à la topologie générale des machines NUMA. Une mise à jour récente [47] de la bibliothèque permet même d'intégrer des mémoires hétérogènes tout en conservant cette structure. Cependant la structure réelle des machines ne correspond pas exactement à la représentation hiérarchique de hwloc (dont la comparaison est donnée en Figure 2.5) sur certains aspects :

– Sur les architectures Sandy-Bridge, Haswell, Broadwell, d'Intel, les cœurs sont interconnectés en anneau. En nombre de « sauts », hwloc considère que le chemin entre deux cœurs passe par le cache partagé le plus proche alors qu'il peut passer plutôt par l'anneau. Sur les architectures Skylake et KNL plus récentes, cette interconnexion est effectuée au niveau du cache L2 par un maillage. Pour des micro-*benchmarks* précis ou certaines fonction de synchronisation telles que les barrières, la topologie du réseau d'interconnexion des cœurs peut éventuellement avoir un effet sur la performance [97].

– hwloc représente parfois les caches L3 partagés hiérarchiquement plus loin des cœurs que les mémoires, alors que les données sont dabord lues/écrites des cœurs (feuilles) au cache, puis du cache vers la mémoire.

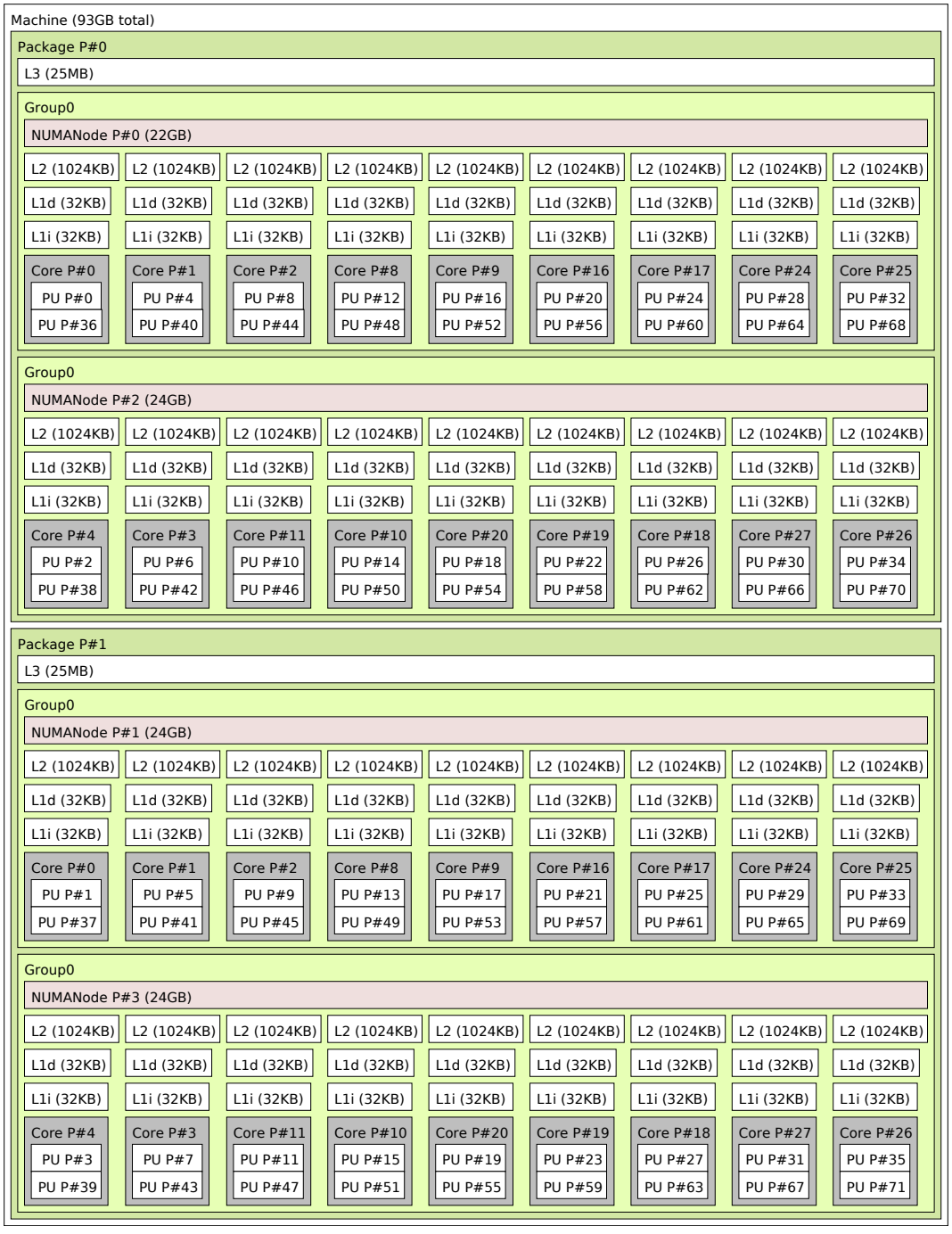
- La configuration de certaines machines permet de découper la mémoire en plusieurs domaines NUMA représentés comme plus proche de certains cœurs par hwloc, mais qui pourtant montrent des performances invariantes (*cf.* sous-section 2.2.5) selon les cœurs qui accèdent à ces mémoires. La topologie de l'interconnexion est toutefois prise en compte en dehors de la structure hiérarchique de la topologie, via une matrice de distances symboliques.
- Dans la plupart des machines, les « groupes » NUMA présentés par hwloc sont connectés au niveau du contrôleur mémoire situé entre le dernier niveau de cache et la mémoire et permettent d'échanger des données sans avoir à passer par la mémoire adressable (par octet) et donc implique qu'il existe un lien entre les derniers niveaux de cache qui casse la hiérarchie telle que proposée par hwloc.

À l'avantage de la vitesse d'obtention de cette information s'oppose la partialité de l'information structurelle qui n'inclut pas des composants qui ne sont pas logiciellement connectable à la topologie mais qui ont bien un impact sur les performance des applications (*e.g.* les contrôleurs mémoire [85]). La méthode de découverte de hwloc n'inclut pas non plus des informations quantitatives telles que la bande passante des mémoires et des caches, le débit d'instructions de calcul flotant des cœurs, *etc.*

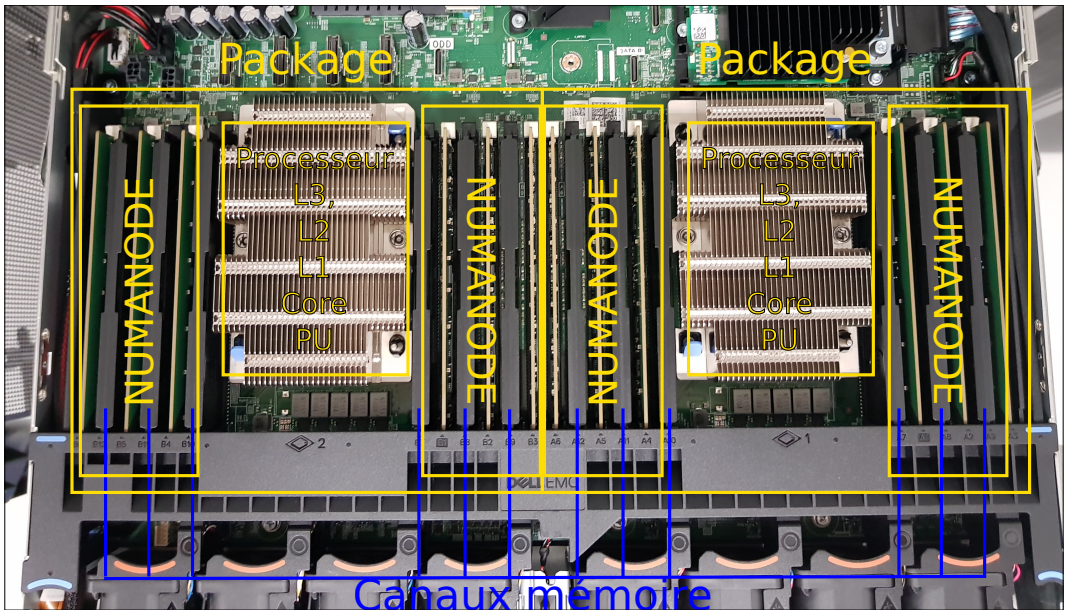
Le modèle structurel fourni est une pièce essentielle à l'optimisation des performance des machines parallèles car il est indispensable d'énumérer et de structurer les goulets d'étranglement matériels sur le chemin des données afin qu'elles soient achéminées plus rapidement près des cœurs où elles sont utilisées ou des mémoires où elles sont stockées. Il nécessite cependant d'être complété par un modèle empirique qui ajoute la quantification des distances dans la topologie, et la découverte/modélisation éventuelle des liens qui ne sont pas représentés.

2.2 Caractérisation des performances des mémoires

La performance des composants mémoire est généralement caractérisée par la bande passante et la latence. La latence correspond au temps du transfert d'une unité de donnée depuis la source vers la destination, tandis que la bande passante mesure le débit des données entre ces deux points. On peut visualiser ces métriques caractéristiques comme représentatives de la longueur du tuyau mémoire pour la latence, et le diamètre du tuyau pour la bande passante. À l'échelle du système de calcul, pour mesurer la bande passante mémoire, il faut tenir compte des composants intermédiaires entre la mémoire et les cœurs, qui peuvent être éventuellement dans une situation de contention selon le schéma



(a) Modèle hwloc en sortie de lstopo



(b) Organisation réelle en photo

FIGURE 2.5 – Comparaison entre le modèle de représentation de hwloc et le positionnement physique réel des composants d'une machine bi-*socket* équipée de processeurs Intel(R) Xeon(R) Gold 6140.

d'accès aux données, et qui bornent la bande passante observable, éventuellement en deça de la bande passante mémoire. À l'instar de la bande passante, la latence agrège plusieurs réactions : la rapidité de la mémoire à répondre à une requête (qui dépend du schéma d'accès), et la longueur du chemin à parcourir, qui traverse éventuellement plusieurs composants et déclenche plusieurs mécanismes qui ralentissent les données.

2.2.1 Micro-benchmark séquentiel de bande passante

Pour atteindre la bande passante d'un niveau de mémoire et exploiter les mécanismes d'optimisation de la mémoire et du cache, il faut nécessairement que le schéma d'accès soit continu et contiguë (principe de localité spatiale *cf.* sous-section 3.1). La bande passante mesurée peut être limitée par le débit d'instructions des cœurs si les instructions comportent des dépendances, des branchements, *etc.* et non par l'accès mémoire lui-même. Il est donc nécessaire de veiller à limiter au maximum toute instruction qui n'est pas un accès mémoire. L'utilisation d'instructions vectorielles permet à débit d'instructions équivalent de charger plus de données, jusqu'à atteindre le débit la mémoire cible. Ceci est particulièrement vrai lorsqu'on mesure la bande-passsante du premier niveau de cache, capable sur certaines architectures de soutenir un débit de 3 instructions vectorielles d'accès mémoire par cycle, alors que la

```
loop_load_repeat:
mov %1, %%r11
mov %2, %%r12
buffer_load_increment:
vmovapd %%r11, %%zmm0
vmovapd 64(%%r11), %%zmm1
vmovapd 128(%%r11), %%zmm2
...
vmovapd 960(%%r11), %%zmm15
add $1024, %%r11
sub $1024, %%r12
jnz buffer_load_increment
sub $1, %0
jnz loop_load_repeat
```

FIGURE 2.6 – Boucle de banc d'essai exploitant le débit maximum d'instructions vectorielles (AVX-512) des cœurs sur une architecture de type x86. Les données sont chargées de manière continue et contiguë, depuis la mémoire, par blocs de 1024 octets.

vectorisation est moins intéressante dans le dernier niveau de mémoire. Par exemple, en Figure 2.6, est décrit le cœur d'un code efficace pour mesurer la bande passante d'une mémoire, en optimisant le débit d'instructions et le schéma d'accès à la mémoire. L'adresse de début des données est chargée dans le registre r11, la taille totale des données dans le registre r12 et le code charge les données dans des registres vectoriels de taille 64 octets, *i.e.* ligne de cache par ligne de cache, par blocs de 1024 octets jusqu'à parcours total des données. L'opération est répétée autant de fois que contenu dans le registre fourni en premier argument de la fonction.

On ne peut pas accéder au cache de niveau R_n de manière explicite, néanmoins on peut accéder à un niveau de cache R_n, père d'un cœur, de manière quasi-déterministe, en exploitant la politique de réutilisation des données et la capacité C_n du cache inférieur R_{n-1}. Une lecture continue et contiguë (éventuellement circulaire) d'une plage d'adresses de « bonne » taille suffit pour mesurer la bande passante d'un niveau de mémoire C_{n+1}.

- Un ensemble de données de taille $T = C_n + R_n$, $T < C_{n+1}$ déborde nécessairement du cache C_n et loge dans le cache C_{n+1}.
- Si on ne fait que lire plusieurs fois l'ensemble T des données, alors celles-ci logeront au moins dans le cache C_{n+1} car le cache doit stocker les données réutilisées.
- L'accès aux C_n premières données provoque des défauts de cache obliga-

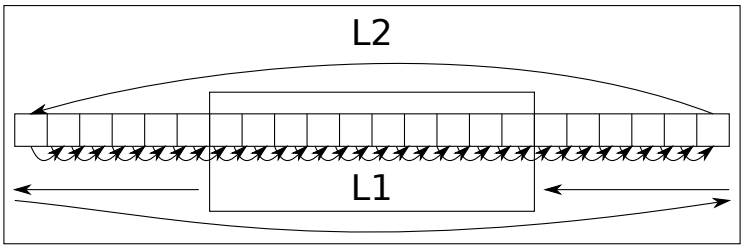


FIGURE 2.7 – Illustration de la technique de débordement de cache pour accéder à des données dans un niveau de cache arbitraire, dans le cas d'une mesure de bande passante avec des accès contiguës.

toires dans le cache C_n qui le remplissent. Donc les données ne sont pas lues depuis C_n.

- Chacun des accès subséquents à une nouvelle donnée remplace nécessairement une des données dans le cache C_n. Toutes les données présentes n'ont été utilisées qu'une seule fois. Donc, si on considère que le cache implémente une politique *Least Frequent Recently Used* (remplacement du plus ancien des moins utilisés) (LFRU), l'élément remplacé est le premier arrivé dans le cache, *i.e.* l'adresse de début de tableau. Au fur et à mesure que les adresses de fin de tableau sont accédées et mises en cache, les adresses de début de tableau sont évincées du cache selon la même logique et remplacées par des adresses qui ne sont utilisées qu'une fois. Donc toutes les lectures suivantes ne se font pas depuis le cache C_n
- Si toutes les données logent dans le cache C_{n+1} et ne sont pas lues depuis C_n ou un niveau inférieur, alors elles sont lues depuis C_{n+1}.

L'illustration (Figure 2.7) schématise ce mécanismes en faisant glisser le cache de petite taille le long des accès. On voit que la donnée la plus ancienne est retirée du petit cache à chaque étape, alors que la nouvelle donnée est lue depuis le cache de plus grande taille.

Donc on peut arbitrairement mesurer le débit d'un niveau de mémoire avec un parcours circulaire, continu et contiguë d'adresses. On utilise cette méthode avec le code optimisé en Figure 2.6 pour mesurer successivement la performance séquentielle (*i.e.* mesurée avec un seul cœur) des caches et des mémoires découverts par hwloc pour la machine représentée en Figure 2.5. Les résultats de cette mesure, dont la valeur médiane sur dix échantillons est rapportée pour chaque point sont représentés en Figure 2.8 (gauche), et montrent l'existence de paliers entre les différents niveaux de cache qui attestent de la validité de la méthodologie pour mesurer la bande passante d'un niveau de cache spécifique.

En conclusion, on constate que plus les données lues sont loin du cœur moins la bande passante est élevée. Cette observation est généralisable à d'autres

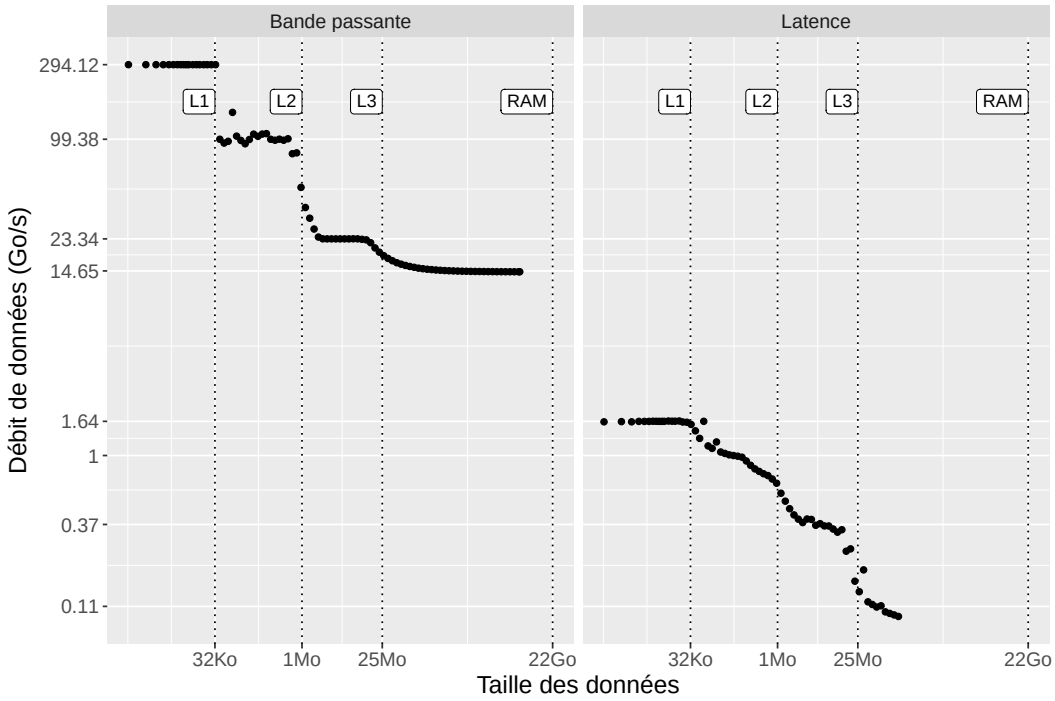


FIGURE 2.8 – Performance séquentielle des différents niveaux de mémoire d'un processeur Intel(R) Xeon(R) Gold 6140 en fonction de la taille des données

systèmes de calcul, car l'organisation de leurs caches est aussi hiérarchique et qu'ils implémentent des politiques de chargement et de remplacement des données similaires.

2.2.2 Impact du schéma d'accès aux données.

Pour mesurer une simili-latence mémoire, on essaye au contraire de se placer dans le pire cas d'utilisation de la mémoire. On décide de parcourir un tableau d'adresses pointant aléatoirement vers une autre partie du tableau. Ce schéma d'accès aux données permet de mettre en défaut plusieurs mécanismes :

- Le préchargement de données échoue parce que les adresses des accès sont totalement imprévisibles,
- Dans la RAM les accès successifs se font dans des modules, bancs, lignes et colonnes différents et empêchent la mémoire de pipeliner les opérations d'accès dans la mémoire,

tout en conservant une utilisation « raisonnable du pipeline du processeur car le code ne contient pas de branchement, ne nécessite pas d'exécution spéculative, et n'occupe pas le processeur avec d'autres instructions que celles prévues pour le banc d'essai. Des outils comme Intel Memory Latency Checker (MLC)

utilisent les droits administrateur pour désactiver les pré-chargeurs de données des caches et mesurer plus précisément la latence mémoire.

En utilisant la méthode de mesure de la bande passante pour mesurer de latence, si les adresses son réparties de manière aléatoire uniforme, la probabilité d'un accès mémoire de se trouver dans le cache est $\frac{1}{1+\frac{R_n}{C_n}}$, donc si $R_n \gg C_n$ on peut négliger les accès au caches de niveau inférieur, ce qui nous place dans les mêmes conditions que des accès continus pour contourner certains niveaux mémoire. Sur la Figure 2.8 (droite) on mesure le débit de données en fonction de la taille de l'ensemble de données à accéder pour un schémas d'accès aléatoire. On observe toujours les paliers du premier niveau de cache et de la mémoire RAM mais les paliers des niveaux intermédiaires sont moins évidents à déceler.

En conclusion, on constate qu'un schéma d'accès non contiguë, non continu et imprévisible impacte largement la performance de la mémoire et plus les données lues sont loin du cœur plus la latence est élevée. On peut également généraliser cette conclusion à d'autres systèmes de calcul, car les technologies employés pour la gestion de la mémoire dans les autres processeurs généralistes sont similaires à celle décrite et expérimenté dans ce chapitre.

2.2.3 Impact du partage de données sur la durée des accès mémoire

Selon l'état d'une donnée tel que décrit par les protocoles de cohérence (MESI, MOESI, *etc.*), *i.e.* selon la nécessité de propager ou non l'information d'un accès à tout ou partie de la hiérarchie mémoire, le débit de lecture ou d'écriture d'une donnée varie. Si on prend une paire de cœurs comme lecteurs/écrivains d'un bloc de données, on peut mesurer la durée de l'accès au bloc, selon si celui-ci est privé ou partagé, selon l'ordre et l'identité des lecteurs et écrivains grâce à l'outil mbench [78]. On exécute 2 scénarios : Pour un gradient de tailles de 1024 Octets à 767 Mo, le cœur `P#0` écrit la donnée puis :

- le cœur P#0 lit la donnée (cas exclusif),
- ou alors, le cœur P#1 lit la donnée (cas partagé).

Pour une taille de donnée contenue dans un cache de niveau 2 (L2), nous représentons en Figure 2.9 le chemin hypothétique des données selon chacun des deux scénarios proposés. D'un point de vue de la distance le cas partagé semble désavantagé, si la donnée se trouve dans le cache L1 ou L2. En théorie il l'est également d'un point de vue des mécanismes de cohérence. Sur cette machine, le seul cache partagé (L3) est non-inclusif, *i.e.* il peut contenir des données des caches fils. Donc le protocole de cohérence peut nécéssiter de marquer certaines lignes de ce cache et de ses fils dans l'état Shared, pour le deuxième scénario, même si les données tiennent dans un cache privé. En

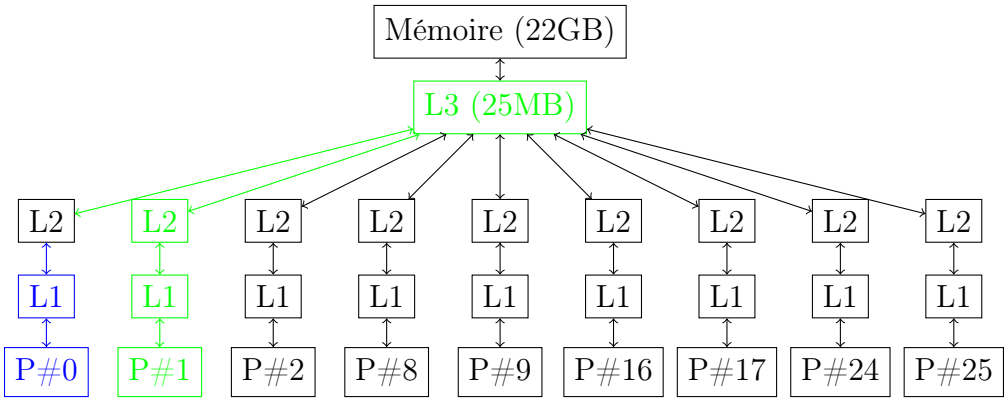


FIGURE 2.9 – Représentation schématique de la topologie du premier nœud NUMA d'un processeur Intel(R) Xeon(R) Gold 6140, avec le chemin emprunté par une donnée du cache L2 du cœur P#0, selon qu'elle est accédée en bleu par le cœur P#0 (cas exclusif) ou en vert par le cœur P#1 (cas partagé).

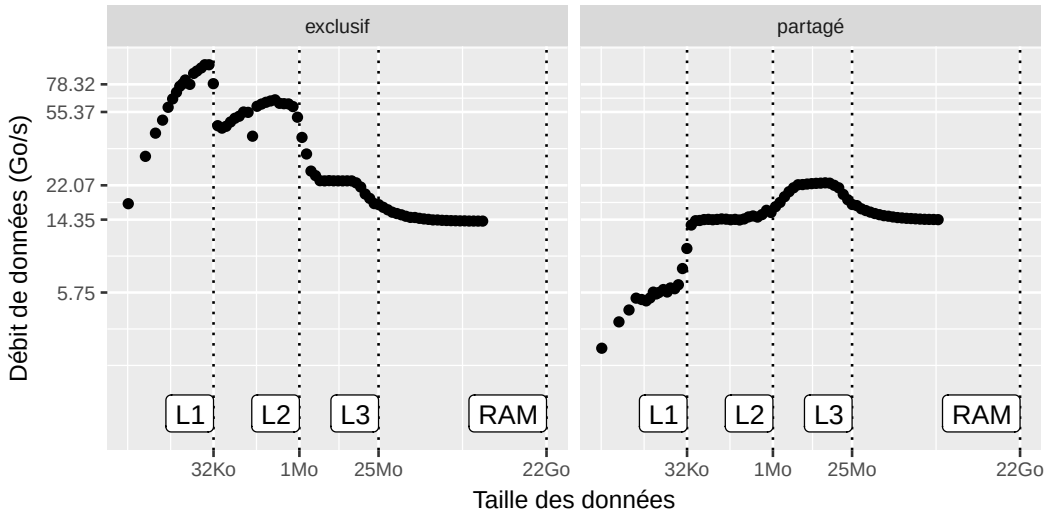


FIGURE 2.10 – Débit des accès selon l'état des données dans le cache et la mémoire sur un processeur Intel(R) Xeon(R) Gold 6140

Figure 2.10 nous représentons la bande passante mesurée[1] selon le scénario (exclusif ou partagé). On remarque que la bande passante obtenue est moins grande dans le cas partagé pour les 2 premiers niveaux de cache privés, puis qu'elle est équivalente à la bande passante mesurée en Figure 2.8 pour les niveaux partagés (L3 et RAM) dans les deux scénarios. De plus, dans le cas partagé c'est le cache partagé le plus loin des cœurs qui obtient le plus de bande passante, car le transit des données et des informations de cohérence est plus court.

En conclusion, on constate que la cohérence mémoire et le partage des données a un coût non négligeable sur la durée des requêtes d'accès aux données.

2.2.4 Effet du parallélisme sur la bande passante et la latence

Dans les processeur multicœurs contemporains, le parallélisme des accès mémoire permet d'augmenter les performance globale des accès à la mémoire par rapport à des accès séquentiels. Par exemple les différents modules mémoire peuvent être accédés en parallèle par différents cœurs à condition qu'il y ait assez de canaux mémoire et que le débit du contrôleur mémoire soit suffisant. Il y a donc une limite à la performance parallèle des mémoires qui ne passe pas nécessairement à l'échelle du nombre de cœurs.

Pour mesurer la bande passante concurrente il faut s'affranchir des mécanismes de cohérence qui peuvent ralentir le débit des données en utilisant uniquement des données privées à chaque fil d'exécution. Nous proposons d'observer l'effet de l'augmentation du nombre de fils d'exécution qui accèdent à un niveau de mémoire (maximum 1 pour le cache L1, 1 pour le L2, 14 pour le L3, 28 pour la RAM) sur la bande passante de cette mémoire.

En Figure 2.11 est comparé le débit de données en lecture, délivré par chacun des niveaux mémoire, lorsqu'il est accédé de manière concurrente, en fonction de la taille des données accédées. Nous représentons en couleur bleue, foncée à clair, le nombre de threads qui accèdent au niveau de mémoire partagé considéré (seuls les niveaux L3 et DRAM sont partagés). Malgré le coût hypothétique de synchronisation des *threads*[2], la bande passante des caches privés reste inchangée par rapport au cas séquentiel. La bande passante du dernier niveau de cache passe à l'échelle du nombre de processus légers et dépasse celle des premiers niveaux de cache. En revanche, en ce qui concerne la mémoire RAM, on voit sur la Figure 2.12 que la bande passante des niveaux partagés ne passe pas à l'échelle du nombre de processus légers, et est donc sensible à la contention. Le nombre optimal de threads pour l'utilisation de

1. Les valeurs reportées correspondent à la médiane sur une dizaine de mesures.
2. fil d'exécution

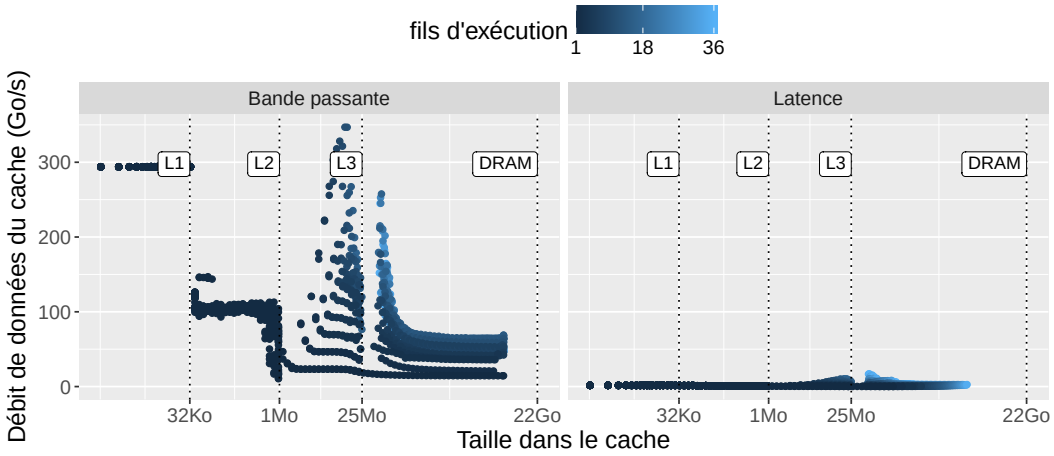


FIGURE 2.11 – Performance parallèle des différents niveaux de mémoire d'un processeur Intel(R) Xeon(R) Gold 6140 en fonction de la taille des données et du niveau de parallélisme (nombre de fils d'exécution simultanés).

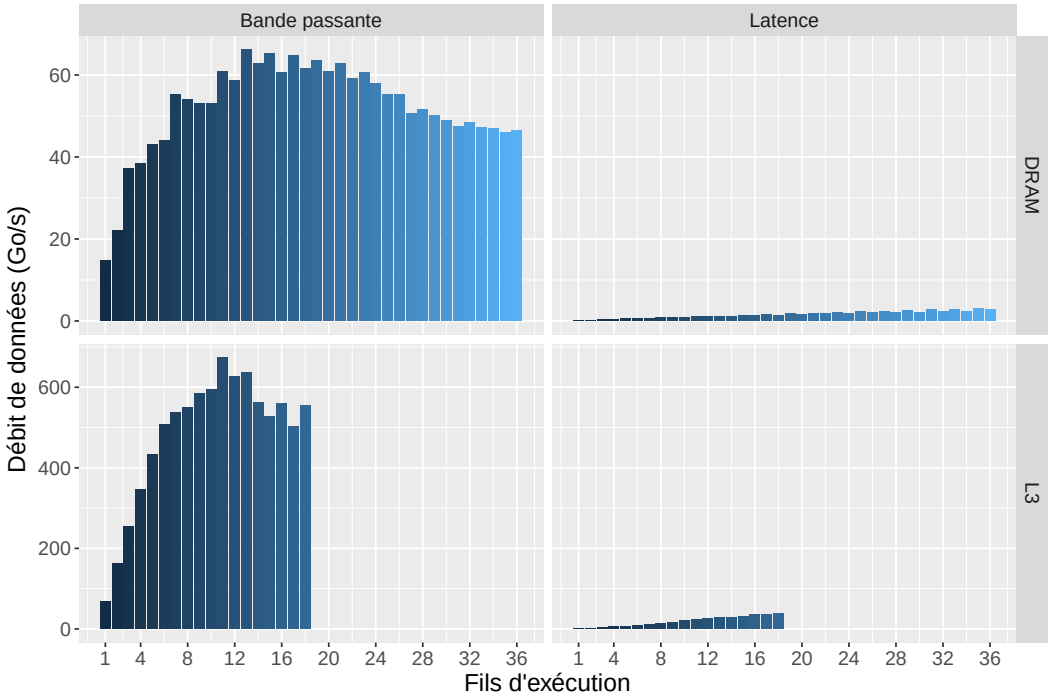


FIGURE 2.12 – Performance parallèle médiane des niveaux de mémoire partagés d'un processeur Intel(R) Xeon(R) Gold 6140

la RAM correspond au nombre de cœurs locaux à cette mémoire 2.5. Si on observe la latence parallèle (Figures 2.11, 2.12), dans tous les cas on a intérêt à utiliser un maximum de cœurs, comme le suggère le gradiant de couleur. Sur la Figure 2.11, au dernier niveau de cache et pour la RAM, plus le nombre de threads est élevé, plus la taille minimum des données dans cette mémoire est grande à cause de l'augmentation de la taille du cache inférieur (relatif au nombre de cœurs utilisés), qui pourrait entrer en jeu pour des tailles plus petites.

En conclusion, on constate que la bande passante des différentes mémoires ne passe pas nécessairement à l'échelle du nombre de cœurs qui la partagent, donc la contention sur la mémoire est un facteur non négligeable de dégradation des performances. De plus, le débit de données en latence est plus petit que le débit maximum de la mémoire de plusieurs ordres de grandeur, donc le schéma d'accès aux données joue un rôle prépondérant dans la performance des accès mémoire.

2.2.5 Impact des effets NUMA sur la performance des accès mémoire

Dans le matériel, la performance d'une mémoire volatile (adressable) est relative à l'observateur, *i.e.* le ou les cœurs qui exécutent les instructions d'accès à la mémoire car il existe plusieurs composants de la machine, connus ou non, partagés ou non, entre les cœurs et la mémoire qui peuvent impacter la performance observable. Nous associons à la notion de perspective une source, *i.e.* le ou les cœurs qui font les accès mémoire et une destination, *i.e.* la ou les mémoires qui servent les requêtes. La connaissance de la structure de la mémoire par rapport aux cœurs permet de réduire l'exploration des scénarios pertinents. En particulier la symétrie de la machine réduit considérablement l'espace à explorer. Le maximum de bande passante mémoire est atteint lorsque tous les cœurs d'un domaine NUMA [3] font des requètes vers ce nœud (*cf.* Figure 2.12). Donc nous considérons sur notre machine de référence (Figure 2.5), 4 sources et 4 destinations. La totalité des distances peut être représentée sour la forme d'une matrice $4 * 4$ dont les lignes sont des ensembles de cœurs associés à une mémoire et les colonnes sont des mémoires. Une latence relative théorique entre chaque paire est inscrite de manière statique dans le matériel et peut être obtenue grâce à la bibliothèque hwloc. La distance (en bande passante) peut être mesurée avec la méthodologie décrite jusqu'ici, en allouant explicitement les données dans la mémoire cible, et en attachant les fils d'exécution aux cœurs du domaine source.

3. ensemble de cœurs fils d'un nœud de la topologie contenant au moins une mémoire NUMA

domaine	P#0	P#2	P#1	P#3
P#0	10	11	21	21
P#2	11	10	21	21
P#1	21	21	10	11
P#3	21	21	11	10

(a) Latence constructeur (sans unité).

domaine	P#0	P#2	P#1	P#3
P#0	55,4	55,5	34,4	32,8
P#2	57,6	57,4	34,5	32,7
P#1	34,5	33,2	57,5	57,5
P#3	34,5	33,2	57,7	57,3

(b) Bande passante (Go/s).

TABLE 2.1 – Matrice des distances d'une machine bi-*socket* équipée de deux processeurs Intel(R) Xeon(R) Gold 6140.

Sur la Figure 2.1 représentant la matrice des latences (donnée par les constructeurs) et des bandes passantes (mesurées), on valide la symétrie et la hiérarchie de la machine telle que décrite par hwloc au niveau des mémoires. On observe une correspondance entre les deux matrices pour les liens inter-processeur : lorsque la latence augmente, la bande passante diminue. Cependant l'écart de bande passante au sein d'un même processeur est mitigé au contraire de la latence (théorique).

En conclusion, on constate que la perspective (source,destination) des accès aux données dans la mémoire principale impacte négativement la performance des accès mémoire lorsque la distance des accès augmente.

2.2.6 Témoins de la localité

Jusqu'ici, la performance des mémoires est déterminée avec des bancs d'essai minutieux, en faisant des suppositions basées sur notre connaissance du fonctionnement de la machine et de l'état de l'art des techniques de mesure de performance. Cependant, les processeurs, commutateurs réseau, carte mère, *etc.* disposent de compteurs matériels pour observer le niveau de localité d'une application relativement à la machine. De tels compteurs permettent donc de quantifier l'utilisation des différents niveaux de mémoire par une application. Parmi la quantités de compteurs disponibles sur une plate-forme x86, on trouve par exemple :

- `ix86arch::LLC_REFERENCES` : compte chaque requête du cœur pour accéder à une ligne de cache dans le dernier niveau de cache, *i.e.* L3[1]. Plus le compte est élevé, moins les données sont locales aux niveaux de mémoire inférieurs et donc aux cœurs.
- `ix86arch::LLC_MISSES` : compte le nombre de défauts de cache dans le dernier niveau de cache lors d'un accès à une référence mémoire[1]. Plus le compte est elevé et moins les données sont proches des cœurs.
- `perf::PERF_COUNT_SW_PAGE_FAULTS` : compte le nombre de défauts de pages lorsqu'un accès à une adresse mémoire ne figure pas dans la table

des pages du système d'exploitation. Une valeur élevée de ce compteur est caractéristique d'une forte emprunte mémoire car ce genre de défaut arrive généralement lors de la première écriture d'une page ou lorsque l'utilisation de la mémoire excède la place disponible.

- `perf::PERF_COUNT_SW_CONTEXT_SWITCHES` : Compte le nombre de fois ou l'ordonnanceur exécute un autre thread sur le cœur concerné. À chacun de ces changements de contexte, le nouveau fil d'exécution doit recharger ses données dans les caches car elles ont été évincées par les autres tâches qui se sont exécutées sur le cœur.
- `perf::L1-DCACHE-LOADS` : compte le nombre de requêtes en lecture pour une donnée dans le premier niveau de cache. Les évènements comptés par ce compteur sont proche du nombre d'instructions d'accès à la mémoire par le cœur. Ce compteur ne donne pas l'information si la donnée se trouvait effectivement dans le cache ou non.
- `MEM_LOAD_UOPS_L3_MISS_RETIRED` : compte le nombre de défauts de cache dans le dernier niveau de cache, causés uniquement par l'application. Ce compteur peut être programmé pour compter uniquement si la donnée a été trouvée dans une mémoire distante, ou la mémoire locale, ou encore si elle a pu être trouvée dans le cache d'un autre cœur.
- `MEM_LOAD_UOPS_L3_HIT_RETIRED` : compte le nombre de requêtes de l'application pour une référence mémoire servie par le dernier niveau de cache. Ce compteur peut être programmé pour ne compter que si la donnée n'a pas (ou a) pu être interceptée sur le réseau entre les cœurs de ce cache.

La liste présentée ici n'est pas exhaustive. Il existe beaucoup de compteurs (de l'ordre du millier [58]) sur les processeurs, et les valeurs comptées sont extrêmement bas niveau (c'est-à-dire difficile d'accès et de compréhension pour les non-experts) et spécifiques aux architectures de processeurs. De plus, on ne peut utiliser que peu de compteurs simultanément. Selon les processeurs, le compte varie, sur l'architecture utilisée ici en exemple (Intel(R) Xeon(R) Gold 6140), on ne peut programmer que 4 compteurs par cœur, sachant que les valeurs comptées sont la plupart du temps spécifiques au cœur. Les compteurs peuvent provenir de plusieurs sources. Parmi les exemples cités, le compteur `MEM_LOAD_UOPS_L3_HIT_RETIRED` compte des évènements matériels, alors que le compteur `perf::PERF_COUNT_SW_CONTEXT_SWITCHES` compte des évènements enregistrés par le système d'exploitation. Il existe des interfaces d'abstraction [83, 77] visant à simplifier l'utilisation des compteurs matériels. Performance Application Programming Interface (PAPI) [83] permet de regrouper des évènements de différentes sources sous une interface cohérente, et propose

1. Le compte inclus l'exécution spéculative des cœurs mais pas le chargement de lignes de cache due au pré-chargement dans les caches inférieurs.

des fonctionnalités comme le multiplexage d'évènements pour contrebalancer les limitations tels que le nombre de compteurs programmable par cœur. PAPI abstrait les mécaniques de collectes d'évènements, et les évènements eux même pour viser une portabilité et simplicité d'utilisation relatives, entre les systèmes de calcul. PAPI et Maqao [77] sont construits au dessus d'autres interfaces qui permettent d'accéder à des évènements matériels, tels que perf [25], perfmon, libpfm [61], *etc.* Il est toutefois possible de programmer les compteurs en espace utilisateur [4] via certaines instructions du micro-code du processeur, ou via l'écriture dans certains fichiers du pseudo système de fichier `/proc`, `/sys`.

4. Certains compteurs nécessitent des privilèges administrateur pour être programmés.

Chapitre 3

Localité des données

Améliorer la localité des données consiste à mettre en place des mécanismes qui permettent de diminuer en moyenne le temps passé par les cœurs à attendre les données, et logiquement d'améliorer les performances des applications. La consommation énergétique peut aussi être dégradée à cause du temps d'inactivité (du à l'attente des données) des ressources de calcul et de la mémoire [112, 65], ou bien au déplacement des données utilisant des ressources (mémoire, réseau) supplémentaires [105]. Donc la localité des données est bénéfique à la fois sur les aspects de la performance et de la consommation énergétique [68].

Dans ce chapitre nous formalisons l'expression logicielle de la localité en mémoire partagée. Nous exprimons le lien entre l'implémentation logicielle et la performance matérielle. Enfin, nous faisons l'état de l'art des techniques d'optimisation de la localité en mémoire partagée, et exposons les problèmes ouverts traités dans cette thèse.

On peut distinguer une localité logicielle et une localité matérielle des données. En effet, le logiciel est écrit dans un langage portable (indépendant de la machine), alors que la machine implémente une politique opaque de mouvement de données, indépendante de l'application, à travers une multitude de sous-systèmes. La performance et les mécanismes des différents niveaux de mémoire sont abordés au Chapitre 2. Dans ce chapitre, nous nous intéressons à l'expression logicielle de la localité et son lien avec le matériel. En effet, il existe une relation entre la façon dont le langage exprime la localité et le temps d'accès aux données dans le matériel. C'est d'ailleurs cette relation que nous exploitons pour caractériser la performance du matériel (Chapitre 2). En revanche, on ne peut généralement pas déduire la position d'une donnée dans la hiérarchie mémoire avec une analyse statique du code. À l'exception de l'allocation et de la migration explicite des pages dans la mémoire globale, il n'y a pas de correspondance exacte entre la manière d'exprimer localité dans le logiciel et celle dans le matériel. L'implémentation matérielle des politiques de déplacement des données n'est pas publique et elle est suffisamment complexe pour disqualifier des optimisations en temps réel à cette granularité, lorsqu'une application complexe fait intervenir une multitude de sous-systèmes dans le matériel. Néanmoins, il est nécessaire d'appréhender de manière concrète la dualité de la localité pour en déduire des mécaniques d'optimisation abstraites.

La localité matérielle est la position relative des données par rapport aux cœurs qui les accèdent. La localité logicielle est l'utilisation de l'abstraction qui agit sur la localité dans le matériel. Elle existe principalement à deux niveaux : le schéma séquentiel d'accès aux données, et le partage des données, induits par l'utilisation d'un espace d'adressage plat et partagé. Le premier niveau impacte l'efficacité de l'utilisation d'une mémoire en particulier, ou l'utilisation du cache (localité verticale). Le second niveau à un effet sur le routage des données et la cohérence dans le matériel (localité horizontale).

3.1 Schéma d'accès aux données

Les données accédées dans les applications prennent la forme d'une plage linéaire d'adresses où sont stockées les valeurs lues et écrites à la granularité de l'octet. Il n'y a pas de contraintes concernant l'ordre et la quantité d'accès, on peut même lire et écrire des segments mémoire disjoints obtenus de plusieurs allocations. De plus les adresses virtuelles sont accessibles depuis n'importe quelle entité de calcul. En mémoire partagée, la localité dérive de la manière de lire et écrire les données, *i.e.* par déréférencement d'adresses. Ce paradigme est également parfois repris à l'échelle des systèmes à mémoire distribuée [87], et donc la recherche d'une interface pour abstraire la localité sur un espace d'adressage plat revêt un intérêt même à l'échelle la plus large. L'espacement (en octets) et la fréquence des accès aux données (localité temporelle), im-

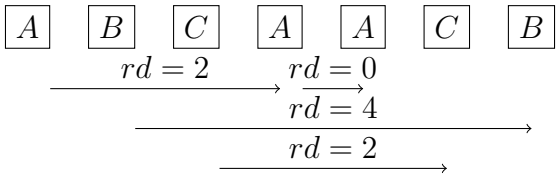


FIGURE 3.1 – Illustration de la distance de réutilisation des données (rd) entre trois données A, B et C.

pactent l'utilisation d'une mémoire en particulier, ou la position de la donnée dans les mémoires et donc la performance.

3.1.1 Localité temporelle

La notion de distance de réutilisation des données [36] peut directement se substituer à celle de la localité temporelle. Elle est définie pour chaque accès à une donnée comme le nombre d'accès à d'autres données jusqu'à un nouvel accès à cette première donnée. Par exemple, on illustre en Figure 3.1 la distance de réutilisation des 4 premiers accès à 4 données pour un schéma d'accès hypothétique. La localité temporelle est directement associée à l'affinité avec le cache, car elle permet d'associer à chaque donnée, une date de dernier accès que l'on peut faire correspondre à la politique LRU, et une fréquence d'utilisation (distance de réutilisation moyenne) que l'on peut faire correspondre à la politique LFU du cache. En effet, plus une donnée est accédée fréquemment (faible distance de réutilisation moyenne), plus elle a intérêt à être proche du cœur contrairement aux données moins souvent accédées, en concurrence pour l'espace disponible.

Dans les machines multicœurs, les données peuvent être accédées de manière concurrente et les caches peuvent être partagés, ce qui influe sur la fréquence de réutilisation des données dans la hiérarchie de caches. On a donc également une notion de distance de réutilisation concurrente [102] adaptée aux machines parallèles pour quantifier la localité temporelle. La mesure logicielle de la distance de réutilisation des données d'une application est très coûteuse car elle nécessite d'instrumenter chaque accès à la mémoire, et de garder en mémoire toutes les adresses accédées, de manière ordonnée. En somme, cette mesure nécessite de simuler un cache de taille au moins équivalente à l'espace occupé par l'application, ce qui n'est pas faisable en temps réel.

3.1.2 Localité spatiale

La localité spatiale est définie par la distance entre les accès consécutifs aux adresses virtuelles, *i.e.* l'espace (en octets) et la direction des accès consécutifs. Elle est optimale quand les adresses sont accédées de manière continue et

contiguë, *i.e.* avec un espace régulier et minimum dans le sens croissant des adresses. La localité spatiale a principalement 3 effets sur le matériel :

- La prévisibilité des accès : Est-ce que les accès pourront être anticipés par le matériel (utilisation implicite du préfetcheur matériel) ?
- L'optimisation de l'utilisation de la mémoire adressable : parallélisme des bancs, pipeline de chargement des lignes, *etc.* ,
- L'exploitation de l'associativité du cache qui permet d'alterner entre les différents ensembles de lignes de cache, et donc de maximiser l'espace effectivement utilisable.
- L'accès des données à la granularité d'une ligne de cache complète rentabilise toutes les données chargées par le déplacement des lignes, il permet également de charger les octets suivants (dans la ligne de cache), en même temps que les octets demandés (au début de la ligne de cache).

De manière générale, le matériel est optimisé pour des accès continus régulièrement espacés.

3.1.3 Placement explicite des pages dans la mémoire globale

Les localités spatiales et temporelles ont un effet implicite sur la position des données dans le matériel. Cependant, au niveau de la mémoire globale, on peut placer les données explicitement, en tenant compte par exemple de la localité spatiale et temporelle pour améliorer la localité dans le matériel. L'allocation des pages virtuelles dans des mémoires physiques particulières peut être implicite ou explicite. Généralement les systèmes d'exploitation Linux implémentent dans le noyau 4 politiques accessibles par la bibliothèque `libnuma` :

- première écriture (*firsttouch*) : Les pages sont allouées dans la mémoire libre la plus proche du premier cœur qui lit ou écrit la page.
- entrelacé (*interleave*) : On associe à un segment de pages un ensemble de nœuds. Puis on place les pages en ronde sur ces nœuds. En Figure 2.4, nous montrons comment le schéma d'accès aux données et la politique de placement des données peu impacter la localité et donc la performance. Les accès contiguës aux différentes pages du segment de mémoire virtuelle alloué induisent des accès à tous les nœuds de mémoire physique.
- fixe (*bind*) : Les pages sont allouées dans la mémoire explicitement choisie si celle-ci est libre, sinon l'allocation échoue.
- équilibrage NUMA (*NUMA balancing*) : Le système d'exploitation implémente une stratégie de migration automatique des processus et des pages. [1]

1. NUMA balancing peut-être activé sous Linux par la commande `echo 1 > /proc/sys/kernel/numa_balancing`.

Firsttouch est celle par défaut. Le système d'exploitation permet également de migrer à la demande des pages déjà allouée vers d'autres nœuds NUMA avec un appel système. Le logiciel peut donc implémenter des stratégies spécifiques ou indépendantes du matériel, à la granularité des pages, pour optimiser la localité des données dans l'espace mémoire adressable. Cette optimisation tient compte du schéma d'accès aux données et du partage des données.

En conclusion, le schéma d'accès spatial et temporel, conjointement à la politique de placement des données est en lien avec plusieurs mécanismes du processeur et de la mémoire, et influence la performance des applications.

3.2 Partage de données

Le partage des données, *i.e.* lectures/écritures concurrentes d'une plage d'adresses communes entre les fils d'exécution, impacte matériellement la cohérence (égalité de l'information identifiée par une adresse virtuelle existante à plusieurs endroits de la machine), la présence en cache des données (distance de réutilisation concurrente), la concurrence pour les ressources partagées, *etc.* Il nous faut formaliser le partage de données dans le logiciel, pour le modéliser dans le matériel et essayer de faire le lien entre le code et la performance à l'exécution.

3.2.1 Identification des tâches et des données

Le partage de données s'exprime selon le lien entre les entités qui accèdent aux données et les données elles-mêmes. On a besoin d'identifier une donnée (adresse, taille) et des entités accédant aux données pour exprimer le partage de données. Comme matériellement la plus petite unité de donnée déplacée est la ligne de cache, on choisit cette unité (64 octets dans la plupart des machines contemporaines) pour identifier une donnée. Dans certains contextes, on peut préférer une métrique plus grossière, à l'instar des pages (4096 octets) qui correspond logiciellement à la granularité de la gestion de la mémoire dans le système d'exploitation.

Nous définissons la granularité la plus fine d'accès aux données comme un morceau séquentiel d'application. Logiciellement il peut s'agir d'une fonction, d'un bloc de code sans branchement, d'une instruction, *etc.* Nous appelons cette entité une tâche. La plupart du temps les tâches sont implémentées dans des bibliothèques utilisateur par des *threads*, qui permettent d'exécuter une fonction de manière asynchrone, et qui partagent le même espace mémoire. Dans ce manuscrit, lorsque ce n'est pas précisé, nous utilisons indifféremment le terme de tâche ou de thread).

Nous associons chaque thread à un cœur, c'est à dire que le travail effectué par un thread le sera sur un seul cœur. Un paradigme de parallélisme populaire, OpenMP [23], fonctionne par la création d'équipes de threads. Les threads peuvent être directement associés à des cœurs, de manière statique, *i.e.* pour la durée de l'application, via une variable d'environnement. Donc cette hypothèse de correspondance entre les tâches et les unités de calcul fait sens par rapport aux paradigmes de programmation parallèle usuels. Les paradigmes de programmation en taĉhes dynamiques comme StarPU [4], OpenMP, ou Parsec [12], distribuent un graphe orienté acyclique de travail, crée dynamiquement à l'exécution sur les cœurs de la machine. Avec de tels modèles, le placement des tâches et le déplacement des données est gérée automatiquement par le support d'exécution grâce au paradigme qui défini de manière explicite les tâches et les transferts de données. Dans un tel paradigme, il n'y a pas de correspondance pré-établie entre le travail et les cœurs, et entre les données et les mémoires, qui doivent être déterminées dynamiquement avant l'exécution des tâches. Par conséquent, la compréhension des paramètres qui font une bonne localité est aussi nécéssaire dans ce paradigme pour prendre des décisions d'ordonnancement.

On s'intéresse au problème du placement statique, c'est-à-dire que l'on considère qu'on ne peut pas gagner à migrer dynamiquement à l'exécution les tâches et/ou les données. La migration de thread d'un cœur à un autre, dans le cas où l'on exécute un seul thread par cœur, avec un travail déterministe pour la durée de l'exécution, peut être bénéfique [14]. Cependant, elle implique aussi de devoir reconstruire l'état du cache et du TLB pour un autre cœur, et donc il y a un compromis à satisfaire entre les bénéfices et les sacrifices impliqués par la migration, dont on s'affranchie.

On peut simplifier le problème, en le décomposant en deux temps : une phase statique pendant laquelle on cherche un placement optimal des threads, et un changement de phase qui nécessite ou non de changer le placement. Dans tous les cas, on ne peut exécuter qu'une seule tâche (ou un seul travail dans le cas d'un graphe de tâches) par cœur et par tranche de temps. Donc, à une certaine granularité, le problème est statique, mais une solution plus générale doit aussi résoudre le problème de la migration.

3.2.2 Partage de données en mémoire partagée

Pour modéliser logiciellement la localité on construit le graphe biparti (Figure 3.2) dont les sommets représentent d'un côté des tâches et de l'autre des données, et dont les arêtes symbolisent les accès des tâches aux données et sont pondérées en nombre d'accès. Le partage d'une donnée entre deux tâches est la présence de deux arêtes connectant deux tâches distinctes par cette donnée. La quantité de partage de cette donnée entre ces deux tâches est la valeur minimum des accès à la donnée par ces mêmes fils.

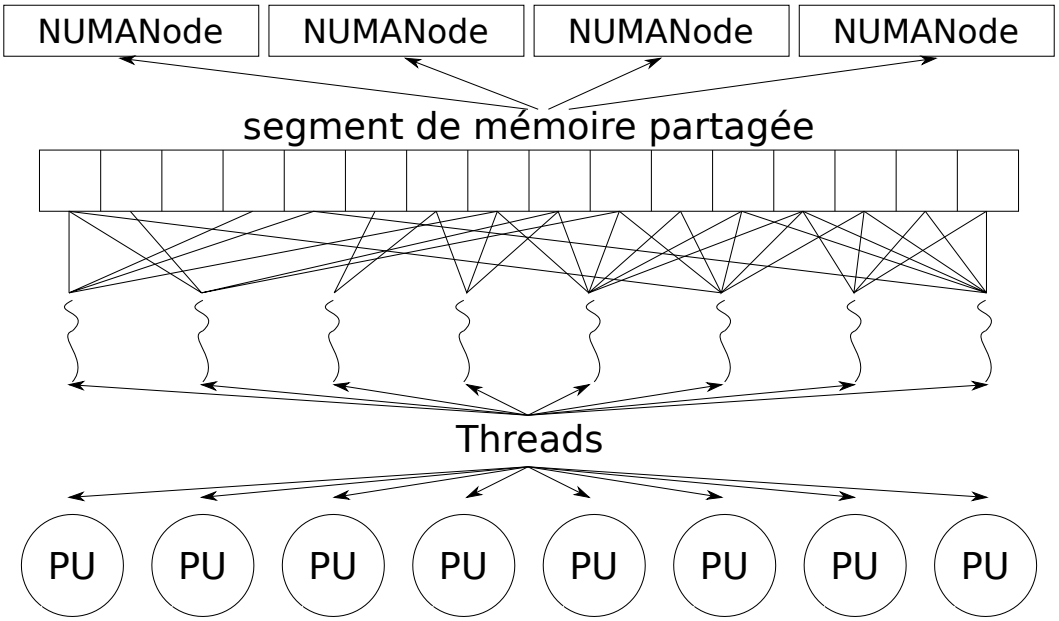


FIGURE 3.2 – Schéma d'association threads sur les cœurs, des données sur les mémoires et des accès des threads aux données. Ces derniers forment un graph biparti entre les threads et les données.

À partir du graphe en Figure 3.2, on peut définir des métriques sur le partage comme le degré de partage de chaque donnée, *i.e.* le nombre d'arêtes connectées à chaque donnée, ou le partage moyen de l'application comme la moyenne des degrés de partage de données. Ces métriques ont pour but d'aiguiller les décisions de placement. Puis, de ce graphe biparti, on peut construire un graphe de partage en le contractant pour ne garder que les tâches, connectées selon leur quantité de partage. Diener et *al.* proposent des métriques haut niveau [35], à partir de la matrice M de partage entre T tâches que nous avons définie et normalisée, afin de déterminer la sensibilité des applications au placement de leurs tâches.

L'hétérogénéité C_H (3.1) du partage, quantifie la variation moyenne entre chaque paire de threads, du partage de leurs données, en comparaison du partage avec les autres threads. L'hétérogénéité augmente lorsque les threads ont plutôt tendance à communiquer beaucoup avec quelques threads et très peu avec les autres, et diminue lorsque les communications sont plutôt homogènes, *i.e.* de valeur équivalente pour tous les threads. La quantité de partage C_A (3.2) quantifie la moyenne de partage normalisée. C_A est grand lorsque les threads communiquent tous de manière homogène, et est petit si peu de paires de threads communiquent beaucoup plus que les autres.

$$C_H = \sum_{i=1}^{T} \sum_{j=1}^{T} \frac{\left(\frac{\sum_{k=1}^{T} M_{i,k}}{T} - M_{i,j}\right)^2}{T^2} \tag{3.1}$$

$$C_A = \frac{\sum_{i=1}^{T} \sum_{j=1}^{T} M_{i,j}}{T^2} \tag{3.2}$$

Cependant, le partage logiciel de données n'implique pas directement des conséquences dans le matériel. En effet, comme décrit en sous-section 2.2.3, une donnée partagée peut être lue avec une bande passante équivalente pour les threads si elle se trouve dans une mémoire partagée. De plus, si une donnée partagée, située dans le cache privé du cœur qui la lit, est à jour lors de la lecture, alors il n'y a pas nécessité de faire intervenir les mécanismes de cohérence. Donc, le fait que la donnée soit partagée ou non n'a pas nécessairement d'incidence sur la performance de ces accès. En revanche, en cas d'écriture, il faut envoyer un message d'invalidation aux autres caches qui contiennent la donnée. Donc, on peut utiliser une autre définition caractéristique du partage, celle de communication en mémoire partagée, qui est plus proche des implications matérielles du partage de données.

3.2.3 Communications implicites en mémoire partagée

On peut définir une communication en mémoire partagée, entre un lecteur et le dernier écrivain, comme la première lecture du lecteur, suivant l'écriture. Cette définition de communication fait echo au protocole de cohérence qui invalide les données partagées dans les caches lors d'une écriture, et qui récupère la valeur de la donnée à la lecture suivante (*cf.* sous-section 2.2.3). Pour toutes les lectures subséquentes dans le même cache, tant que la donnée reste valide, il n'y a pas nécessité de transférer la donnée d'un cache à un autre. Elle tient compte des échanges de données lorsque celle-ci passent de l'état *Invalid* à *Shared* (*cf.* protocoles de cohérence : sous-section 2.2.3), *i.e.* lorsque la donnée nécessite d'être mise à jour à sa valeur la plus récente à partir d'un autre cache. En revanche, elle ne prend pas en compte l'envoi du message de cohérence à l'écriture si la donnée est dans l'état *Shared* et qu'elle passe à l'état *Invalid* pour les autres caches, *i.e.* lorsqu'il faut envoyer un message de pour invalider les copies non à jour dans les autres caches. Pourtant, le trafic de cohérence peut être responsable d'une occupation significative de la bande passante du matériel [15], et risque donc de diminuer le débit du transit des données de l'application. Enfin cette définition, comme celle du partage, ne tient pas compte de la distance de réutilisation des données. Pourtant, lorsqu'une donnée est dans un cache partagé, il n'y a pas vraiment de surcoût lié au partage ou à la cohérence (*cf.* sous-section 2.2.3). Donc, les données peu réutilisées, qui remontent dans des mémoires partagées ont moins d'importance dans le calcul du placement, et ne devraient pas être comptées comme partagées.

Cruz et *al.* utilisent ainsi le protocole de cohérence pour détecter les communications [21]. Les données partagées valides sont dans l'état *Shared* du protocol MOESI et à chaque écriture un message d'invalidation est envoyé aux cœurs qui partagent cette ligne de cache pour la mettre dans un état *Invalid*. Pour reconstruire la matrice (ou graphe) de communication, ils proposent de stocker dans le premier niveau de cache un vecteur de partage, de taille équivalente au nombre de cœurs, et dont les éléments sont incrémentés à chaque réception d'un message d'invalidation de l'écrivain. Chaque vecteur de communications propre à un cœur serait accessible par le système d'exploitation et permettrait de reconstruire la matrice de communications. L'avantage de cette méthode est que son coût est faible, et qu'elle tient compte de la temporalité. L'inconvénient est qu'il faut changer le matériel pour la mettre en œuvre.

A l'instar du schéma d'accès aux données, le partage et les communications en mémoire partagée sont en lien avec plusieurs mécanismes de déplacement des données dans le matériel, *e.g.* , cohérence, réutilisation dans le cache, *etc.* Il est donc nécessaire de prendre en compte la façon dont l'application partage ses données pour placer ses tâches de manière à optimiser le temps d'exécution.

3.3 Stratégies d'optimisation de la localité

Une bonne localité, logicielle ou matérielle, ne garantit pas une meilleur performance. La compétition pour les ressources partagées [32] telles que l'espace des caches, les contrôleurs mémoire, *etc.* privilégie d'autres stratégies basées sur l'équilibrage de charge et l'arbitrage de l'accès aux ressources. Il existe d'ailleurs un compromis entre la localité et la concurrence [76, 33].

3.3.1 Politiques centrées sur la localité

Si l'on suppose qu'une communication (comme défini en Section 3.2) induit un transfert de données entre deux cœurs dans la machine, alors le graphe de communication dont les sommets sont des tâches et les arêtes des quantités de communications est une approximation des transferts de données d'une application entre ses tâches de calcul (associées à des coeurs). Dans certains paradigmes comme MPI, ou graphe de flots de données, les échanges de données sont explicites et la bibliothèque permet d'extraire la matrice de communications avec un surcoût négligeable. Dans notre cas on peut en obtenir une approximation (avec des communications implicites en mémoire partagée) pour un surcoût qui limite son champ d'utilisation. Compte tenu du graphe de communication entre les tâches d'une application, il peut être désirable de limiter le chemin à parcourir par les données durant les communications. On défini le *hopbyte* comme la quantité de liens dans la machine à traverser par

chaque octet de données de l'application. Le *hopbyte* est donc un indicateur de la localité que l'on souhaite minimiser.

On peut essayer de minimiser le *hopbyte* en coupant récursivement le graphe de partage (ou de communications) de manière à minimiser la coupe [120], *i.e.* la quantité de communications inter-groupe, et en plaçant chaque groupe sur des nœuds interconnectés de la topologie, de sorte que l'on minimise l'utilisation du réseau inter-nœud. Sur des topologies symétriques, on peut la couper en deux et associer à chaque sous arbre un des sous-graphes contenant des tâches. On peut appliquer cette technique récursivement, à tous les étages d'arité paire. Le logiciel Scotch [88] permet de faire ce découpage, même pour les étages dont l'arité n'est pas paire. L'algorithme TreeMatch [63] utilise cette technique avec le partitionneur de graphe Scotch (lorsque nécessaire), pour résoudre le problème à l'échelle distribuée, car certaines topologies réseau de systèmes de calcul à mémoire distribuée prennent la forme d'un arbre de type fat-tree [70]. De plus, l'algorithme de TreeMatch s'adapte aux cas où la machine n'est pas symétrique, ou encore si le nombre de threads est différent du nombre de cœurs. Les méthodes de calcul pour l'association des threads aux cœurs effectuées en direct, pendant l'exécution, nécessitent d'anticiper si le recalcul du placement n'est pas nécessaire de sorte que le coût de l'optimisation ne soit pas supérieur au gain. À cette fin, Cruz et *al.* [22] utilisent une méthode qui tient compte de l'aspect dynamique du schéma de communication et implémentent une détection de phases des applications avant de recalculer le potentiel pour une nouvelle association des threads aux cœurs. De plus, l'algorithme d'association utilisé est moins coûteux que TreeMatch car il doit être calculé en concurrence avec l'exécution dont on doit déterminer le placement. En effet, il faut que le calcul d'une solution de placement prenne moins de temps que le gain obtenu par la solution.

ForestGOMP [14] utilise le parallélisme imbriqué des régions parallèles de OpenMP pour automatiquement créer des groupes hiérarchiques de tâches dont l'affinité et le partage sont supposés induits par le paradigme. La hiérarchie de tâches est finalement associée à la hiérarchie des ressources de la machine, en prenant en compte le schéma d'accès aux données pour optimiser la localité, voire en migrant des données vers des mémoires proches.

Certaines approches [43] adoptent un point de vue centré sur les données et gèrent le placement des données de manière indépendante du placement des tâches avec des statistiques sur les pages obtenue dynamiquement à l'exécution via des compteurs d'échantillonnages d'instruction en tenant compte notamment de l'utilisation du cache par l'application, du nombre d'accès à la mémoire locale / distante, de la latence des accès à une page, de la bande passante utilisée sur le réseau d'interconnexion des nœuds NUMA, de l'équilibre des accès sur les contrôleurs mémoire, *etc.* Carrefour [43] implémente une stratégie de rapatriement, réplication, répartition des données, afin respectivement d'améliorer la localité des accès exclusifs, ou la localité des données en lecture

seule, et de diminuer la contention sur les contrôleurs mémoire.

3.3.2 Réduction de la contention

Jusqu'ici, les stratégies d'optimisation décrites mettent l'emphase sur le partage des données pour grouper les threads et déplacer les données. À l'image du partage, la contention est un critère déterminant dans le placement des tâches. L'ordonnancement simultané de plusieurs tâches ralentit l'exécution globale [2], et les stratégies d'ordonnancement des applications ne sont pas équivalentes et peuvent coûter de l'ordre de 20% du temps d'exécution entre la meilleure et la plus mauvaise [124].

La concurrence pour des ressources partagées agit directement sur le partage de données dans le matériel. En effet, par exemple, deux tâches en concurrence pour de l'espace en cache partagé voient un espace effectif de cache réduire de moitié. L'espace disponible ainsi réduit impose à une partie des données de résider plus loin des cœurs, dans des niveaux éventuellement partagés avec plus d'unités de calcul. Donc la contention réduit l'intérêt du placement de tâches par rapport au partage de leurs données.

L'utilisation d'applications séquentielles (n'utilisant qu'une seule unité de calcul) en concurrence plutôt que d'applications parallèles permet d'isoler le problème de contention par rapport au partage qui est inclus dans le second cas.

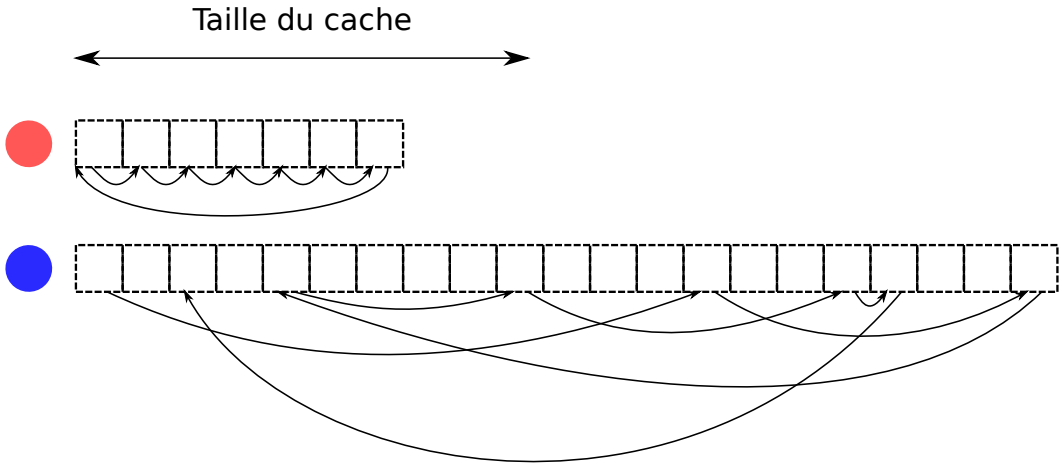
La contention peut s'exercer à tous les niveaux de ressources partagées, *e.g.* caches, contrôleurs mémoire, mémoires, réseau intégré au processeur, *etc.* Au niveau du cache, les politiques de type *Least Frequent Recently Used* (remplacement du plus ancien des moins utilisés) ne sont pas optimales. Par exemple, elle peut créer des interférences entre les données de chaque cœur qui écrasent successivement les données les uns des autres et annulent le bénéfice du cache. Dans un tel cas, des stratégies récentes basées sur le partitionnement de cache [95] pourraient permettre de réduire le problème. Cependant, les problèmes de contention sont sujets à une interaction complexe qui n'est pas nécessairement soluble par le partitionnement de ressources matérielles, mais peuvent l'être par le placement. Zhuravlev et *al.* [124] proposent d'étudier une solution à ce problème en deux étapes : la classification des threads et l'algorithme d'ordonnancement qui tient compte de cette classification. Ils identifient le nombre de défauts de cache des applications (seules sur le système) dans le LLC comme le paramètre prépondérant à la classification des applications, par rapport à l'objectif qui est de minimiser la pénalité qu'elles subissent en partageant les ressources et parviennent à atteindre à 2% près leur définition de d'ordonnancement optimal, en ordonnançant plusieurs applications.

2. En Section 2.2, nous montrons que la performance parallèle des composants mémoire ne passe pas à l'échelle des cœurs qui peuvent les utiliser.

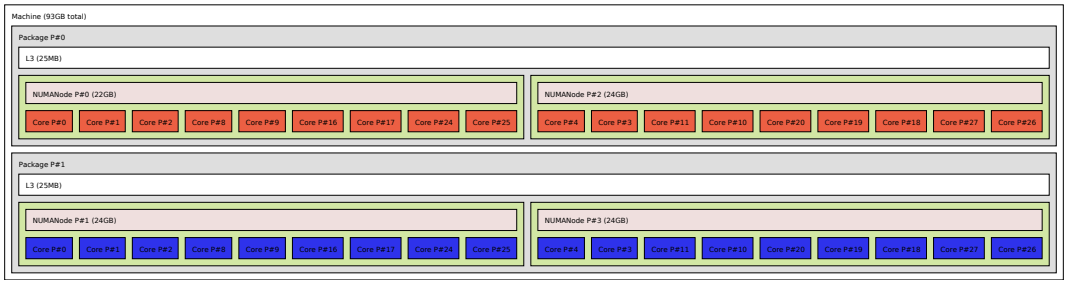
On peut vérifier qu'il existe un lien entre la concurrence pour l'espace en cache et la quantité de défauts de cache, et le temps d'exécution. Avec la méthodologie décrite en sous-section 2.2.1, on peut parcourir arbitrairement des données dans un cache ou à l'extérieur d'un cache. Sur la machine en Figure 2.5, on instancie deux groupes de 18 threads, dont le premier parcourt des données dans le cache partagé, et le second parcourt des données dans la mémoire principale (Figure 3.3a). À partir de ce contexte, on observe deux scénarios, le premier « équilibré » (Figure 3.3c) où les threads du premier et second groupe sont répartis sur la machine pour minimiser la contention et un « regroupé » (Figure 3.3b) où le premier groupe de threads partage un cache, alors que le second partage l'autre cache. Le nombre d'éléments parcourus est ajusté dans chaque groupe de sorte que dans le scénario « équilibré », les threads de chacun des groupes ont à peu près le même temps d'exécution. Le premier groupe ne fait théoriquement pas de défauts de cache (sauf si il y a de la concurrence) alors que le second groupe fait théoriquement exclusivement des défauts de cache. Donc si on observe les compteurs de défauts de cache, et de nombre d'accès au cache, on devrait trouver (à la concurrence près) que le premier groupe crée peu de défauts de cache en comparaison du nombre d'accès au cache, alors que pour le second groupe, les ordres de grandeur de ces deux compteurs sont équivalents. Au résultat (Table 3.1b), dans le scénario équilibré (*i.e.* avec le moins de contention), le groupe bleu fait 10 fois plus de d'accès au LLC que de défauts alors que le groupe rouge en fait 100 fois plus, ce qui est conforme à nos attentes. On voit que la stratégie qui fait le moins de défauts de caches est aussi la plus rapide (Table 3.1a). Lorsque les threads qui concourent pour le cache sont regroupés, le nombre de défauts de cache pour ce groupe est multiplié par 10 (Table 3.1b). Donc il y a une corrélation, entre la concurrence, le nombre de défauts de caches et le temps d'exécution. On remarque également que la variance de défauts dans le LLC du groupe rouge est élevée dans le cas « regroupé », ce qui montre que certains threads ont été avantagés par la politique du cache, car ils font beaucoup moins de défauts que les autres. Cette variance vient impacter les threads du groupe bleu dans le cas équilibré, ce qui montre la propension des threads du groupe rouge à perturber leurs voisins. Cet exemple est réutilisé au Chapitre 5 pour démontrer l'intérêt d'associer les modèles d'application avec la topologie du système.

3.4 Problèmes ouverts de la localité des données

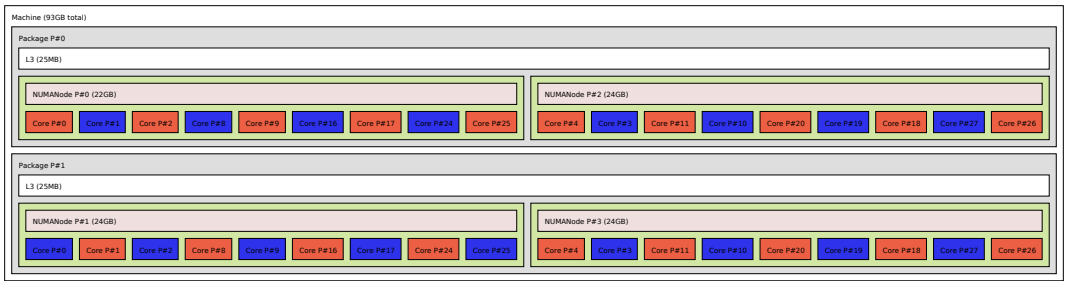
Il existe plusieurs stratégies d'optimisation de la localité, basées sur le partage, la contention, ou un compromis de ces derniers. Elles ont en commun de manquer d'un lien quantitatif avec les performances de la machines. **Les solutions existantes se comparent rarement à une borne supérieure des performances atteignables**, et essaient encore moins d'expliquer cette diffé-



(a) Exemple de listes chaînées.



(b) Placement « regroupé »



(c) Placement « équilibré »

FIGURE 3.3 – Placement des threads qui parcourent des listes chaînées : longues avec chaînage aléatoire en bleu, et courtes avec chaînage contigu en rouge.

politique	ms	défaut LLC	accès LLC	var (ms)	var (défaut LLC)	var (accès LLC)
regroupé	2.2e+01	1.2e+05	2.5e+06	2.1e+00	1.1e+05	1.1e+06
équilibré	2.0e+01	8.8e+04	8.1e+05	3.6e+00	8.6e+04	6.3e+05

(a) LLC accès/défauts accumulés par politique de placement.

politique	groupe	ms	défaut LLC	accès LLC	var (ms)	var défaut LLC	var accès LLC
regroupé	bleu	2.0e+01	2.1e+05	1.4e+06	2.3e-01	2.1e+02	8.3e+02
regroupé	rouge	2.4e+01	2.9e+04	3.5e+06	3.4e-01	9.4e+04	1.1e+05
équilibré	bleu	1.7e+01	1.7e+05	1.4e+06	4.4e-01	4.4e+03	1.5e+03
équilibré	rouge	2.4e+01	3.1e+03	2.1e+05	1.4e-01	4.4e+03	2.4e+05

(b) LLC accès/défauts accumulés par politique et par groupe.

TABLE 3.1 – Défauts et accès dans le dernier niveau de cache pour plusieurs scénarios de placement de threads parcourant des ensembles de données avec des schémas d'accès visant ou non à occuper le LLC.

rence. **Les solutions existantes sont rarement couplées à la structure du système.** Certaines propositions se basent sur une connaissance abstraite du système, sans analyser les différents compromis qui entrent en jeu, et qui nécessitent d'intégrer la topologie du système aux modèles de performance. **Enfin, il n'existe pas de solution généraliste ou de consensus implémenté dans un support d'exécution.** Les optimisations proposés le sont souvent dans un cadre particulier, *e.g.* collocation d'applications, applications sensibles au placement sur les mémoires, *etc.* Mais on voit plus rarement des études exhaustives sur les critères identifiés dans ce chapitre. Nous traitons donc ces problèmes dans l'ordre suivant :

1. Modéliser la performance de la machine et ses goulets d'étranglement relativement à la localité des données

Les algorithmes d'optimisation de la localité utilisent un modèle hiérarchique de la topologie, dont les latences entre les nœuds sont exprimées de manière arbitraire par les constructeurs [3]. Or selon [71] [4], la bande passante de la mémoire est un critère plus important que la latence (ou le nombre de liens de la topologie) pour le placement des tâches et des données. D'où l'intérêt de modéliser les performances de la machine. Wei Wang et *al.* [24] confirment que l'importance de l'asymétrie de la machine et des applications sont des facteurs caractéristiques de la performance des accès mémoire. Donc la topologie de la machine est une pièce essentielle du puzzle de la localité. De plus, la contention [24, 71, 124] est à ajouter à la bande passante parmi les caractéristiques qui bornent la performance des applications. En conclusion, il est important de modéliser les performances et les goulets d'étranglements de la machine

3. La matrice de latences peut-être obtenue par l'interface de la bibliothèque hwloc (Table 2.1a).

4. *« Bandwidth between the nodes matters more than the distance between them »*

relativement à sa structure.

Dans le Chapitre 4, notre première contribution [126, 129] est une extension d'un modèle quantitatif de performances de la machine et des applications. Ce modèle détermine les bornes de performances du système et la distance des applications à ces bornes. Notre contribution inclut notamment la prise en compte de la localité, des bandes passantes, et de la contention sur les contrôleurs mémoire dans le modèle original. Nous définissons par la suite un modèle de bande passante hétérogène, caractérisant la bande passante atteignable lorsque plusieurs mémoires avec différentes caractéristiques sont accédées simultanément et pour plusieurs types d'opérations. Ce modèle est complémentaire du premier et peut y être inclus.

2. Mettre en lien les évènements logiciels avec le matériel pour modéliser le système dans son ensemble

Les systèmes de calcul haute performance sont trop complexes pour espérer les modéliser entièrement, avec les applications qui les utilisent, dans un temps et un budget raisonnable. Cependant, il est clair que le système est décomposé en sous-systèmes critiques, en interaction mutuelle et en interaction avec les applications. Donc, les solutions au placement des tâches et des données doivent tenir compte des composants structurés de la plate-forme et de leur exploitation par les applications. À ce jour, il n'existe pas de solutions unanime ou normalisée pour répondre au problème de la localité dans son ensemble. Étant donné l'accroissement de la complexité des systèmes et les difficultés pour en tirer de bonne performances, poser les bases communes et durables à la localité reste un problème de haute importance. Il faut construire des abstractions et des solutions portables, pour s'adapter d'une génération de machine à l'autre à moindre coût. Pourtant, alors que le problème de la taille des machines est de plus en plus prégnant, il n'existe pas d'outil proposant d'assister le processus de modélisation de la localité pour en comprendre les paramètres plus rapidement.

Une des contributions [125, 127] du Chapitre 5 est un outil de modélisation des applications en relation avec la topologie des systèmes. Nous proposons une méthodologie et son implémentation pour associer des évènements logiciels et matériels à des objets de la topologie matérielle, et de les combiner le long des liens de la topologie pour en extraire une valeur objective du modèle ainsi décrit.

3. Prendre en compte tous les aspects de la localité à la fois et décider de la stratégie à adopter

Devant la complexité des systèmes, la quasi-totalité des approches d'optimisation de la localité se focalisent sur une sous partie du problème, *e.g.* placement des pages [71], collocation d'applications [124], migration de tâches [10],

détermination du bon nombre de ressources de calcul à utiliser [24], optimisation statique du compromis localité/équilibre à l'échelle de la mémoire [33], réduction du trafique de cohérence [15], *etc.*

Dans le Chapitre 6, nous proposons une approche statistique à l'étude des performances des applications relativement au placement de leurs tâches et de leurs données. Nous proposons d'exécuter des applications parallèles représentatives des travaux soumis aux systèmes CHP, avec plusieurs politiques de placement, et de tenter de déduire des liens entre les politiques choisies, les temps d'exécution et les métriques des applications et du matériel.

Chapitre 4

Modèle d'application et de plate-forme à mémoire hétérogène

Afin de résoudre les problèmes de placement des calculs sur les cœurs, et des données dans la mémoire, les développeurs attendent des informations claires par rapport à la performance maximale des sous-systèmes et des goulets d'étranglements de la machine. Le Cache-Aware Roofline Model (CARM) [57] est un modèle qui fourni une évaluation visuelle des performances des applications et de leur distance par rapport à celles des éléments de la machine qui peuvent être limitants. Dans ce chapitre nous décrivons ce modèle ainsi que notre contribution, en collaboration avec les auteurs du CARM, pour y adjoindre des caractéristiques critiques des plates-formes NUMA contemporaines. Enfin, nous proposons un modèle de bande passante hybride, inspiré par une des limitations du modèle, qui décrit la performance de la machine soumise à des requêtes concurrentes vers plusieurs mémoires.

4.1 Modèle de plate-forme et d'application à seuil de performance

Les plates-formes de Calcul Haute Performance (CHP) à mémoire partagée sont de plus en plus complexes avec une hiérarchie mémoire de plus en plus profonde. Afin d'espérer exploiter le potentiel de telles ressources de calcul, il est nécessaire d'optimiser les applications pour s'adapter au mieux à la hiérarchie mémoire d'une machine à une autre. La complexité de cette tâche est croissante avec celle de la complexité des plates-formes. En amont d'un investissement important en ressources humaines pour optimiser une application, il est nécessaire de :

1. évaluer le potentiel de gain de performance de l'application relativement à celle de la plate-forme,
2. localiser les goulets d'étranglement physiques et logiciels responsables de mauvaises performances.

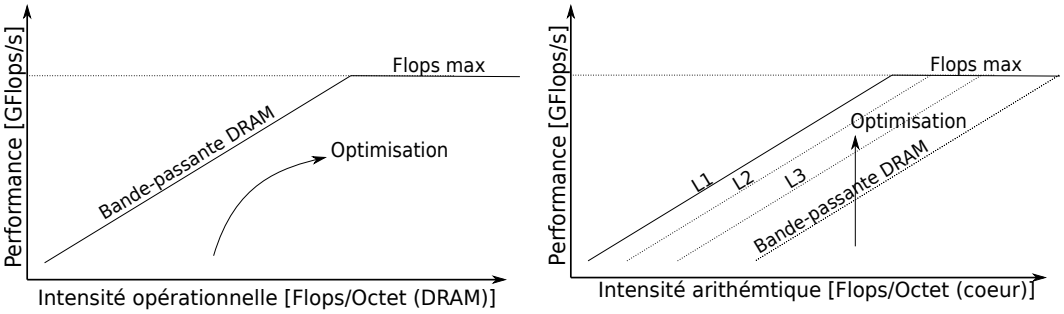
Les différentes évolution du Roofline Model [121, 57, 126] répondent à ces deux problèmes.

4.1.1 Original Roofline Model

Le Roofline Model de Samuel Williams et *al.* [121] modélise les applications comme un flux d'instructions composé de chargements de données depuis la mémoire, de calculs sur les données chargées, et enfin d'écritures des résultats dans la mémoire. De manière similaire, il considère les machines de calcul comme étant principalement composées d'une unité de calcul et d'une unité de mémoire. Ces hypothèses permettent de décrire la machine comme un système de files d'attente dont le débit est limité soit par l'unité de calcul, soit par l'unité mémoire selon que le flot d'instructions d'une application est principalement constitué respectivement d'opérations arithmétiques ou d'accès à la mémoire.

Le ratio d'opérations flottantes (FLOP) exécutées par les unités de calcul, sur la quantité d'accès à la mémoire (en octets) de l'application et servis par la mémoire principale, est nommé intensité opérationnelle (OI). L'intensité opérationnelle est une caractéristique à la fois de la plate-forme et de l'application car la quantité de calculs est une propriété uniquement de l'application tandis que la quantité d'accès à la mémoire dépend de la quantité de données accédées par l'application et des caches qui peuvent économiser des accès à la mémoire principale. Enfin, les performances de la machine (débit de la file d'attente) et de l'application (en FLOPs/s) sont respectivement caractérisées par le débit de calcul de la machine et de l'application sujets à une intensité opérationnelle.

Le Roofline Model original est donc un modèle de plate-forme et d'application qui propose une représentation en deux dimensions des performances des



(a) Roofline Model Original

(b) Cache-Aware Roofline Model

FIGURE 4.1 – Représentation du ORM et CARM pour une plate-forme multi-cœur, non-NUMA, hypothétique et une application hypothétique.

applications et des plates-formes. Le plan décrit par l'intensité opérationnelle en abscisse et la performance en ordonnée est représenté en Figure 4.1a, en échelle logarithmique. Celui-ci est borné verticalement par une droite oblique dont le coefficient directeur est la bande passante de la mémoire globale (non-NUMA) et une droite horizontale dont l'ordonnée est le débit d'opérations flottantes des cœurs.

Une application caractérisée par son intensité opérationnelle et sa performance est représentable dans un tel plan par un point. Selon l'intensité opérationnelle de l'application, celle-ci peut être bornée verticalement par la bande passante (*i.e.* si elle fait beaucoup d'accès à la mémoire par rapport aux calculs), ou bien par le débit de calculs (*i.e.* si elle fait beaucoup de calculs par rapport aux accès à la mémoire). L'intersection de la droite de bande passante avec celle du débit de calcul marque une frontière verticale entre les applications limitées par la performance de la mémoire (memory-bound) et celles limitées par la capacité de calcul du processeur (compute-bound).

Plus récemment, le modèle a été augmenté de la représentation des bandes passantes des différents niveaux de cache comme des droites supplémentaires [73]. Cependant, il reste difficile de faire le lien entre la bande passante de l'application mesurée vers la mémoire principale et celle des caches qui n'est pas caractérisée dans l'intensité opérationnelle. En effet, l'optimisation d'une application relativement à son utilisation de la mémoire peut avoir différentes conséquences observables dans le modèle. Entre autre :

– Les accès à la mémoire sont réorganisés de sorte à améliorer l'utilisation de la mémoire principale. Par exemple, on peut transformer un tableau de structures en structure de tableaux, de manière à accéder des éléments de manière contiguës, et donc avec des accès à la mémoire plus performants. Dans ce cas, le nombre d'accès à la mémoire et le nombre de calculs ne changent pas. Donc le point (caractéristique de l'application) conserve son abscisse et voit sont ordonnée grandir jusqu'à éventuelle-

ment atteindre la borne de bande passante ou de calcul.

– Les accès à la mémoire sont réorganisés d'une manière qui impacte l'utilisation du cache, *e.g.* on interchange les boucles d'un nid de boucles. Dans ce cas l'abscisse de l'application ainsi que son ordonnée changent (car la réutilisation des données dans le cache évite certains accès à la mémoire, alors que la quantité de calcul reste constante, et donc l'intensité opérationnelle change) et la borne supérieure de performance atteignable n'est plus prévisible. Elle reste toutefois bornée par la droite horizontale, mais l'utilité du modèle est plus limitée dans ce cas.

Le Roofline Model original est mis en œuvre par Intel dans son outil propriétaire Advisor [60]. Pour des raisons de portabilité, les FLOPs des applications (éventuellement détaillés par région du code) sont obtenus par une pré-exécution de l'application instrumentée. Cette technique permet de se passer de l'utilisation des compteurs qui peuvent être absents voir non-fonctionnels sur certaines architectures. Les transferts de données sont en revanche nécessairement mesurés à travers des compteurs matériels car aucune instrumentation ne peut parfaitement anticiper le comportement du cache pour déduire les accès réellement effectués vers la mémoire, alors que les compteurs d'accès à la mémoire sont le plus souvent présents et fiables.

Le Roofline Model est un modèle d'application et de machine parallèle qui permet d'évaluer la distance d'une application à certaines bornes supérieures de performance de la machine et de les représenter de manière intelligible pour les développeurs. Cependant, d'une part, la borne décrite par ce modèle peut être imprévisible par rapport à certaines optimisations, et d'autre part, le modèle décrit très peu de bornes au regard de la complexité de la machine et des optimisations à la portée des développeurs, qui visent une meilleur utilisation de la hiérarchie de caches ou bien des mémoires NUMA.

4.1.2 Cache-Aware Roofline Model

Le Cache-Aware Roofline Model[57] est un modèle inspiré du modèle original et qui poursuit les mêmes objectifs. Cependant celui-ci propose un autre point de vue sur la mesure des accès à la mémoire qui sont dorénavant comptés au niveau des cœurs. Le nombre de requêtes pour des données ainsi compté n'est plus une propriété du matériel mais uniquement de l'application. Elle est entre autre invariante aux réarrangements de données visant l'optimisation de l'utilisation de la mémoire ou du cache. L'intensité arithmétique (en opération flottantes (Flops)/Octet) (IA) qui remplace l'intensité opérationnelle est une constante de l'application insensible au schéma d'accès aux données. Dans le CARM toute optimisation du schéma d'accès aux données ne modifie pas l'intensité arithmétique des applications mais modifie leur performance. Selon la position des données dans la hiérarchie de caches, le débit d'instructions mémoire observé par les coeurs varie. La bande passante des caches est croissante

à mesure que l'on se rapproche des cœurs. Donc, le goulet d'étranglement de performance des accès à la mémoire est directement le niveau où se trouvent les données, et pas les niveaux inférieurs dont la bande passante est plus élevée. On peut représenter pour chaque niveau de cache, la bande passante maximum atteignable de ce niveau de mémoire dans le modèle. Cette représentation fait sens, car la mesure de la quantité de données accédées par l'application n'est pas influencée par la position des données dans la machine et reste cohérente quelque soit le niveau de mémoire par rapport auquel elle est comparée, alors que le débit de données lui est relatif à la position des données. Ainsi, comme il est représenté en Figure 4.1b, l'évolution d'une application dans le CARM est uniquement verticale, et permet d'évaluer précisément les distances entre la performance de l'application et celle des différents niveaux (L1, L2, L3, RAM) de la hiérarchie mémoire. Dans cette représentation, le coefficient directeur des droites représentant les différents niveaux de mémoire, est égal à leur bande passante, et l'ordonnée de la droite horizontale correspond toujours au débit de calcul des cœurs.

Enfin, le Cache-Aware Roofline Model associe au modèle une méthodologie minutieuse de mesure de la bande passante et des performances. Celle-ci permet d'approcher les performances théoriques documentées par les constructeurs. Le CARM est également implémenté dans l'outil propriétaire Intel Advisor [60], et ne nécessite pas de mesure matérielle (mis à part le temps d'exécution), en dehors de l'instrumentation de l'application, car la quantité d'opérations arithmétiques et d'accès à la mémoire sont des propriétés de l'application uniquement et pas du matériel.

Le CARM se prête facilement à l'ajout de nouvelles bandes passantes telles que celles d'une mémoire sur la puce du processeur (comme la Multi Channel DRAM (MCDRAM) qui équipe les processeurs KNL ou les prochaines RAM non-volatiles). Cependant celui-ci ne tient pas compte d'autres problèmes de localité dus au placement explicite des fils d'exécution et des données tels que les accès distants, la contention, l'équilibrage de charge, l'utilisation simultanée de plusieurs mémoires, *etc.* Le 3DyRM [74] propose d'ajouter une troisième dimension de localité en représentant sur un troisième axe la latence des accès à la mémoire. Cependant il hérite de la perspective distordue du modèle original (*i.e.* mesure des transferts entre la mémoire et le cache). Le 3DyRM ne donne pas réellement d'information supplémentaire sur les goulets d'étranglement car il n'y a pas de valeur seuil ni de comportement attendu pour les latences observées. De plus les latences mesurées sont obtenues par échantillonage et ne sont pas nécessairement représentatives de tous les accès à la mémoire de l'application.

Ainsi, le Roofline Model et son extension (CARM) fournissent des informations claires et utiles vis-à-vis des performances de la machine, des performances des applications et des goulets d'étranglements (logiciels et matériels).

Cependant l'information fournie est limitée aux plates-formes dont les accès aux différentes mémoires sont uniformes entre les cœurs, sans tenir compte des problématiques des plates-formes multiprocesseurs.

4.2 Locality-Aware Roofline Model

Du point de vue des applications, la mémoire est abstraite par un espace d'adressage plat. Cependant, L'architecture des grands nœuds de calcul contemporains inclut des mémoires hétérogènes, locales ou distantes. Pour exploiter ces mémoires, les interfaces de programmation actuelles [80, 67, 13] requièrent un placement explicite des tâches et des données pour espérer obtenir de bonnes performances. En association avec les auteurs du CARM [57], nous avons développé le Locality-Aware Roofline Model (LARM), au dessus de ce dernier, pour prendre en compte les enjeux du placement des tâches et des données dans la performance des applications.

4.2.1 Goulets d'étranglements dans les plates-formes NUMA

En section 2.2 sont présentés différents goulets d'étranglement caractéristiques d'une plate-forme NUMA représentative des nœuds de CHP. En particulier, sont caractérisés les problèmes de l'allocation des données et des tâches par rapport aux accès distants (sous-section 2.2.5), et de la contention (sous-section 2.2.4), qui sont problématiques sur ces plates-formes et qui mériteraient d'être pris en compte dans le CARM.

En Figure 4.2 sont présentés les différents schémas d'accès aux données proposés dans notre extension LARM du CARM et caractéristiques d'une machine hypothétique composée de deux domaines NUMA. Les Figures 4.2a,4.2b illustrent la différence entre un schéma d'accès local ou distant. Conformément aux expériences menées en sous-section 2.2.5, selon que les cœurs d'un domaine NUMA accèdent à une mémoire locale ou distante, la bande passante n'est pas la même. Donc, la performance des applications est impactée par la localité des données. La représentation des bandes passantes locales et distantes que nous ajoutons au CARM permet de donner une notion de localité dans la performance du matériel et des applications.

La contention correspond à une surabondance de requêtes vers ou depuis un composant de la plate-forme, qui réduit la vitesse de traitement de chaque requête individuellement, voir collectivement. Par exemple, la contention de la mémoire par la lecture simultanée par tous les coeurs de celle-ci réduit à la fois pour chaque cœur et globalement la bande passante (*cf.* sous-section 2.2.4). Respectivement, en sous-section 3.3.1 et sous-section 3.3.2 sont cités des méthodes de placement de données et de fils d'exécution centrés sur l'optimisation

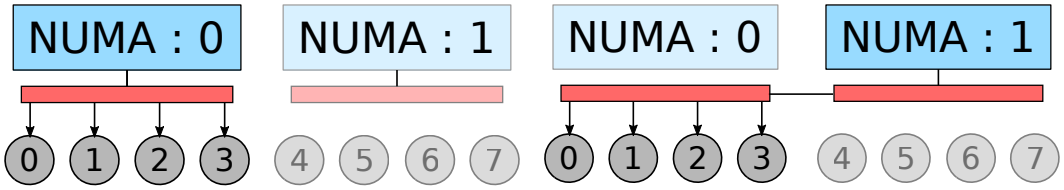
de la distance (thread, donnée) et de la contention dans la machine. Donc la caractérisation de la contention que nous incluons dans le CARM est pertinente. La Figure 4.2c exhibe un schéma d'accès maximisant la contention sur une mémoire, *i.e.* lorsque toutes les unités de calcul de la machine accèdent simultanément à une même mémoire. La bande passante mesurée par un des domaines NUMA, lorsque la mémoire est accédée selon ce schéma, est considérée comme la bande passante de contention et est représentée dans le modèle, à l'instar des bandes passantes des caches, comme une droite supplémentaire.

Nous ajoutons à ces goulets d'étranglement un cas moins étudié mais existant en pratique, celui de la congestion. La congestion dans un réseau correspond à la diminution du trafic global à cause de la contention sur un ou plusieurs liens du réseau. Lorsqu'une route d'acheminement des données est saturée, une autre route peu éventuellement être empruntée, mais si toutes les routes sont surchargées, alors il n'y a pas d'alternative à la perte de temps. De plus, sur les machines NUMA, le routage est souvent statique, c'est-à-dire que si un lien est saturée, on ne peut pas emprunter une autre route, donc la congestion est d'autant plus fréquente. En Figure A.2, nous représentons un tel cas de contention sur un processeur KNL (*cf.* annexe A.1) entre les unités de calcul 31 et 18, dont le routage fait nécessairement transiter les données par des liens saturés, alors que le routage entre les unités de calcul 31 et 27 ne présente pas ce problème.

Dans une situation de contention mémoire, une solution efficace pour améliorer la performance est de répartir les données sur les autres nœuds. La politique de placement de données *interleave* (présentée en sous-section 3.1.2) permet d'obtenir ce résultat. Cependant, un tel placement peut mettre en relief un autre goulet d'étranglement qui est la performance du réseau d'interconnexion entre les cœurs/processeurs en présence de congestion. Par conséquent, nous ajoutons un quatrième schéma d'accès mémoire (en Figure 4.2d) au CARM, *i.e.* où tous les cœurs accèdent à des données privées entrelacées sur les mémoires pour caractériser la congestion éventuelle de la plate-forme.

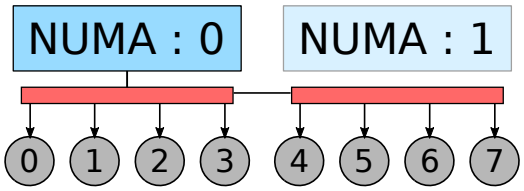
Enfin, nous ajoutons au modèle une notion de perspective, *i.e.* nous énumérons dans le modèle les domaines NUMA où sont caractérisées les bandes passantes. D'une part, cette représentation découpée par domaine est nécessaire pour caractériser les bandes passantes locales et distantes, qui peuvent varier, car toutes les plates-formes NUMA ne sont pas symétriques [24, 71]. D'autre part les applications peuvent aussi être asymétriques et comporter des problèmes d'équilibrage de charge [62] et que l'on peut exhiber avec une représentation par domaine NUMA, mais pas avec une représentation globale de la machine.

La représentation d'un tel modèle est donnée en Figure 4.3 pour un système hypothétique composé de deux domaines NUMA et d'une mémoire par domaine, tel que représenté en Figure 4.2. Sur le graphique représentatif de la plate-forme (Figure 4.3) on représente les bandes passantes du premier niveau

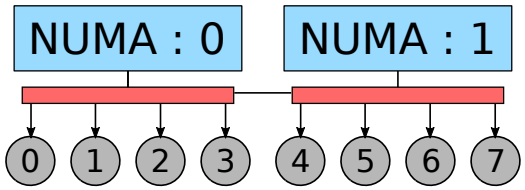


(a) Accès mémoire locaux au premier domaine NUMA.

(b) Accès mémoire distants au premier domaine NUMA.



(c) Contention : accès mémoire simultané de tous les cœurs au premier domaine NUMA.



(d) Congestion : accès mémoire simultané de tous les cœurs à tous les domaines NUMA.

FIGURE 4.2 – Schéma des accès à la mémoire modélisés dans le LARM pour une plate-forme composé de deux mémoires en bleu : NUMA:0 et NUMA:1, interconnectés par leur contrôleur mémoire en rouge.

de cache et de la mémoire du premier domaine, avec ou sans contention. On y représente également une application, telle que modélisée dans chacun des domaines. Cette application présente la particularité d'être limitée par la bande passante et d'effectuer ses requêtes uniquement vers la première mémoire. Pour un tel cas de figure, le modèle permet de détecter l'asymétrie dans l'utilisation des mémoires car la performance de l'application varie selon le domaine NUMA, et le goulet d'étranglement en cause, *i.e.* il y a de la contention sur la première mémoire, car la performance de l'application est bornée par la bande passante de contention.

En conclusion, le LARM est une évolution du Cache-Aware Roofline Model,

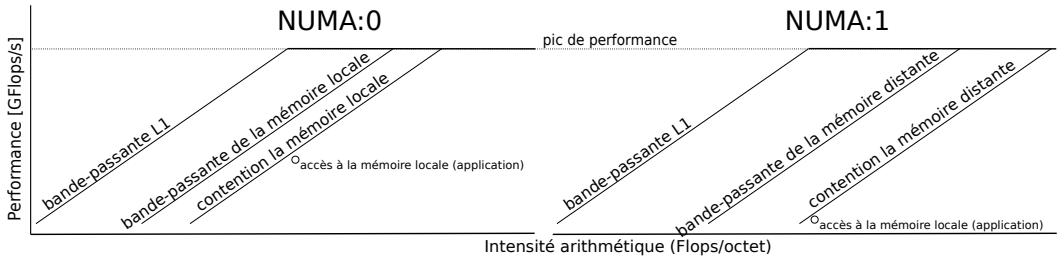


FIGURE 4.3 – Représentation graphique du LARM pour une plate-forme NUMA hypothétique composée de 2 domaines NUMA, et pour une application limitée par la bande passante mémoire effectuant la totalité de ses accès sur la mémoire du premier domaine.

Débit d'instruction	Load	Store	ADD	MUL	FMA
Théorique	2	1	1	2	2
Expérimental	1.99	0.99	0.99	1.99	1.99

TABLE 4.1 – Débit d'instructions (instructions/cycle) par cœur, d'un processeur Xeon E5-2650L v4.

tel que si il est instancié sur une machine NUMA, il possède une représentation par domaine NUMA, et des bandes passantes additionnelles pour les accès aux mémoires distantes, pour la contention des mémoires, et pour la congestion du réseau d'interconnexion.

4.2.2 Instanciation et validation du modèle sur machine NUMA

Dans un premier temps, nous détaillons le modèle sur une plate-forme NUMA bi-*socket* présentée en annexe A.3.

$$\underbrace{Fpeak}_{GFlop/s} = \underbrace{Throughput}_{Instructions/Cycle} \times \frac{Flops}{Instruction} \times N \times \underbrace{Frquence}_{GHz} \tag{4.1}$$

En s'appuyant sur la méthodologie du Cache-Aware Roofline Model [57], nous sommes capable d'approcher le débit théorique maximum (Table 4.1) des cœurs de la machine pour les instructions d'accès mémoire (*Load*, *Store*) et de calcul (*Add*, *Mul*, *Fused Multiply Add (FMA)*). Le débit de calcul de chaque cœur est dérivé (équation 4.1) de la mesure du débit d'instructions des cœurs (*Throughput*), de la fréquence des cœurs et de la quantité de calculs par instruction, multiplié par le nombre de cœurs N pour atteindre une performance de 190 GFLOPs/seconde (s) par domaine NUMA sur cette machine.

Comme décrit en sous-section 4.2.1, une évaluation complète de la plate-forme nécessite de mettre à l'épreuve la mémoire avec différents schémas d'accès aux données, *i.e.* locaux, distants, avec/sans congestion, avec/sans contention, pour chaque domaine NUMA de la plate-forme. Les bandes passantes mesurées sont présentées en Table 4.2 pour le premier domaine NUMA de la plate-forme. Sauf si spécifié autrement, les modèles présentés ci-inclus, sont restreints à un seul domaine NUMA à cause de la symétrie entre les domaines NUMA.

Les mesures de performance (Table 4.1) et de bandes passantes (Table 4.2) sont ensuite utilisées pour construire le modèle proposé et représenté en Figure 4.4 pour le premier domaine NUMA de la topologie. Le modèle est validé dans un premier temps avec des micro-*benchmarks* similaires à ceux utilisés pour la mesure des seuils de performance des cœurs et entrelaçants des instructions d'accès mémoire et de calcul sans dépendances. L'objectif de cette étape

Niveau de mémoire	Bande passante (Go/s)
L1	760.1
L2	309.2
L3	154.0
NUMANODE:0 (local)	36.1
NUMANODE:1 (distant)	17.5
NUMANODE:2 (distant)	15.0
NUMANODE:3 (distant)	14.3
NUMANODE:0 (contention locale)	16.7
NUMANODE:1 (contention distante)	8.3
NUMANODE:2 (contention distante)	6.8
NUMANODE:3 (contention distante)	6.2
NUMANODE:(0,1,2,3) (congestion)	18.1

TABLE 4.2 – Bandes passantes du premier domaine NUMA, *i.e.* NUMANODE:0, d'une plate-forme bi-*socket* Xeon E5-2650L v4. Les *benchmarks* sont exécutés sur les cœurs du premier domaine NUMA, tandis que les données sont allouées sur la mémoire du(des) nœud(s) correspondant(s) dans la colonne de gauche.

est de vérifier que la micro-architecture est effectivement capable de conserver son débit d'instructions en présence des deux types d'instructions et donc de se conformer au modèle. En Figure 4.4 on représente la performance atteinte pour une série de micro-*benchmarks* d'intensités arithmétiques variées. On constate que quasiment tous les points collent au modèle [1], donc la micro-architecture recouvre quasi-parfaitement les deux types d'instructions, excepté à l'intersection des limites de débit de calcul et de bande passante (lorsque le ratio d'instructions de calcul et d'instructions d'accès vaut 1 pour le L1) lorsque le débit d'instructions devient limitant.

L'instanciation et et la validation automatique du modèle utilisées ici sont implémentées pour les architectures x86-64, et pour les extensions du jeu d'instruction : SSE à AVX512 [129]. Le logiciel composé de $\simeq$ 3800 lignes de code, est hébergé à l'adresse : https://github.com/NicolasDenoyelle/Locality-Aware-Roofline-Model.

1. L'erreur globale visible sur la Figure 4.4 est calculée en légende comme $\frac{100}{n} \times \sqrt{\sum_{i=1..n} \left(\frac{y_i - \hat{y}_i}{\hat{y}_i}\right)^2}$ où y_i est la performance d'un micro-*benchmark* de validation et $\hat{y}_i$ le seuil de performance de l'intensité arithmétique correspondante.—

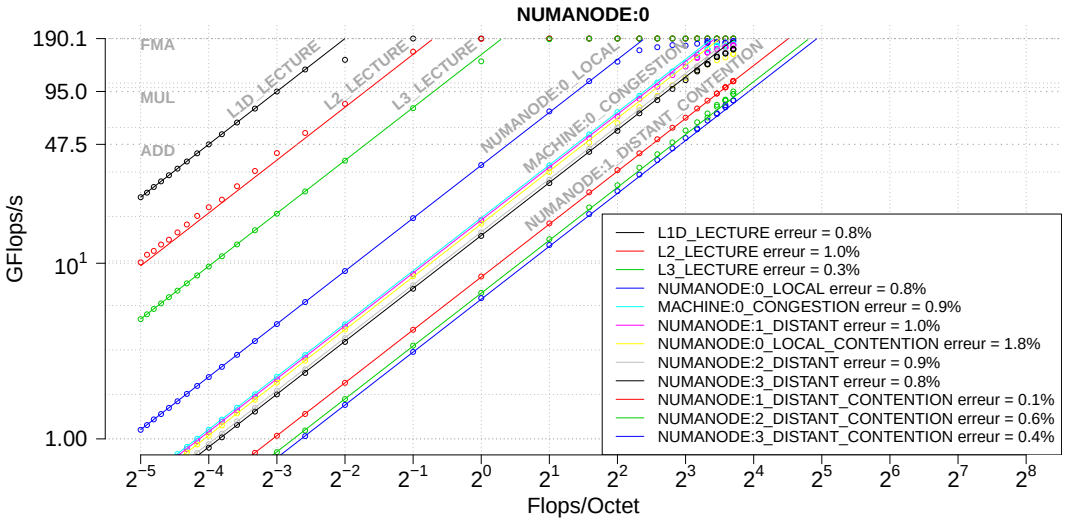


FIGURE 4.4 – Validation du modèle pour un domaine NUMA de plate-forme bi-*socket* Xeon E5-2650L v4. Les échantillons de validation sont visibles le long des seuils de performance. La légende donne le taux d'erreur du modèle par rapport aux micro-*benchmarks* de validation. Les seuils de performance en calcul flottant sont représentés en lignes pointillées horizontales pour les instructions FMA, MUL et ADD, dont les valeurs sont notées sur l'axe des ordonnées.

4.2.3 Cas d'utilisation du modèle sur machine NUMA

Avec des *benchmarks* synthétiques

Afin de démontrer l'intérêt du modèle nous exécutons des noyaux d'algèbre linéaire communément utilisés dans certaines applications, avec des schémas d'accès mémoire pathologiques correspondants aux cas décrits par le modèle. `ddot` (Figure 4.5a) est un produit scalaire. À chaque itération du calcul, `ddot` charge 2 nouveaux éléments depuis la mémoire et accumule leur produit dans un registre contenant le résultat intermédiaire. Son schéma d'accès lui confère la propriété d'utiliser toute la bande passante mémoire disponible. `dgemm` (Figure 4.5b) est un produit de matrice. À l'instar de `ddot` chaque itération du calcul, `dgemm` charge 2 nouveaux éléments depuis la mémoire et accumule leur produit dans un registre contenant le résultat intermédiaire. À la différence de `ddot`, le schéma d'accès aux données de `dgemm` peut-être optimisé de manière à exploiter les caches et à cacher les temps d'accès à la mémoire globale.

Sur le premier processeur de notre machine bi-*socket* Xeon E5-2650L v4 nous exécutons en concurrence sur chaque cœur dans un premier temps, un noyau `ddot` par cœur, puis dans un second temps, un noyau `dgemm` par cœur pour différentes politiques d'allocation des données. Sur la Figure 4.6, chacun des noyaux est représenté sur chaque domaine NUMA du processeur et

```
for(i = 0; i < n; i++){
  c = c + a[i] * b[i];
}
```

(a) ddot : produit scalaire de deux vecteurs a et b.

```
for(i = 0; i < n; i++){
  for(j = 0; j < n; j++){
    for(k = 0; k < k; j++){
      c[i,j] = c[i,j] + a[i,k] * b[k,j]
    }
  }
}
```

(b) dgemm produit de matrices a et b dont le résultat est stocké dans c.

FIGURE 4.5 – Pseudo-code de deux noyaux majeurs d'algèbre linéaire dense.

groupé par intensité arithmétique. Les points *firsttouch* représentent les exécution utilisant la politique d'allocation *firsttouch* (*cf.* sous-section 3.1.2), les points *interleave* représentent les exécution utilisant la politique d'allocation *interleave*. Enfin les points `NUMA:0` représentent les points dont les données sont exclusivement allouées sur la première des deux mémoires. On constate pour `dgemm` que la politique d'allocation mémoire n'a pas d'impact sur la performance du noyau. Celui-ci exploite suffisamment les caches pour recouvrir les temps d'accès à la mémoire. Cette observation est cohérente avec la position du point au dessus du seuil décrit par le dernier niveau de cache et suggérant de cibler un niveau d'optimisation plus près des cœurs que la politique d'allocation des données.

Au contraire la politique d'accès à la mémoire a une influence sur la performance du noyau `ddot`. La performance du noyau est bornée à celle de la mémoire lorsque les données sont locales aux cœurs (*firsttouch*). En revanche, celle-ci est limitée à la bande passante de congestion lorsque les données sont réparties (*interleave*) ou à la bande passante de contention (`NUMA:0`) lorsque tous les cœurs accèdent simultanément au même nœud NUMA. La limite de bande passante de l'application `ddot` est donc correctement caractérisée dans le LARM vis à vis de celle de la machine et permet d'identifier correctement les problème de localité suite à la politique d'allocation des données.

Enfin, on observe sur la Figure 4.6 le déséquilibre de performance qui se produit lorsque les données sont allouées de manière asymétriques (`NUMA:0`). Donc la représentation par domaine revêt aussi un intérêt par rapport aux

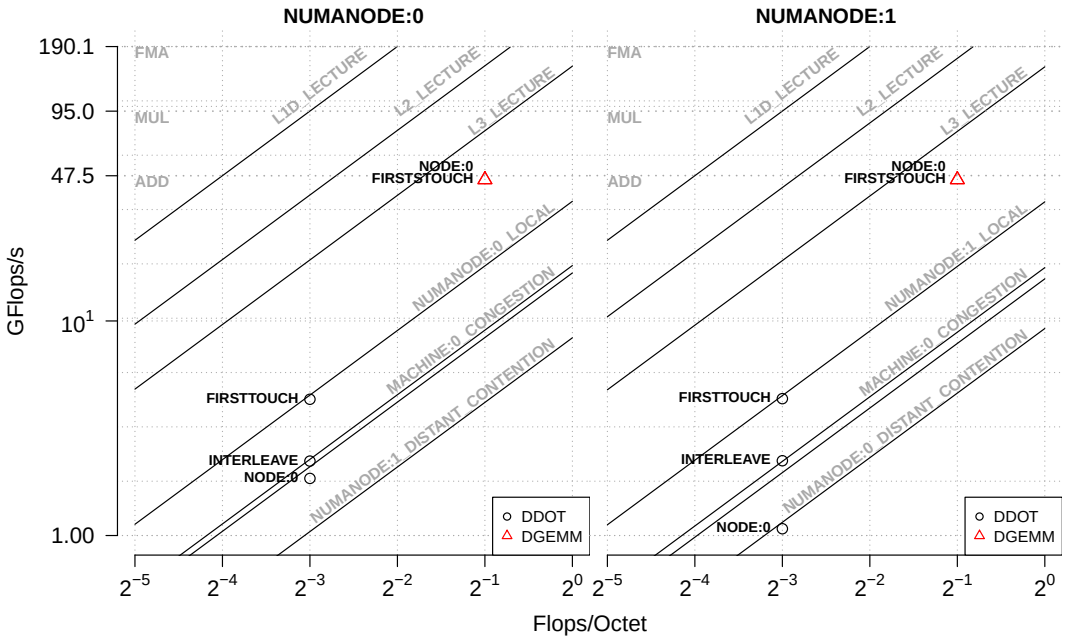


FIGURE 4.6 – Modélisation d'une implémentation de ddot et dgemm dans le LARM pour un processeur Xeon E5-2650L v4.

conséquences de la localité des données sur la performance des applications.

Avec une application

Nous exécutons l'application *MG* issue de la suite de mini-applications parallèles NAS [5]. *MG* est une application multi-grille sur une séquence de maillages qui induit des échanges de données sur de courtes et longues distances, et qui solicite la mémoire de manière intensive. Elle représente donc un cas d'utilisation potentiellement intéressant pour le modèle. Sur la machine complète utilisée dans ce chapitre, nous exécutons une implémentation en C [2] parallélisée avec OpenMP de l'application. Pour la première exécution, nous utilisons la politique de placement des données par défaut de Linux : *i.e. first-touch*. Les performances des trois fonctions les plus longues de l'application (obtenues avec un profilage préalable), *i.e.* `RESID`, `INTERP` et `RPRJ3`, sont représentées sur la Figure 4.7, avec respectivement un rond noir, un + vert, et un triangle rouge. Pour chaque fonction on représente trois points, avec le label *firsttouch* lorsque les donées sont allouées avec la politique correspondante, avec le label *interleave* lorsque les données sont allouées avec la politique éponyme, et enfin *enhanced firsttouch* lorsque les données dont allouées avec la politique firsttouch mais que la région d'initialisation des données est parallé-

2. https://github.com/benchmark-subsetting/NPB3.0-omp-C

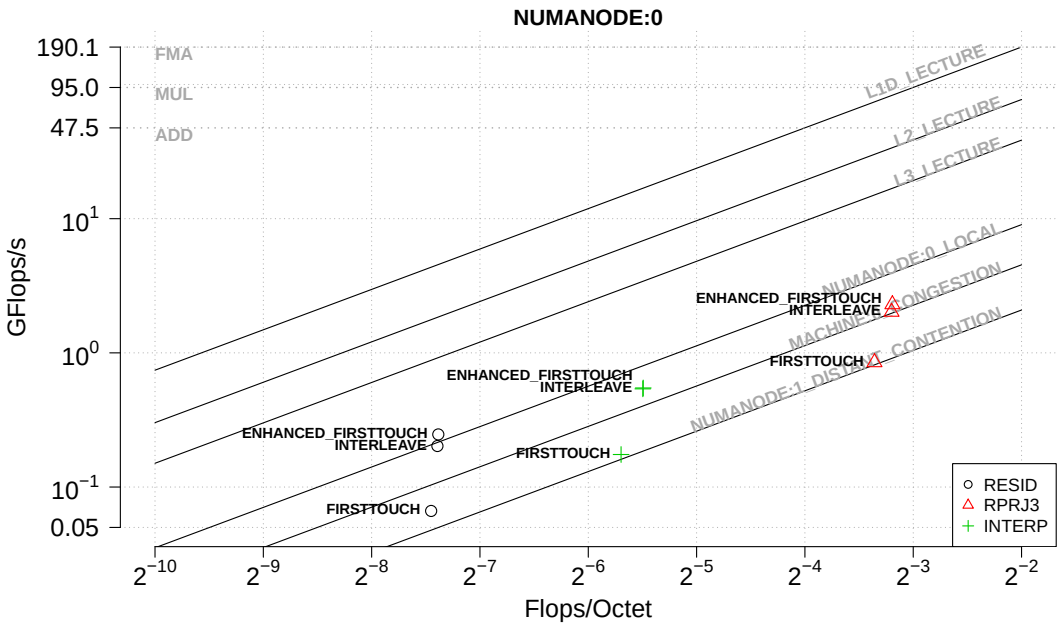


FIGURE 4.7 – Modélisation des trois fonctions principales de l'application *MG* dans le LARM pour une plate-forme bi-*socket* Xeon E5-2650L v4.

lisée. On constate dans un premier temps que les trois fonctions avec le label *firsstouch* sont caractérisées proches de la bande passante de contention.

En supposant que la contention est effectivement un goulet d'étranglement qui limite la performance de l'application, nous modifions le placement des données en appliquant la politique de placement *interleave* dans une deuxième exécution. L'augmentation significative de la performance qui déplace les points au delà de la bande passante de congestion confirme l'utilité de l'information fournie par le modèle. En observant le code source de l'application pour comprendre la cause de la contention, on peut s'apercevoir que l'initialisation des données est séquentielle et force les données à être allouées sur le seul nœud proche du thread principal qui initialise les données dans le cas de la politique *firsttouch*. Après parallélisation de l'initialisation des données, celles-ci sont mieux réparties sur la machine, et une dernière exécution *enhanced firsttouch* montre encore une légère amélioration des performances de l'application par rapport à la politique *interleave*.

En conclusion, le Locality-Aware Roofline Model est un modèle de machine et d'application construit au dessus du CARM qui fourni les premières intuitions pour optimiser des applications sur les machines NUMA. Il ajoute au modèle antérieur une représentation des goulets d'étranglement caractéristiques de tels plates-formes et permet de détecter avec succès si la performance d'une application est limitée par ces derniers. Ce modèle construit en collabo-

ration avec les auteurs du CARM fonctionne aussi sur le processeur KNL (*cf.* Annexe A.1) et permet de caractériser les seuils de performance de la machine et de ses différentes mémoires, selon sa configuration, tel que décrit dans une de nos contributions [126].

4.3 Modèle de bande passante hybride pour mémoire hétérogène.

Certaines applications contraintes dans leur schéma d'accès à la mémoire mais optimisées pour exploiter le potentiel maximum de la micro-architecture peuvent ne pas être caractérisées comme étant proche d'une des limites proposées dans le modèle. Pourtant, dans une telle situation, les utilisateurs du modèle s'attendent à obtenir l'information qu'une limite de performance de la plate-forme a été atteinte. Dans cette section nous proposons un modèle de performance [128] de la mémoire globale sous contrainte d'un schéma d'accès réaliste à celle-ci qui permet d'obtenir une limite plus fine de la performance du système de calcul en accord avec certaines contraintes des applications.

4.3.1 Motivations

Les applications issues de la suite de *benchmarks* STREAM [79] sont construites pour caractériser la performance de la mémoire des systèmes CHP. Cependant, si on les exécute sur un domaine NUMA d'un processeur KNL, en mode *flat* (avec la MCDRAM explicitement adressable) (*cf.* Annexe A.1), en l'absence de contention ou de congestion, avec des accès locaux, et qu'on les représente dans le LARM en Figure 4.8, on constate d'abord que la plupart des points ne coïncident pas avec les bandes passantes de la plate-forme représentées. Dans le cas de `ddot`, même sans congestion (un seul domaine NUMA utilisé), la politique *interleave* ne permet d'atteindre ni la bande passante de la RAM ni celle de la MCDRAM contrairement au schéma d'allocation en RAM (*firsttouch*) ou en MCDRAM. Donc un schéma hybride d'allocation des données limite la performance des accès à la mémoire. Dans le cas de `scale`[3] et `triad`[4], l'allocation en RAM (*firsttouch*) n'atteint pas la bande passante de la mémoire cible en lecture et excède celle en écriture. Donc le schéma de lecture et d'écriture des applications impacte aussi la performance des accès à la mémoire. En conclusion, pour une intensité arithmétique équivalente, mais des schémas d'accès aux mémoires différents, les applications limitées par la bande passante de la plate-forme ont des performances différentes qui ne sont pas représentées dans le LARM.

3. `scale` met à l'échelle les éléments d'un vecteur
4. `triad` calcul la somme élément par élément de deux vecteurs dont un est mis à l'échelle

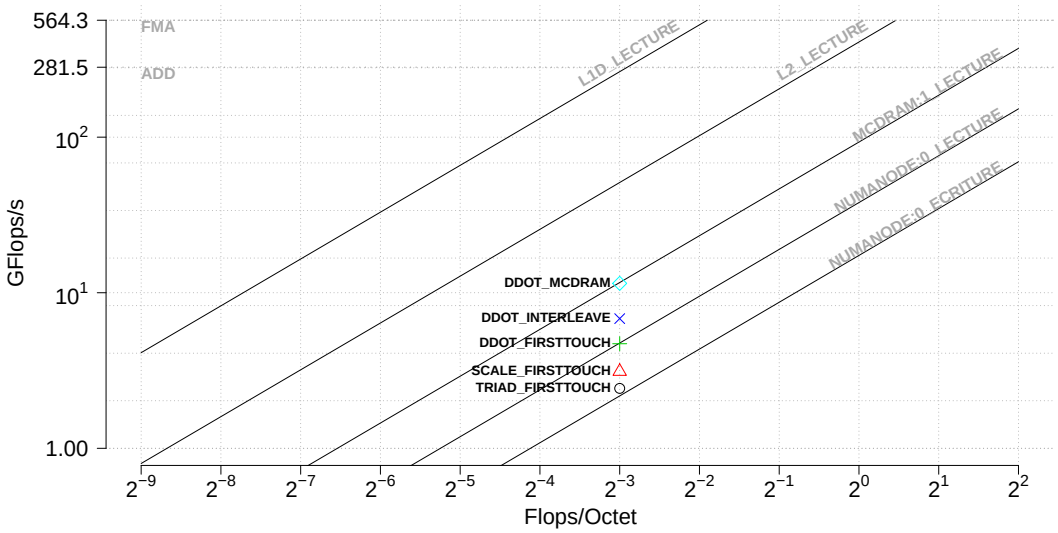


FIGURE 4.8 – Modélisation de certains bancs de test de la suite STREAM, dans le LARM, pour le premier domaine NUMA d'un processeur KNL configuré en mode SNC-4 *flat*.

4.3.2 Description du modèle

Le CARM décrit un système de files d'attente dont le débit global est limité soit par le débit des accès à la mémoire soit par le débit de calcul des cœurs, selon la composition en instructions d'accès mémoire ou de calculs de celle-ci. L'unité de traitement de la mémoire y est décrite comme une suite de sous-mémoires de débit décroissant (des cœurs vers les caches puis vers la mémoire globale). Nous représentons cette unité en Figure 4.9 par un entonnoir sur la partie gauche ou des cercles concentriques sur la partie droite. Ces éléments symbolisent le fait que le débit maximum atteignable est celui du niveau de mémoire où se trouvent les données, *i.e.* l'aire du plus petit des cercles inclus dans les niveaux de mémoire à traverser. Dans le LARM et dans la partie gauche de la Figure 4.9, nous modélisons des sous-systèmes de mémoire additionnels, de débit moindre, représentants les goulets d'étranglement des plates-formes NUMA. Dans cette section nous proposons de remplacer les éléments au dessus du cache par un modèle de bande passante paramétré par certaines contraintes des applications. Dans ce modèle, les cercles dans la partie droite de la Figure 4.9, ne sont plus concentriques mais disjoints avec une intersection commune. Cette représentation indique que les différentes requêtes traversent des canaux différents, mais dont une partie est commune. Cet aspect commun limite la bande passante atteignable à une valeur inférieure à la somme des aires des canaux, lorsque plusieurs canaux sont exploités en même temps. Pour construire ce modèle nous considérons une machine composée de deux mémoires ayant des bandes passantes différentes, comme c'est par exemple le cas dans un processeur KNL équipé d'une mémoire rapide sur la puce du pro-

cesseur (MCDRAM) et d'une mémoire lente à l'extérieur de celle-ci. Ensuite, nous caractérisons le schéma d'accès à la mémoire des applications selon la proportion d'utilisation d'une des quatre bandes passantes : lecture en mémoire rapide, écriture en mémoire rapide, lecture en mémoire lente, et écriture en mémoire lente. Enfin, le modèle que nous décrivons, considère que l'usage des 4 types d'accès mémoire peut s'effectuer en partie de manière simultanée et propose de modéliser de manière linéaire la partie séquentielle des accès qui correspond à l'intersection des cercles à l'intérieur du L3 sur la Figure 4.9.

En pratique ce schéma d'accès se matérialise par la quantité de lectures, et d'écritures dans le micro-code [5] de l'application et par la politique de placement des pages dans les différentes mémoires. On peut mesurer pour une application la quantité de lectures et d'écritures ainsi que la localité des accès à la mémoire avec des compteurs matériels. Donc on peut effectivement quantifier l'usage en lecture ou en écriture de l'une ou de l'autre des mémoires. De plus, on peut écrire des bancs d'essai avec une quantité arbitraire de lectures et d'écritures et un usage arbitraire des deux mémoires (à la granularité d'une page). Donc on peut tester la réponse de la plate-forme à des caractéristiques de certaines applications, la modéliser et la comparer à celle des applications.

La mémoire globale de la machine est caractérisée par différentes bandes passantes selon la mémoire particulière que l'on considère et sa distance avec les cœurs qui font les requêtes d'accès. La combinaison de toutes ces bandes passantes est sujette à la superposition de différents mécanismes implémentés dans le matériel, et impliqués dans le service des requêtes d'accès aux données. Depuis le pipeline d'instructions, les différentes requêtes peuvent déclencher des préchargements de données, des messages de cohérence, à travers toute la hiérarchie de caches, le réseau d'interconnexion des cœurs dans le processeur, les contrôleurs mémoires, le réseau d'interconnexion entre les processeurs, *etc.* Chaque élément de cette liste (non-exhaustive) sur le chemin des données implique des politiques complexes, qui ne sont pas toujours divulguées. Il est donc impossible de se baser sur une connaissance exacte du matériel pour modéliser la performance des requêtes à la mémoire. Par conséquent, à la différence de la méthodologie du CARM, nous adoptons une approche systémique qui considère certains éléments matériels comme opaques. Le but étant de modéliser de manière simple une série de mécanismes extrêmement complexes.

4.3.3 Exploration de l'espace des bandes passantes

Avec une méthode similaire à celle employée pour valider le LARM avec différentes intensités arithmétiques, nous générons dynamiquement des micro-*benchmarks* avec des ratios d'écritures et de lectures arbitraires exécutés par tous les cœurs d'un seul domaine NUMA de la plate-forme. À l'aide de la bibliothèque hwloc, nous allouons individuellement chacune des pages des données

5. code traduit en instructions spécifiques du processeur.

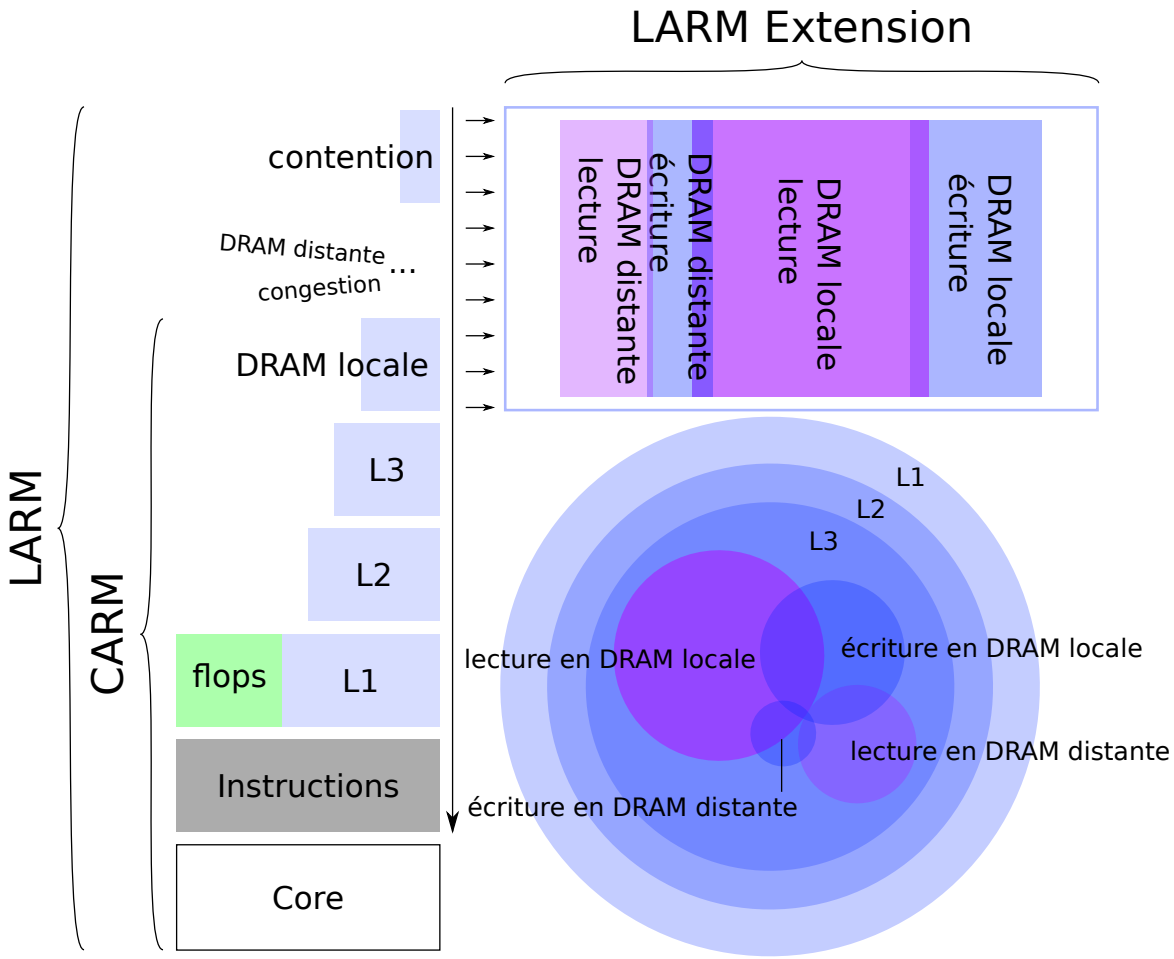


FIGURE 4.9 – Schéma d'inclusion d'un modèle de bande passante hybride dans le LARM.

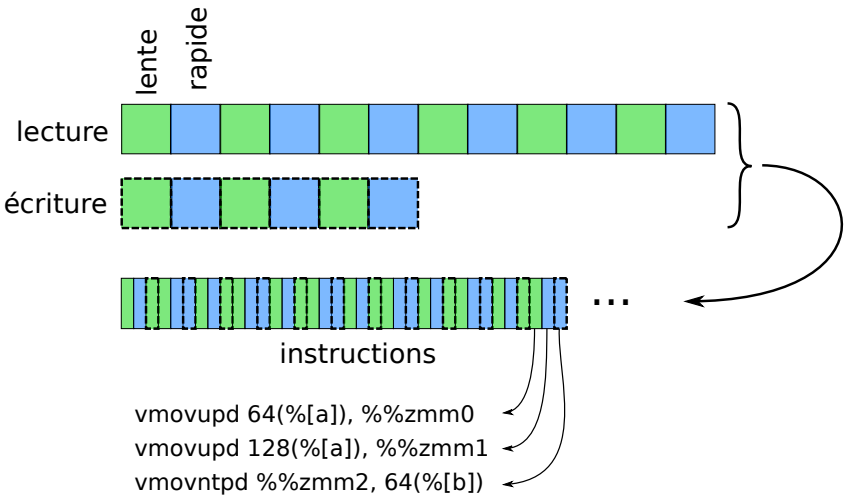


FIGURE 4.10 – Exemple de micro-*benchmark*, avec un schéma d'accès à la mémoire équilibrant les accès aux deux mémoires (bleue et verte), et avec deux fois plus de lectures (en trait plein) que d'écritures (en trait pointillé).

à lire et écrire dans la mémoire rapide ou la mémoire lente considérée selon la proportion choisie d'utilisation de l'une ou de l'autre. Les données allouées sont d'abord entrelacées sur chacune des mémoires de manière à paralléliser l'utilisation des contrôleurs mémoire, puis si les quantités d'utilisation des deux mémoires diffèrent, le reste des données est alloué sur la mémoire cible, comme décrit en Figure 4.10 [6].

Nous notons «ratio_rapide» la quantité $\frac{\text{rapide}}{\text{rapide+lent}}$, où «rapide» représente la quantité de données en mémoire rapide (en octets) et «lent» représente la quantité de données en mémoire lente. De même, nous notons «ratio_lecture» la quantité $\frac{\text{lecture}}{\text{lecture+écriture}}$, où «lecture» représente la quantité de données lues (en octets) et «écriture» représente la quantité de données écrites. Enfin nous parcourons les dimensions «ratio_rapide» et «ratio_lecture», tout en mesurant la bande passante fournie par deux plates-formes NUMA : Un processeur KNL (*cf.* Annexe A.1 et un processeur Skylake (*cf.* Annexe A.4), dont nous utilisons la mémoire locale au premier domaine NUMA comme mémoire rapide, et la première mémoire du second domaine NUMA comme mémoire lente.

Le résultat de cette mesure est représenté en trait plein rouge sur la Figure 4.11. Chacune des colonnes qui y est représentée caractérise une machine différente. Chacune des lignes caractérise une valeur différente de «ratio_lecture». Enfin pour chaque graphe, la bande passante mesurée est représentée en ordonnée en fonction de «ratio_rapide» en abscisse. On observe une continuité de la bande passante dans les deux dimensions «ratio_lecture»,

6. Ce schéma d'accès à la mémoire pour caractériser la performance de la mémoire est sujet à discussion. On peut par exemple imaginer une répartition régulière des données pour éviter un parcours à vitesse variable.

«ratio_rapide», avec un point anguleux lorsque l'usage des deux mémoires est parfaitement équilibré (autour de 50% des données en mémoire rapide). Donc un modèle linéaire de la bande passante observé doit être construit par morceaux pour être ajusté de part et d'autre des points anguleux.

La mémoire rapide du processeur KNL est gravée sur la puce du processeur, avec des accès aux contrôleurs mémoires disjoints selon que la mémoire rapide ou la mémoire lente sont utilisées. En contraste, la machine Skylake possède deux mémoires ayant les mêmes caractéristiques, mais dont les requêtes vers la mémoire lente traversent à la fois le contrôleur mémoire local et le contrôleur mémoire distant, ainsi qu'un réseau externe aux processeurs. Bien que ces plates-formes proposent des topologies fondamentalement différentes, la bande passante observée a la même allure. Par conséquent, on peut les modéliser avec le même type de modèle.

4.3.4 Formalisation et instanciation du modèle

Dans un premier temps, nous décrivons les différentes quantités nécessaires à l'expression formelle du modèle. Ensuite nous donnons une définition formelle d'un modèle monolithique simple exprimant les propriétés de parallélisme et de séquentialité des requêtes mémoires illustrés en Figure 4.9. Enfin, nous formalisons le modèle final, qui est un découpage par parties du modèle qui le précède. Une telle modélisation des performances de la mémoire d'une machine à mémoire hétérogène n'a à notre connaissance pas encore été proposée.

Notons :

- q_{ll}, la **q**uantité de données **l**ues depuis la mémoire **l**ente,
- q_{lr}, la **q**uantité de données **l**ues depuis la mémoire **r**apide,
- q_{el}, la **q**uantité de données **é**crites vers la mémoire **l**ente,
- q_{er}, la **q**uantité de données **é**crites vers la mémoire **r**apide,

tels que la quantité totale de données : $q_{ll} + q_{lr} + q_{el} + q_{er}$ soit constante entre les échantillons mesurés.

Notons :

- b_{ll} la **b**ande-passante en **l**ecture de la mémoire **l**ente,
- b_{lr} la **b**ande-passante en **l**ecture de la mémoire **r**apide,
- b_{el} la **b**ande-passante en **é**criture de la mémoire **l**ente,
- b_{er} la **b**ande-passante en **é**criture de la mémoire **r**apide,

mesurées avec la méthodologie du LARM.

Les temps minimums pour satisfaire les requêtes depuis/vers les différentes mémoires est alors :

- $t_{ll} = q_{ll}/b_{ll}$ secondes pour lire l'ensemble des données depuis la mémoire lente.

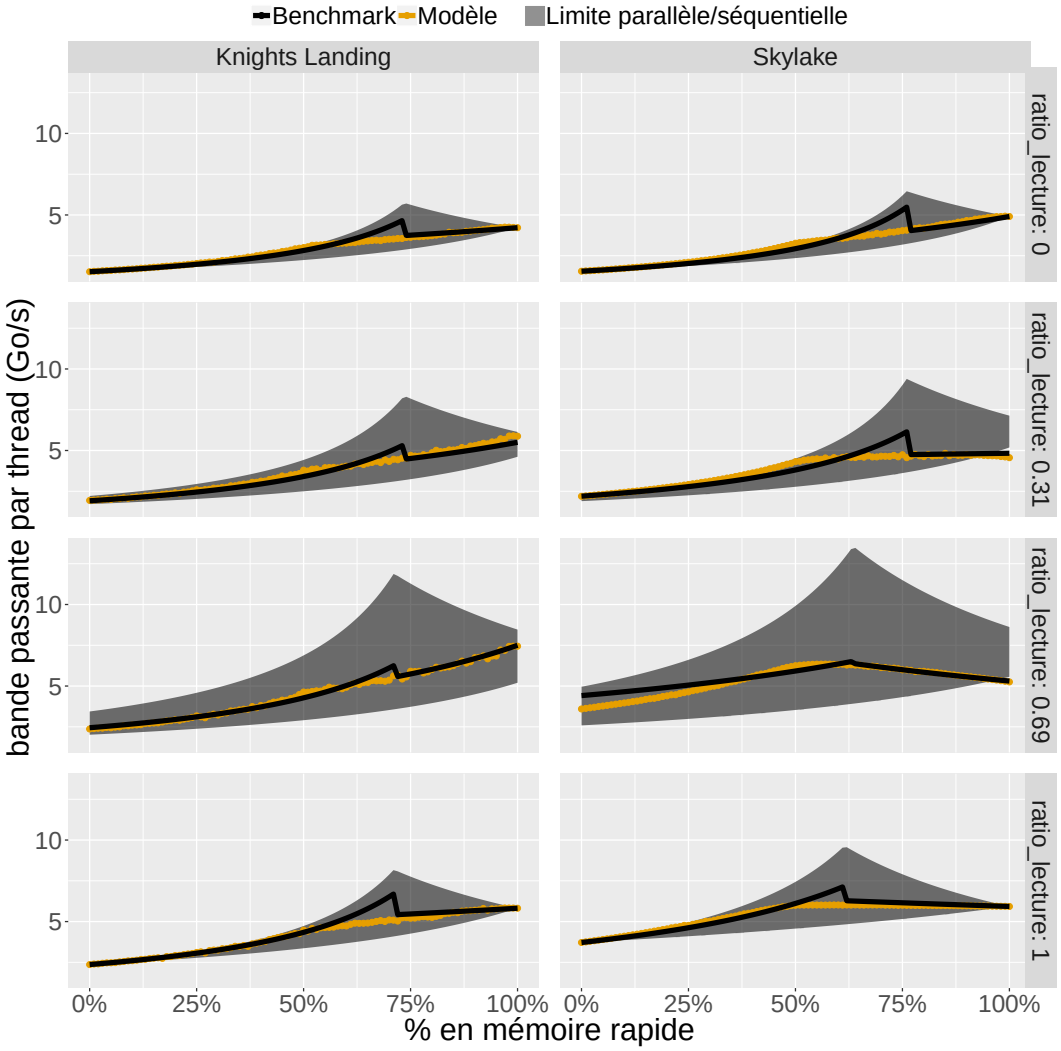


FIGURE 4.11 – Représentation graphique de la réponse en bande passante et de leur modèle pour deux plates-formes NUMA : KNL et Xeon(R) Gold 6140 bi-*socket*.

- $t_{lr} = q_{lr}/b_{lr}$ secondes pour lire l'ensemble des données depuis la mémoire rapide.
- $t_{el} = q_{el}/b_{el}$ secondes pour écrire l'ensemble des données vers la mémoire lente.
- $t_{er} = q_{er}/b_{er}$ secondes pour écrire l'ensemble des données vers la mémoire rapide.

$$t_{min} = max(t_{ll}, t_{lr}, t_{el}, t_{er}) \tag{4.2}$$

$$t_{max} = t_{ll} + t_{lr} + t_{el} + t_{er} \tag{4.3}$$

t_{min} (équation 4.2) est la durée minimum pour servir les requêtes mémoire si chacune d'entre elles peuvent être exécutées simultanément avec un recouvrement parfait. Avec ces notations, la courbe notée *max théorique* en Figure 4.11 représente la bande passante maximum atteignable : $(q_{ll} + q_{lr} + q_{el} + q_{er})/t_{min}$. Elle est déduite des bandes passantes de la machine et des quantités de données correspondantes aux requêtes pour chaque échantillon qui sont connues.

t_{max} (équation 4.3) est la durée maximum pour servir les requêtes mémoire si aucune d'entre elles ne peuvent être exécutées simultanément, *i.e.* si elles sont exécutées séquentiellement. Avec ces notations, la courbe notée *min théorique* en Figure 4.11 représente la bande passante minimum $(q_{ll} + q_{lr} + q_{el} + q_{er})/t_{max}$. De même, elle est déduite des bandes passantes de la machine et des quantités de données correspondantes aux requêtes pour chaque échantillon qui sont connues.

Lorsque seule une mémoire est utilisée, *e.g.* $q_{ll} + q_{lr} + q_{el} + q_{er} = q_{er} \implies ratio\_lecture = 0\ ET\ ratio\_rapide = 1$, la bande passante de l'échantillon est égale à celle de la mémoire aux bornes théoriques. Donc, comme le modèle le prévoit et l'expérience semble le confirmer, on peut prendre t_{min} comme ordonnée à l'origine du modèle linéaire du temps passé en transferts mémoire, et modéliser le temps supplémentaire séquentialisé des transferts.

Modèle monolithique

Dans un premier temps, en supposant l'absence du point anguleux de l'expérience, on note :

- θ_{ll} la part séquentielle des accès en **lecture** à la mémoire **lente**,
- θ_{lr} la part séquentielle des accès en **lecture** à la mémoire **rapide**,
- θ_{el} la part séquentielle des accès en **écriture** à la mémoire **lente**,
- θ_{er} la part séquentielle des accès en **écriture** à la mémoire **rapide**.

$$t_{modèle} = t_{min} + \theta_{ll} \times t_{ll} + \theta_{lr} \times t_{lr} + \theta_{el} \times t_{el} + \theta_{er} \times t_{er} \tag{4.4}$$

Le temps total $t_{modèle}$ (4.4) passé à attendre la fin des requêtes peut alors s'écrire comme le temps pour satisfaire toutes les requêtes de manière simultanée plus une proportion séquentielle inconnue de chacune des requêtes, et calculée avec une régression linéaire à partir des échantillons.

Modèle par morceaux

Afin que notre modèle épouse au mieux le comportement de la machine, tout en conservant une signification crédible par rapport au fonctionnement réel de la machine, nous proposons de considérer que le matériel implémente une politique de service des données différente selon le type de requête dominant, *e.g.* si on passe plus de temps à charger des données depuis la mémoire rapide qu'à satisfaire les autres requêtes. De cette manière, nous créons des points anguleux comme observés expérimentalement. De plus on peut légitimimement penser que c'est le transfert le plus long qui dicte la performance globale. Cela se traduit par une multiplication par 4 de la complexité du système dont les paramètres sont recalculés pour chacun des 4 cas considérés. Ainsi nous notons les variables binaires :

- $\text{ismax}_{ll} = 1$ si $max(t_{ll}, t_{lr}, t_{el}, t_{er}) = t_{ll}$ sinon 0
- $\text{ismax}_{lr} = 1$ si $max(t_{ll}, t_{lr}, t_{el}, t_{er}) = t_{lr}$ sinon 0
- $\text{ismax}_{el} = 1$ si $max(t_{ll}, t_{lr}, t_{el}, t_{er}) = t_{el}$ sinon 0
- $\text{ismax}_{er} = 1$ si $max(t_{ll}, t_{lr}, t_{el}, t_{er}) = t_{er}$ sinon 0

telles que $\text{ismax}_{ll} + \text{ismax}_{lr} + \text{ismax}_{el} + \text{ismax}_{er} = 1$. De cette manière on peut découper l'ordonnée à l'origine : $t_{min} = t_{ll} \times \text{ismax}_{ll} + t_{lr} \times \text{ismax}_{lr} + t_{el} \times \text{ismax}_{el} + t_{er} \times \text{ismax}_{er}$.

Pour chacun des 4 morceaux du modèle, la pente est définie comme la somme pondérée (par la part séquentielle) des requêtes qui ne dominent pas la durée des transferts, et l'ordonnée à l'origine comme la durée du transfert dominant. Ainsi nous re-écrivons les paramètres θ du modèle avec en indice le transfert dominant et en exposant le transfert dont une part est séquentielle. Par exemple, θ_{ll}^{el} représente la part séquentialisée des écritures en mémoire lente lorsque la durée du transfert est dominée par les lectures en mémoire lente, *i.e.* $\text{ismax}_{ll} = 1$. Le modèle ainsi découpé et paramétré s'écrit :

$$
\begin{aligned}
t_{modèle} = & \qquad (4.5)\\
& \text{ismax}_{ll} \times (\theta_{ll}^{lr} \times t_{lr} + \theta_{ll}^{el} \times t_{el} + \theta_{ll}^{er} \times t_{er} + t_{ll}) +\\
& \text{ismax}_{lr} \times (\theta_{lr}^{ll} \times t_{ll} + \theta_{lr}^{el} \times t_{el} + \theta_{lr}^{er} \times t_{er} + t_{lr}) +\\
& \text{ismax}_{el} \times (\theta_{el}^{ll} \times t_{ll} + \theta_{el}^{lr} \times t_{lr} + \theta_{el}^{er} \times t_{er} + t_{el}) +\\
& \text{ismax}_{er} \times (\theta_{er}^{lm} \times t_{ll} + \theta_{er}^{lr} \times t_{lr} + \theta_{er}^{el} \times t_{el} + t_{er})
\end{aligned}
$$

et est représenté en Figure 4.11 par la courbe notée *modèle*, et dont les paramètres θ (Table 4.3) sont calculés une seule fois et sont constants entre

chaque facette de la Figure. L'erreur relative moyenne du modèle par rapport aux bancs de test sur chacune des machines est inférieur à 2%. Toutefois, on remarque visuellement sur la Figure 4.11 que la majeur partie de cette erreur se situe autour du point anguleux. Il nous est possible de la réduire en augmentant la complexité du modèle, *e.g.* en ajoutant des termes correspondants à la fonction observée visuellement. Cependant, ce que le modèle gagnerai en précision, serait perdu en compréhension.

	θ^{er}_{lr}	θ^{ll}_{lr}	θ^{el}_{lr}	θ^{er}_{ll}	θ^{lr}_{ll}	θ^{el}_{ll}
Skylake	0.966	0.600	-0.102	0.465	0.294	0.373
KNL	0.238	0.722	0.985	0.956	0.611	0.564
	θ^{lr}_{er}	θ^{ll}_{er}	θ^{el}_{er}	θ^{er}_{el}	θ^{ll}_{el}	θ^{lr}_{el}
Skylake	0.912	0.067	0.337	0.059	0.293	0.54
KNL	0.183	0.953	0.797	0.726	0.571	0.65

TABLE 4.3 – Paramètres du modèle (*cf.* équation 4.5), calculés pour les plates-formes Skylake et KNL, à partir des échantillons représentés en Figure 4.11.

4.3.5 Validation du modèle

Afin de valider le modèle, nous proposons d'exécuter de nouveau les *benchmarks* proposés en sous-section 4.3.1. La valeur de «ratio_lecture» de chacune de ces fonctions est fixe, et nous les exécutons pour plusieurs valeurs de «ratio_rapide». Enfin nous observons la bande passante mesurée par ces *benchmarks* (dont la caractéristique est d'être limités par la bande passante de la plate-forme) en comparaison de celle mesurée par nos micro-*benchmarks* et de celle calculée par notre modèle en Figure 4.12. Nous représentons également sur la Figure 4.12 les bandes passantes en lecture et en écriture de chacune des mémoires pour visualiser la différence entre la caractérisation proposée dans le LARM et la performance réelle de la plate-forme. On constate pour le KNL que la bande passante de la plate-forme mesurée et celle des applications sont similaires, et le modèle construit est également proche des valeurs de bande passante mesurées, à l'exception de la région autour de 50% des données en mémoire rapide, car par construction, le modèle n'est pas découpé autour de cette région et ne peut donc pas parfaitement coller à la réalité. En revanche, la bande passante observée par les applications sur la machine Skylake, est proche (`ddot`) ou inférieure à celle modélisée. Cela témoigne des différences entre la bande passante de la machine et celle des applications, sans toutefois que ces dernières soient suffisamment supérieure pour invalider la nouvelle borne supérieure de performance de la mémoire.

Une version figée du code de cette expérience est disponible sur la plate-forme de dépôt d'expériences Code Ocean. Les bancs d'essais

sont à l'adresse : https://codeocean.com/capsule/e69c6f61-527f-4cff-bb28-5ee9c461e8ca/code, et les données expérimentales ainsi que le script d'analyse des données et de calcul du modèle sont disponibles à l'adresse : https://codeocean.com/2018/06/18/results-analysis-colon-modeling-non-uniform-memory-access-on-large-compute-nodes-with-the-cache-aware-roofline-model/code, et représentent un total de $\simeq$ 2800 lignes de code.

Notre contribution au Cache-Aware Roofline Model permet de rendre le modèle utilisable pour des machines NUMA, et ainsi ouvrir de nouvelles possibilités pour l'analyse des performances des applications. De plus nous proposons un modèle de bande passante hybride qui permet de caractériser de manière précise des situations hybrides d'utilisation de la mémoire qui apparaissent fréquemment en pratique. Enfin, nos contributions au modèle sont scrupuleusement validées avec des mesures approfondies des capacités de la plate-forme, et une confrontation avec les performances d'applications réelles.

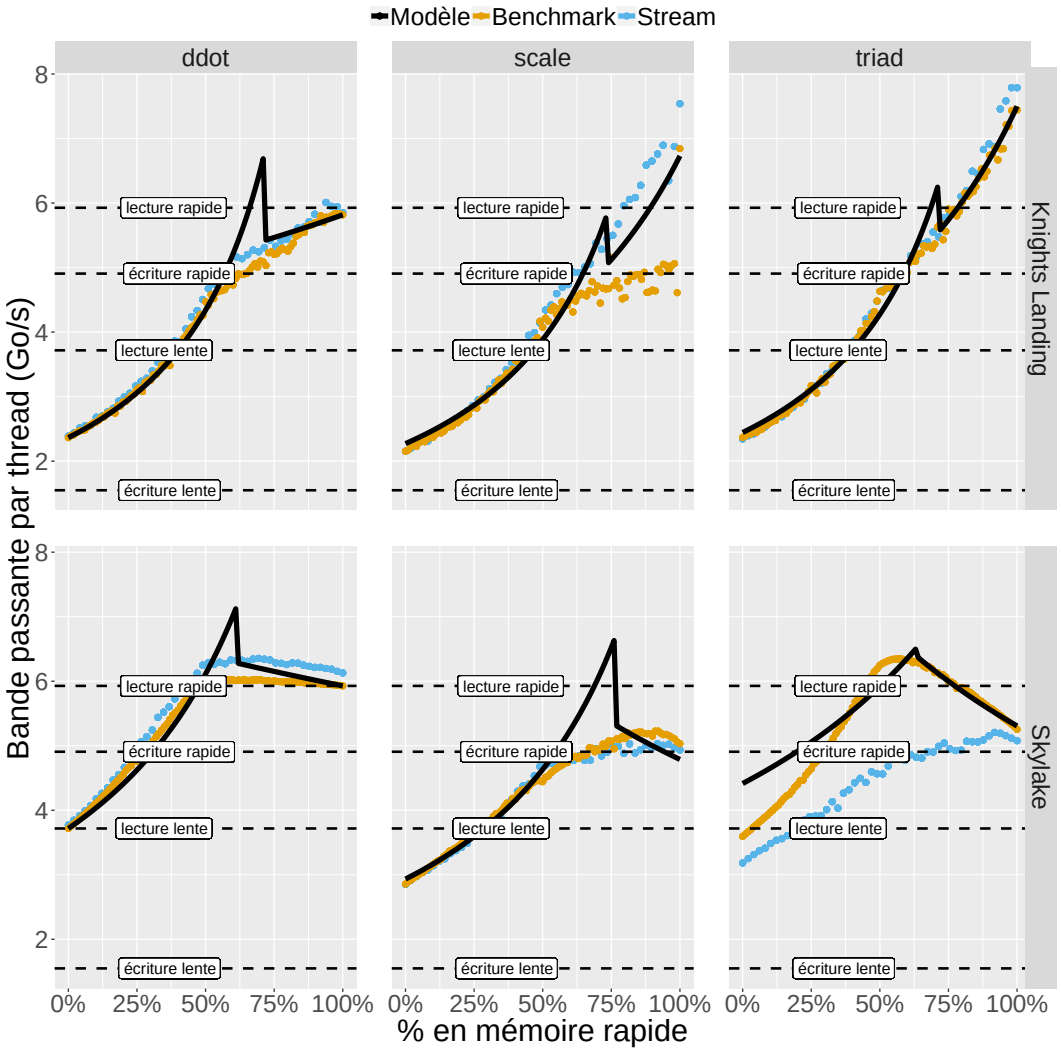


FIGURE 4.12 – Comparaison entre la bande passante par fil d'exécution des STREAM *benchmarks* et celle de la plate-forme pour des valeurs de «ratio_lecture» et «ratio_rapide» équivalentes. On représente également les bandes passantes (en lecture et en écriture) de la plate-forme comme point de comparaison si les applications avaient été caractérisées dans le LARM.

Chapitre 5

Moniteurs hiérarchiques d'application

La prévision de toutes les causes et les conséquences du placement des tâches et des données sur la localité dans la plate-forme n'est pas réalisable dans un temps raisonnable, à l'exécution. Néanmoins, la modélisation des machines et des applications contemporaines nécessite de prendre en compte la structure du système. Dans ce Chapitre nous proposons un outil et une méthodologie pour faciliter la mise en place des liens entre les évènements logiciels et les évènements matériels tout en prenant en compte la topologie du système de calcul, dans le but de facilement construire des métriques de localité.

5.1 La complexité du système (machine, application) nécessite la collections d'information localisée

La performance dépend de l'interaction du code et des différents composants du système de calcul. De nos jours, l'optimisation des performances nécessite de placer les tâches et les données en prenant en compte les besoins des applications parallèles relativement au matériel sur lequel elles sont exécutées. Les évènements qui se produisent dans le matériel et qui caractérisent la performance des applications trouvent leur source dans le code mais également dans le matériel dont les différents composants sont en interaction permanente. On distingue les évènements propres au code, et ceux propres à la machine relativement au code qu'elle exécute, ou encore ceux relatifs au système d'exploitation. Par exemple le nombre d'instructions exécutées et le nombre d'opérations arithmétiques, sont des caractéristiques de l'application indépendantes de la machine cible. En revanche le préchargement de données dans un niveau intermédiaire de mémoire ou l'exécution spéculative d'instructions, sont des conséquences de l'implémentation du matériel. La quantité de données préchargées dans un niveau de cache dépend de la fréquence d'utilisation des données déjà présentes, car il ne faut pas remplacer des données fréquemment utilisées par d'autres moins utilisées. Et la fréquence d'utilisation des données dans un cache dépend de la politique de réutilisation des données de l'application, qui selon la topologie des caches peut influencer la présence des données dans tel ou tel niveau de mémoire (*cf.* sous-section 2.1.2). On voit donc qu'il y a une interaction entre les différents niveaux de mémoire et l'application. Par conséquent, de la bonne synergie des composants de la machine dépendent la vitesse et la consommation énergétique des applications. Donc, pour comprendre comment optimiser une application relativement à une machine, il paraît nécessaire de modéliser l'interaction entre les composants en lien avec l'application qui y est exécutée.

5.1.1 Modéliser la localité nécessite d'abstraire les applications et la structure de la machine

On ne peut pas simuler entièrement la complexité des interactions au sein de la machine dans un temps raisonnable. On pourrait considérer une approche consistant à simuler logiciellement le matériel comme le proposent les simulateurs à la précision du cycle. Mais cette approche présente plusieurs inconvénients. D'abord il est nécessaire de dérouler la simulation à un surcoût exorbitant (la durée d'exécution est supérieure à 1000 fois la durée sur un processeur classique). Pour traiter de tels cas, on simule uniquement des portions de code extrêmement réduites comme le proposent les suites Parsec [8] (in-

put sim) ou NAS [2] (classe S), mais moins représentatives du comportement réel des applications. Rien que l'instrumentation logicielle de chaque instruction, accès mémoire *etc.* peut multiplier par plus de 100 le temps d'exécution sans même avoir à traduire toutes les conséquences matérielles de leur exécution. Ensuite, en tenant compte du coût humain de développement d'un processeur, il n'existe pas d'initiative ouverte pour reproduire avec exactitude l'entièreté d'un processeur contemporain. En 2017, Intel le principal fondeur de processeurs et un fournisseur émergents de mémoires pour le Calcul Haute Performance (CHP) comptait 102700 salariés, dont 87% à des postes techniques. 39% des revenus de la compagnie provenaient de la vente de puces (processeurs, matrice de portes programmables in situ (FPGA), 3%), *etc.*), de mémoires (6%) et de solutions pour les centres de calcul et d'hébergements de données (30%) [59]. En estimant une masse salariale en pourcentage équivalente aux revenus, cela représenterait de l'ordre du millier salariés rien que dans la conception et fabrication des processeurs, sans tenir compte de la spécificité des mémoires et de l'interconnection entre les lames et les racks de calcul, *etc.* Au delà de ce chiffrage grossier, il est certain que simuler au cycle près un processeur complexe, performant et crédible est hors de porté du monde académique.

Donc pour améliorer la performance par la localité, il faut simplifier le problème par une approche haut niveau. De la synergie entre le code et la machine dépend la performance de l'exécution. On ne peut pas la modéliser complètement. Néanmoins, il reste nécessaire d'optimiser la localité, et donc de modéliser le système (machine, code) par rapport aux compromis de la localité des données. On a donc besoin de modèles structurels de machine (*cf.* sous-section 2.1.5), de modèles d'application et d'un lien entre eux. En l'occurence, on décide de fixer le modèle de machine avec le modèle hiérarchique proposé par hwloc [13], et on fourni une abstraction pour construire un modèle d'application associé à la structure du système. La mise en relation d'évènements de la mémoire avec la topologie [46] de cette dernière permet avec une analyse *post-mortem* de déduire des problèmes de localité (*e.g.* contention sur certaines mémoires) et de corréler certains évènements.

Cependant le champ des problèmes de performance est plus large et comprend entre autres l'interaction entre les différents composants de la hiérarchie de caches qui n'est pas modélisée. La mise en relation de la topologie via l'interface hwloc avec des compteurs matériels [100] a été proposé simultanément à notre contribution. En particulier, il est argumenté que les causes possibles des problèmes de performances sont nombreuses et notamment liées à la performance séquentielle des applications. Ceci rejoint notre proposition d'abstraire la collecte des évènements pour inclure la diversité des causes de dégradation des performances. Cependant, la preuve de concept proposée inclut des évènements statiques et ne permet donc pas de s'adapter au contexte (*e.g.* matériel). De plus, l'analyse dynamique, voir *in-situ* des évènements n'y est pas proposée

pour privilégier une analyse directe, qui vise un panel d'optimisations moins large que celui proposé ici. En contraste, notre approche a pour objectif de modéliser le système (machine, application) en entier, dynamiquement, avec des liens structurels entre les acteurs. Tiptop [99] propose une analyse similaire à celle proposée par le même auteur [100] , qui associe des évènements précis à des objets de la topologie. Néanmoins, Tiptop se prive de l'association (évènement, matériel) au profit d'un affichage en coup d'œil à la manière de l'outil `top` mais avec des compteurs matériels.

5.1.2 Cas d'utilisation de l'information localisée

À la granularité d'un nœud de calcul, l'efficacité du placement des fils d'exécution dépend en partie de la pression exercée sur les caches [124]. En sous-section 3.3.2 nous décrivons comment la collocation de threads occupants un cache partagé (Figure 3.3) peut entraver leur exécution, et nous mettons en évidence cette propriété (Table 3.1) en observant la variation du temps d'exécution selon le placement des threads qui utilisent beaucoup le cache avec des threads qui ne l'utilisent pas.

Un indicateur de potentiel d'amélioration du placement des fils d'exécution relativement à la pression exercée sur les caches peut être la variance des accès cumulés à chaque cache. Si la variance est élevée, alors il y a un déséquilibre dans les accès aux caches partagés, si elle est faible alors les accès au cache sont équilibrés. Si la pression sur les caches est élevée et déséquilibré, alors on peut la réduire globalement en équilibrant les accès sur la machine (*cf.* Table 3.1). Au contraire si la pression est déjà équilibré, on ne peut pas les réduire sur un cache sans augmenter la pression sur un autre cache et pénaliser globalement l'ordonnancement.

Pour mesurer l'indicateur de pression sur les caches, il faut procéder en plusieurs étapes :

- Collecter sur les cœurs les compteurs d'accès au cache partagé ;
- Accumuler sur chaque cache la valeur de ces compteur en tenant compte de la topologie de la machine ;
- Calculer la variance des valeurs sur chaque cache.

Nous proposons un outil permettant d'abstraire ce processus pour modéliser cet indicateur, et plus généralement ceux qui tiennent compte de la structure de la machine.

5.2 Modèle hiérarchique de performance

Dans l'optique de résoudre le problème d'abstraction de la performance, relativement à l'interaction application/machine, nous proposons un outil de

mesure et d'agrégation d'évènements sur la topologie matérielle. Afin de mesurer des indicateurs de la performance d'une application en lien avec un objet particulier de la topologie (*e.g.* un cœur de calcul, un cache, une mémoire, *etc.*) de la machine, nous associons des moniteurs avec des objets de la topologie.

5.2.1 Modèle hiérarchique de moniteurs

Un moniteur est une entité logicielle capable de collecter des évènements à la demande et de les traiter pour fournir un ou plusieurs nouveaux évènements en sortie tel que décrit en Figure 5.4. Pour chaque moniteur, les évènements en entrée sont issus d'une seule source implémentant l'interface de mesure d'évènements. Cette interface est instanciée par objet de la topologie associé à un ou plusieurs moniteurs, *e.g.* un seul compteur d'instruction par cœur, un seul compteur de défauts de cache par cœur, *etc*, tels que présents physiquement dans la machine, et partagée entre les moniteurs associés à cet objet. Au sein d'un objet de la topologie matérielle, les moniteurs sont structurés hiérarchiquement, tout comme le sont les nœuds de la topologie eux-mêmes. Donc la structure globale des moniteurs et de leurs liens est également hiérarchique, conduisant éventuellement à agréger tous les évènements en une seule information synthétique à la racine de la topologie.

Cette opération de réduction hiérarchique est effectuée de manière similaire au calcul de la variance des défauts de cache mesurée par les compteur sur les cœurs, accumulés dans les caches puis agrégés à la racine. On peut par exemple représenter l'indicateur de variance sur les caches en 3 moniteurs (Figure 5.2). Dans le premier (Figure 5.2a) on collecte sur les *PU* (threads matériels) les compteurs d'accès au cache *PAPI_L3_TCA*, via l'interface PAPI [83]. Dans le second (Figure 5.2b) on accumule les compteurs de défauts de cache dans les caches *L3* correspondants, via un greffon *accumulate* de la bibliothèque dont le fonctionnement est illustré en Figure 5.1, et qui permet de sommer plusieurs vecteurs d'évènements de nœuds fils de la topologie dans un nœud parent. Dans le dernier (Figure 5.2c) on calcule la variance *hmonitor_evset_var* des moniteurs sur les caches, à la racine de la topologie (objet *Machine*), via un greffon de jointure *hierarchical*. Ce greffon permet de joindre les évènements des nœuds fils de la topologie dans un nœud parent, et est associé à une fonction de réduction qui permet de calculer la variance de ces évènements.

5.2.2 Des moniteurs à l'interface de la mesure et de l'agrégation d'évènements

Pour faire le lien entre les objets de la topologie matériel et des caractéristiques de performance, on a besoin de collecter des évènements et de les traduire dans une information synthétique. Donc, chaque moniteur a besoin

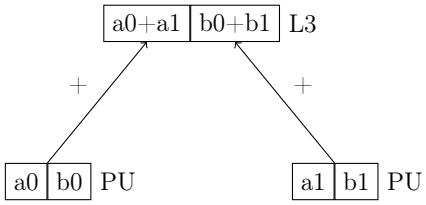


FIGURE 5.1 – Illustration du fonctionnement de l'accumulation d'évènements dans les niveaux supérieurs de la topologie grâce à la bibliothèque de moniteurs.

```
L3_ACCES_PU{
  OBJ:=PU;
  PERF_LIB:=papi;
  EVSET:=PAPI_L3_TCA;
}
```

(a) Collecte des accès au cache partagé

```
L3_ACCES_L3{
  OBJ:=L3;
  PERF_LIB:=accumulate;
  EVSET:=L3_ACCES_PU;
}
```

(b) Accumulation dans les caches

```
L3_var{
  OBJ:=Machine;
  PERF_LIB:=hierarchical;
  EVSET:=L3_ACCES_L3;
  REDUCTION:=1#hmonitor_evset_var;
  OUTPUT:=1;
}
```

(c) Calcul de l'indicateur

FIGURE 5.2 – Description des moniteurs pour le calcul de l'équilibre des accès au cache partagé.

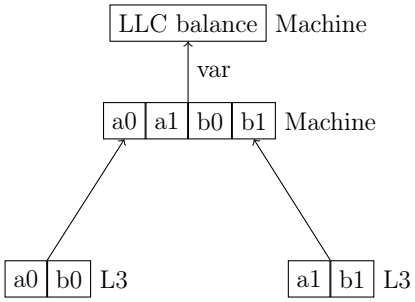


FIGURE 5.3 – Illustration du fonctionnement de la jointure d'évènements dans les niveaux supérieurs de la topologie grâce à la bibliothèque de moniteurs.

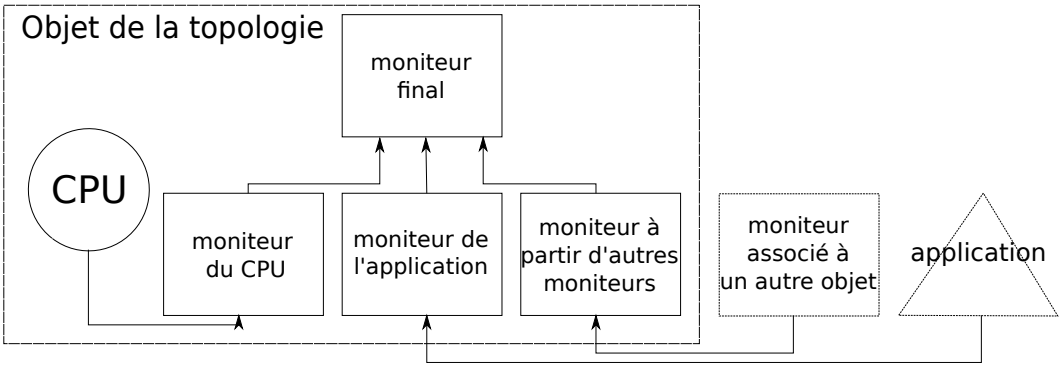


FIGURE 5.4 – Schéma structurel de la connection entre les moniteurs et de leur association avec un objet de la topologie.

d'une interface pour mesurer des évènements, et d'une interface pour les agréger comme décrit en Figure 5.5.

Collecte d'évènements L'abstraction de la collecte d'évènements pose plusieurs problèmes, car les évènements proviennent de sources différentes (*e.g.* compteurs matériels du processeur, du réseau, instrumentation du code, évènements du système d'exploitation, *etc.*) qui exposent des interfaces différentes, et sont collectés avec des méthodes variées : échantillonage, sondes, interruption par débordement, *etc.* Ce problème est largement traité par l'interface PAPI [83] qui fourni déjà une abstraction pour lire les compteurs de la machine, réaliser des opérations simples entre eux, faire le lien entre les évènements et le processus qui les génère, et une abstraction pour collecter d'autres types d'évènements, comme ceux issus du système d'exploitation ou du réseau d'interconnexion des processeurs d'une plate-forme. Cependant il existe d'autres couches d'abstraction qui réalisent en partie la même chose que l'interface de PAPI ou bien des choses différentes, qui ne sont pas encore intégrées dans PAPI parce qu'elles sont récentes, utilisent une approche différente ou bien collectent des évènements qui ne pourraient pas être collectés à travers la méthodologie de PAPI. Pour faire le lien entre les sources d'évènements, nous proposons une interface simple au dessus de PAPI et inspirée de cette dernière afin de connecter plusieurs sources d'évènements.

Elle est implémentée par défaut pour collecter des évènements matériels issus d'un objet de la topologie, par les abstractions de PAPI et maqao. Elle est également implémentée pour collecter les sorties d'autres moniteurs. On pourrait également l'implémenter pour enregistrer des évènements issus de l'application elle-même. Celle-ci propose de collecter les évènements comptés à la demande de la même manière que sont lus les registres où sont comptés les évènements matériels, ou bien de la même manière qu'on lit les évènements du système d'exploitation dans le système de fichiers `/proc`, tel que décrit en

Figure 5.7. La bibliothèque de moniteurs implémente également une couche au dessus permettant d'échantillonner la lecture des compteurs entre deux points de mesure dans le temps pour capturer les variations dans le comportement d'une application entre deux points de l'exécution.

Agrégation d'évènements Il y a un gradient de complexité dans l'analyse des évènements. Il est parfois possible de conclure directement de la valeur d'un évènement par rapport à un contexte, alors qu'il peut aussi être nécessaire d'effectuer une analyse complexe (*e.g.* de l'apprentissage statistique), coûteuse en calcul avec des outils d'analyse statistique tels que proposés par l'environnement R. Par exemple, lorsqu'on veut valider l'effet d'un changement, *e.g.* un placement des fils d'exécution qui modifie la localité des données, on observe directement le changement de la valeur d'un compteur lié à l'objectif, *i.e.* dans la hiérarchie mémoire. Lorsqu'on recherche un problème de performance, on commence déjà à observer des analyses plus complexes qui font écho à la complexité du système de calcul, *e.g.* la mise en relation du débit de donnée et de calcul du processeur avec les évènements de calcul flottant et de chargement/écriture de données générés par l'application dans le Roofline Model [121].

Nous envisageons d'autres applications que celle présenté plus haut. En particulier, la détection dynamique de changements de phase des applications ou encore la prévision (hiérarchique) en cascade de l'utilisation de certaines ressources est rendue possible par l'enregistrement de plusieurs échantillons d'évènements répartis au cours du temps dans les moniteurs. Les valeurs des échantillons enregistrés au cours du temps sont accessible par l'interface d'agrégation d'évènements. Celle-ci est appelée après chaque collecte, pour calculer les sorties des moniteurs. La bibliothèque de moniteurs hiérarchiques peut fonctionner à l'exécution. Donc on peut effectuer la collecte, l'analyse et l'optimisation d'une application dynamiquement. On peut facilement activer/désactiver l'utilisation des moniteurs, donc on peut aussi mesurer le surcoût du traitement des données par rapport au bénéfice des optimisations associées.

5.2.3 Architecture générale de l'outil

L'organisation schématique de la structure de l'outil est donnée en Figure 5.6 et fonctionne de la manière suivante : L'utilisateur donne en entrée de la bibliothèque un fichier de **description des moniteurs**. Il décrit ce qui est mesuré, et par quel moyen (**interface de collecte**). Cette interface est appelée à chaque mesure faite par les différents moniteurs associés à une implémentation (**PAPI**, **maqao**) de celle-ci. La bibliothèque s'occupe d'instancier des moniteurs logiciels sur les ressources de la machine grâce au modèle de représentation de la **topologie de la machine** fourni par **hwloc**. À chaque appel de l'**application** à la bibliothèque pour déclencher une mesure, les évènements sont collectés aux feuilles de l'arbre des moniteurs. Ils sont alors immédiate-

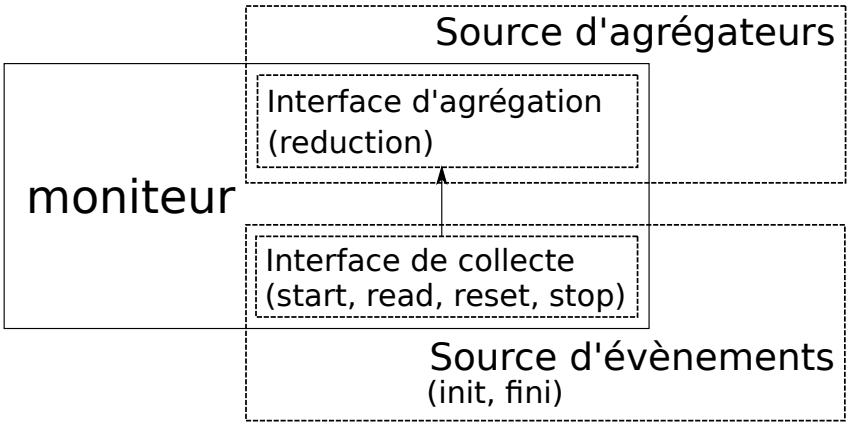


FIGURE 5.5 – Interaction entre la collecte et l'agrégation d'évènements dans un moniteur.

ment agrégés sur la topologie, par de simples opérations ou par des opérations plus complexes. Celles-ci sont implémentées selon l'**interface d'agrégation** sélectionnée dans la **description des moniteurs**, et présentée en Figure 5.2. Les résultats des agrégations sont optionnellement retranscrits dans une **trace d'évènements de sortie** ou bien dans une **sortie graphique dynamique** obtenue grâce à la **bibliothèque d'affichage** de **hwloc**.

5.2.4 Fonctionnalités et utilisation de l'outil

L'outil de modélisation peut être utilisé directement à l'intérieur de l'application pour une mesure localisée par le biais d'une bibliothèque. En Figure 5.7 est représenté un exemple basique d'utilisation. La bibliothèque nécessite d'être initialisée avec la topologie qui est associée aux évènements. Si celle-ci n'est pas fournie alors celle de la machine qui exécute l'application est utilisée. Il est ensuite nécessaire d'importer la description des moniteurs via la fonction *hmon_import_hmonitors*, pour les instancier. Après avoir démarré la collecte d'évènements, on peut lire leur valeur lors d'un appel à la fonction *hmon_update*. Celle-ci réveille les threads de la bibliothèque qui appellent depuis chaque feuille de la topologie les compteurs locaux puis calculent en parallèle les sorties du modèles suivant un arbre de réduction décrit par les moniteurs, avant de se rendormir. La bibliothèque permet d'échantillonner les mesures afin de détecter des variations de comportement via des interruptions périodiques qui réveillent les threads de mesure. Dans l'exemple en Figure 5.7 l'échantillonnage est encadré par les appels *hmon_sampling_start* et *hmon_sampling_stop*. Entre ces deux appels nous créons un processus dont la durée de vie est d'une seconde nous permettant de mesurer l'activité de système de manière échantillonnée sans endormir le processus de mesure, afin de démontrer la fonctionnalité d'échantillonage.

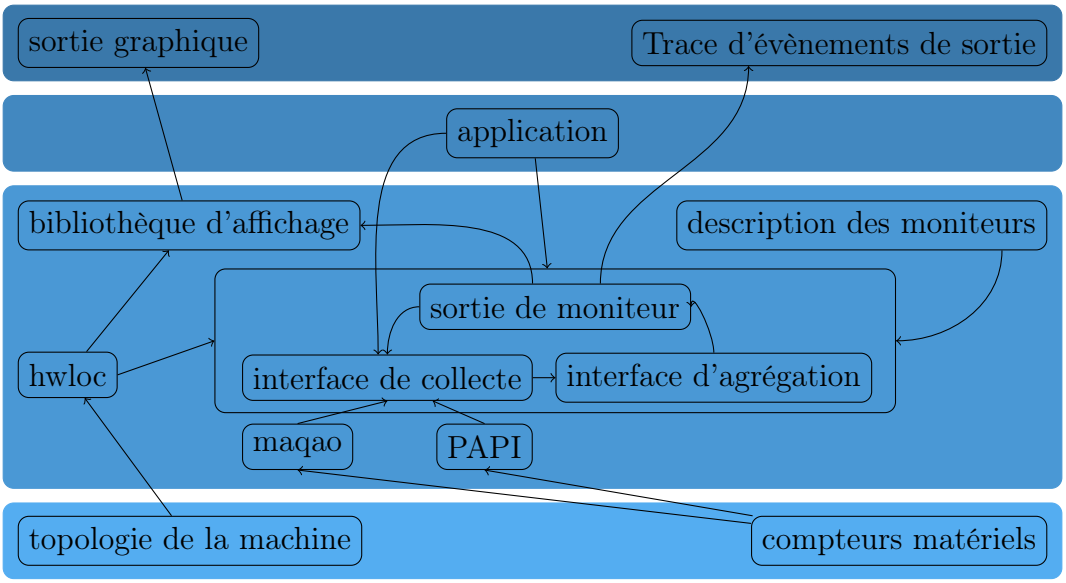


FIGURE 5.6 – Organisation de la bibliothèque de mesure d'évènement sur la topologie matérielle.

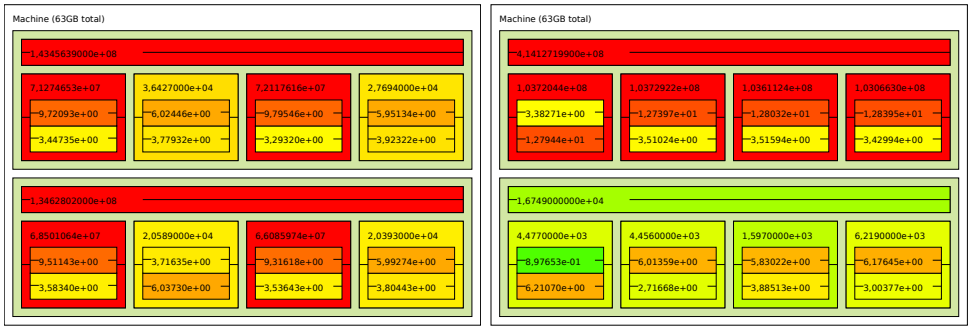
```
//Initialisation avec la topologie courante
hmon_lib_init(NULL);
//Importation de la description des moniteurs
hmon_import_hmonitors(file);
//Début du comptage d'évènements.
hmon_start();

//Relève des évènements dont la valeur est marqué du drapeau : 0
hmon_update(0);
sleep(1);
//Relève des évènements dont la valeur est marqué du drapeau : 1
hmon_update(1);

//Relève des évènements toutes les millisecondes
hmon_sampling_start(10000);
system("sleep 1");
hmon_sampling_stop();

//Fin d'utilisation de la bibliothèque
hmon_lib_finalize();
```

FIGURE 5.7 – Exemple d'utilisation de la bibliothèque de moniteurs à l'intérieur d'un code en C.



(a) Placement équilibré (b) Placement ordonné

FIGURE 5.8 – Sortie graphique du parcours de listes chaînées avec la bibliothèque de moniteurs hiérarchiques.

5.2.5 Application à un cas d'utilisation

En Figure 5.8, on a appliqué la mesure hiérarchique d'évènements de manière similaire à celle décrite par les moniteurs en Figure 5.2. Sur chaque cœur sont accumulés les défauts de cache comptés par les *PU* fils. Sur chaque cache L3 sont accumulés les défauts de cache comptés par les cœurs fils. Sur chaque *PU* sont affichés le pourcentage d'utilisation de l'unité de calcul pour l'application. La couleur variant de vert à rouge montre la distance de la valeur de l'évènement mesuré au maximum atteint au cours de l'exécution échantillonnée et dont on a pris une capture statique. On peut ainsi visuellement observer les threads matériels sur lesquels s'exécutent l'application par leur couleur plus foncée, *i.e.* un par cœur. Dans le scénario équilibré (Figure 5.8a), le nombre de défauts de cache dans les derniers niveaux de cache sont du même ordre de grandeur. A fortiori, leur couleur est semblable. De plus on observe que le placement voulu est respecté, en distinguant les cœurs qui accèdent le plus au cache (en rouge), entrelacés avec ceux qui y accèdent moins (en jaune). Dans le scénario regroupé (Figure 5.8b), on observe le large déséquilibre des défauts de cache, avec un cache rouge pour lequel le nombre de défauts de cache est de l'ordre de 10000 fois plus élevé que le cache vert. Ici encore, on peut vérifier que le placement est bien respecté par la couleur des cœurs rouges pour le premier domaine NUMA, et jaunes pour le second.

5.2.6 Conclusion

La performance dépend de l'interaction du code et des différent composants du système de calcul. On ne peut pas simuler entièrement cette complexité dans un temps raisonnable. Nous avons proposé un outil basé sur une abstraction de la structure du système et des compteurs associés pour dériver des métriques de performance pertinentes liées à la localité. L'outil est publiquement dispo-

nible [1]. La télémétrie du code de l'outil rapporte qu'un total de $\simeq$ 8000 lignes de code composent cet outil.

1. https://github.com/NicolasDenoyelle/Hierarchical-monitors

Chapitre 6

Sélection automatique de politique de placement des tâches et des données

Les techniques décrites jusqu'ici pour placer les fils d'exécution sur la machine utilisent une approche par le bas se basant sur une connaissance relativement profonde du matériel pour en déduire des stratégies de placement, sans toutefois parvenir à des solutions générales. On peut adopter une autre approche en observant le système dans son ensemble puis en déduisant les caractéristiques des applications prépondérantes pour le placement de leurs tâches et de leurs données. Dans ce chapitre nous adoptons une telle approche avec des techniques issues de l'apprentissage statistique, pour sélectionner automatiquement la politique de placement des threads d'une application et déterminer si seulement les applications sont sensibles au placement.

6.1 Motivations

Les modèles bas-niveau comme le Locality-Aware Roofline Model (*cf.* Section 4.2) se basent sur une connaissance minutieuse du matériel, des causes et des conséquences de l'exécution des applications sur la machine. Mais ils sont rapidement limités lorsqu'ils sont employés pour optimiser des applications extrêmement complexes, faisant intervenir une multitude de composants de la machine. Les plates-formes parallèles sont d'une telle complexité que cette connaissance est nécessairement partielle. Donc, les optimisations proposées à un haut niveau d'abstraction sont issues d'abord de l'intuition, en partant de la connaissance du fonctionnement de certains sous-systèmes, puis sont validées par des bancs de tests spécifiques et enfin par l'optimisation d'applications réelles. De telles optimisations ne peuvent couvrir qu'un sous-ensemble des compromis de localité. Sans une analyse minutieuse, elles ignorent souvent des cas particuliers. Cette approche nécessite donc d'être complétée par une approche exploratoire, qui ne fait pas d'hypothèse préalable sur le fonctionnement interne du système de calcul, pour en déduire les paramètres prépondérants dans le choix des politiques de placement. Dans notre cas, l'approche exploratoire consiste à exécuter plusieurs applications avec des politiques d'exécution variées, tout en collectant des métriques pertinentes à observer. L'analyse statistique des échantillons permet ensuite de déduire des liens entre les métriques observées et les stratégies de placement à adopter. Avec cette méthode, nous voulons classer les applications rapidement à l'exécution selon les objectifs suivants : la sensibilité à la localité, et la politique de placement optimale. Plus généralement, la méthodologie décrite dans ce chapitre permet de classer les applications selon d'autres objectifs et dans des contextes nécessitant une décision rapide et peu coûteuse.

Plusieurs approches se sont intéressées au placement automatique de *threads* en utilisant l'apprentissage statistique. Pour les applications basées sur des interfaces de mémoire transactionnelle, M. Castro et *al.* [16] observent les défauts de cache, la proportion de temps passé dans les transactions, la proportion de temps passé dans les transactions échouées, la politique de détection de conflits de transactions et la politique de résolution des conflits de transaction, pour entraîner un arbre de décision. Ils sélectionnent une politique de placement semblable à celles proposées ci-dessous (Section 6.2.1), pour obtenir entre 6% et 18% d'amélioration moyenne en comparaison du placement par défaut décidé par l'ordonnanceur du système d'exploitation. Wang et O'Boyle [119] utilisent un réseau de neurones artificiels pour prédire le passage à l'échelle des applications OpenMP à partir d'une exécution séquentielle et une machine à vecteurs de support (*Support Vector Machine*) (SVM) pour prédire la politique de placement à partir de plusieurs exécutions parallèles. Ils utilisent le nombre d'instructions dans le code statique, le ratio d'instructions de lecture et d'écriture, le nombre de branchements, le nombre d'itérations de la boucle cible (la

modélisation et le placement visent une partie sensible de l'application), le nombre d'accès au premier niveau de cache, le ratio de branchement en dehors du cache d'instruction, en entrée du modèle, pour d'obtenir en moyenne 37% d'amélioration du temps d'exécution par rapport au support d'exécution OpenMP sur le choix de la politique (parmi les politiques de l'interface OpenMP) et le choix du nombre de threads. Tournavitis et *al.* [115] utilisent le nombre d'instructions du code et de l'exécution, le ratio de lectures/écritures, le nombre de branchements, le nombre d'itérations de boucle, pour classer les applications en fonction de leur politique de placement OpenMP préférée.

Selon la méthodologie décrite par les piliers du domaine [45, 86], une étape préalable à l'entraînement d'un algorithme d'apprentissage, est l'observation des données. Un prédicteur ne peut être efficace que si les exemples utilisés pour l'apprentissage sont de nature à pouvoir généraliser le comportement des applications, et si les paramètres observés sont effectivement liés aux causes qui déterminent le bon placement des threads et des données d'une application. En observant les paramètres, on peut parfois déceler au préalable un schéma caractéristique de l'existance d'un lien concret entre les paramètres sélectionnés et la valeur objectif, ce qui permet de choisir le modèle approprié. Pourtant, les publications d'apprentissage statistique appliquées au placement omettent cette étape pour présenter directement la performance de leur modèle sans en expliquer les raisons. De plus, alors que les expériences citées s'intéressent à la prédiction du passage à l'échelle des applications et de leur placement sur la machine, le cas du placement des données n'est à notre connaissance pas encore traité de cette manière. Enfin, la question du placement des threads et des données dans le cas général est un problème difficile. Sans toutefois tenter de résoudre le problème de manière directe, on peut essayer de répondre au dilemme du choix entre plusieurs politiques de placement des threads et des données, ainsi que des paramètres qui régissent cette décision.

Le reste de ce chapitre s'articule donc de la manière suivante : Dans une première partie, nous décrivons notre méthodologie issue de l'apprentissage statistique pour analyser les applications et entraîner des classificateurs. Enfin, en deuxième partie, nous détaillons la recherche d'un classificateur de sensibilité à la localité, et la recherche d'un modèle de sélection de politique d'exécution.

6.2 Méthodologie

Nous souhaitons observer l'incidence du placement des threads, celle du placement des pages en mémoire et celle du nombre de threads, sur la performance des applications, exécutées sur une machine bi-*socket* (détaillée en Annexe A.2). Nous choisissons un ensemble d'applications (*cf.* Annexe B) représentatives des problèmes résolus sur les systèmes parallèles pour cette étude.

	indice des threads							
	0	1	2	3	4	5	6	7
politique	indice des cœurs							
round-robin	0	1	2	3	4	5	6	7
éparpillé	0	12	6	18	2	14	6	20

TABLE 6.1 – Exemple de placement des 8 premiers threads selon leur indice logique sur les 24 unités de calcul (également numérotés dans l'ordre logique) pour le système représenté en Figure A.5.

6.2.1 Politiques d'exécution des applications

Chaque application est exécutée plusieurs fois, en appliquant une politique d'exécution globale, parmi un produit cartésien de plusieurs politiques de placement des threads, plusieurs politiques d'allocation des données, en utilisant un thread par cœur disponible sur la plate-forme.

Politiques de placement des threads

Les politiques de placement des threads sont choisies parmi les politiques : *round-robin* et éparpillé, et représentées en Table 6.1, pour les 8 premiers threads exécutés sur la plate-forme représentée en Figure A.5. Le placement *round-robin* associe les threads aux cœurs selon leur numérotation logique et favorise les échanges de données de proche en proche. Le placement éparpillé répartit les threads successivement sur la machine de manière à les espacer le plus possible, et favorise l'équilibre de l'usage des ressources.

Politiques d'allocation

Les politiques d'allocation mémoire utilisées sont soit *firsttouch*, soit *interleave*, telles que décrites en sous-section 3.1.2.

Politique de référence

Parmi l'ensemble des politiques (placement de threads, allocation des données), la politique utilisant tous les cœurs de la plate-forme, avec les fils d'exécution placés dans l'ordre logique des cœurs, et les données allouées proches des fils d'exécution qui les initialisent, *i.e.* (*round-robin*, *firsttouch*), est désignée comme étant la politique d'exécution par défaut. En effet, par défaut on ne s'occupe pas de la politique d'allocation des données (*i.e.* on choisit celle par défaut) et si on choisit de placer les fils d'exécution, on les place de la manière la plus simple, *i.e.* dans l'ordre logique. Par la suite, on comparera les durées d'exécution des applications selon une politique d'exécution relativement à cette politique par défaut.

Paramètres mesurés

Pour chaque application et chaque politique (placement de threads, allocation des données), on mesure la durée d'exécution des applications ainsi que plusieurs compteurs matériels. La bibliothèque PAPI [83] fournit une couche d'abstraction pour faciliter la programmation et la lecture des compteurs de performance du système de calcul. Parmi les compteurs disponibles, nous sélectionnons des compteurs de la hiérarchie mémoire dont les valeurs comptées sont susceptibles de caractériser la localité des applications et le besoin ou non d'optimiser leur placement :

- `PAPI_TOT_CYCLE` : Le nombre de cycles d'horloge du processeur pendant l'exécution. Cette valeur peut être différente de la durée d'exécution selon si la fréquence du processeur change au cours de l'exécution. Le nombre de cycles du processeur est plus représentatif du travail total effectué par le processeur que le temps d'exécution.
- `PAPI_LD_INS` : Le nombre total d'instructions de lecture exécutées par tous les cœurs hébergeants un thread de l'application. Cette valeur comparée par exemple au nombre de cycles permet de rendre compte de l'intensivité des accès mémoire en lecture de l'application en comparaison aux autres opérations.
- `PAPI_SR_INS` : Le nombre total d'instructions d'écriture exécutées par tous les cœurs hébergeants un thread de l'application.
- `PAPI_L1_DCM` : Le total de requêtes pour des données dans le premier niveau de cache du cœur et pour lesquelles la donnée ne se trouvait pas dans le cache. Comme présenté en sous-section 2.2.3, si les données se trouvent dans un niveau partagé (*e.g.* L3) plutôt qu'un niveau privé (*e.g.* L1), le coût du partage n'est pas le même. Donc par exemple, le nombre de défauts de cache dans le premier niveau de cache rapporté au nombre d'accès aux données indique la fréquence de présence des données au plus près des cœurs et est donc un indicateur du coût potentiel du partage si les données accédées sont partagées entre plusieurs cœurs.
- `PAPI_L2_DCM` : Le total de requêtes de données au second niveau de cache pour lesquelles la donnée ne se trouvait pas dans le cache. Le cache L2 est le dernier niveau de cache privé. Donc un défaut de cache dans le L2 signifie que la donnée se trouve dans un niveau partagé.
- `PAPI_L3_TCM` : Nombre total de requêtes pour des lignes de cache absentes du dernier niveau de cache (et des niveaux inférieurs) incluant l'exécution spéculative mais pas le préchargement. La donnée se trouve alors, soit dans la mémoire locale, soit dans un cache ou une mémoire distants.
- `perf::NODE-LOADS` : Nombre de requêtes en lecture du processeur servies par le nœud mémoire local.

– `perf::NODE-LOAD-MISSES` : Nombre de requêtes en lecture du processeur servies par un *socket* distant.
– `perf::NODE-STORES` : Nombre de requêtes en écriture du processeur vers par le nœud mémoire local.
– `perf::NODE-STORE-MISSES` : Nombre de requêtes en écriture du processeur vers un *socket* distant.

Le nombre de compteurs simultanément programmables étant limité à 4 par unité de calcul sur le système cible, nous collectons ces compteurs sur plusieurs exécutions de la même application dans les mêmes conditions. Chaque échantillon est exécuté 6 fois, et les valeurs médianes des temps d'exécution et des compteurs sont retenues.

6.2.2 Modèles de classification

Pour chacun de nos objectifs, *i.e.* détection de la sensibilité à la localité et choix de la politique d'exécution, nous entraînons plusieurs types de classificateurs, dont nous décrivons certains principes fondamentaux ci-dessous.

Arbre de décision

Dans ce qui suit, supposons que l'on veuille classer les applications : celles qui sont sensibles à la localité et celles qui ne le sont pas, selon la valeur des compteurs de performance PAPI_L2_DCM et perf : :MINOR-FAULTS. Sur un ensemble d'apprentissage, connaissant la classe des applications en fonction des évènements, on peut calculer les probabilités conditionnées par la valeur des évènements d'obtenir une issue, *e.g.* la probabilité que les applications soient sensibles à la localité, sachant que le nombre d'évènements `PAPI_L2_DCM` est supérieur à un seuil θ_1 fixé. On choisit une valeur de θ_1 qui minimise le nombre de faux positifs et de faux négatifs. Cet exemple de scénario dessine un arbre de decision à deux branches dont le nœud racine représente le test `PAPI_L2_DCM > \theta_{1}` et les deux arêtes filles sont les issues possibles selon le résultat. Avec le théorême de Bayes [1] sur les probabilités conditionnelles, on peut poursuivre le développement de l'arbre en Figure 6.1 en composant les évènements. Les feuilles de l'arbre restent les issues possibles, *i.e.* OUI si l'application préfère la politique par défaut, NON sinon sur la Figure 6.1. On peut construire un deuxième étage, avec l'évènement `perf::MINOR-FAULTS` et un autre seuil θ_2. La probabilité que les applications préfèrent la politique par défaut avec perf : :MINOR-FAULTS $> \theta_2$, sachant PAPI_L2_DCM $> \theta_1$ (première feuille de l'arbre) est égale au nombre d'applications préférant la politique par défaut lorsque PAPI_L2_DCM $> \theta_1$ ET perf : :MINOR-FAULTS $> \theta_2$ divisé par le nombre d'applications préférant la politique par défaut lorsque

1. $P(A|B) = \frac{P(A \cap B)}{P(B)}$

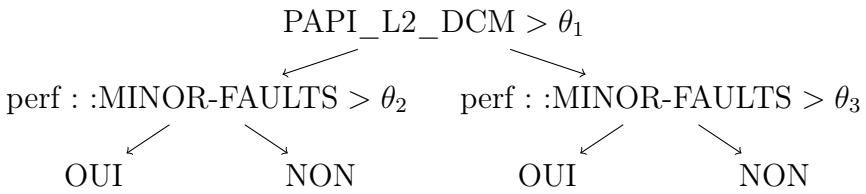


FIGURE 6.1 – Arbre de décision hypothétique pour décider si une application préfère l'exécution avec la politique par défaut sachant la valeur des évènements PAPI_L2_DCM et perf : :MINOR-FAULTS pour chacune.

PAPI_L2_DCM > θ_1. Dans ce cas on cherche θ_1 et θ_2 tels que les probabilités de réalisation des issues aux feuilles soient maximales. Enfin, une fois l'arbre construit, pour classer les nouvelles applications, il suffit de parcourir l'arbre en fonction de la valeur des paramètres mesurés et des seuils à chaque nœud, avant d'arriver à une feuille qui indique l'issue la plus probable. Diverses boîtes à outil basées sur les arbres de décision proposent des techniques plus avancées pour traiter des problèmes plus généraux et de manière plus efficace. Parmi ceux-là, nous choisissons le modèle de forêt d'arbres décisionnels (Random-Forest) [54, 6]. L'avantage de ce genre de modèle est qu'il permet de quantifier l'importance d'un paramètre en fonction de sa position dans l'arbre. Par exemple, si `perf::MINOR-FAULTS` est déterminant dans la probabilité d'une issue alors que `PAPI_L2_DCM` a peu d'incidence sur le choix du placement alors le niveau `perf::MINOR-FAULTS` sera au dessus du niveau `PAPI_L2_DCM` dans l'arbre de décision final.

Régression logistique

La régression logistique [122] est une variation de la régression linéaire qui transforme la somme pondérée des entrées $(X_1, X_2, \ldots, X_n)$ et des poids $(\theta_1, \theta_2, \ldots, \theta_n)$ pour que la sortie du réseau soit à valeur dans $[0, 1]$ et la fonction de coût J_θ pour qu'elle reste convexe, *i.e.* avec un minimum global. Le fonctionnement de la prédiction d'une régression logistique et du calcul de la fonction de coût est représenté en Figure 6.2. La matrice dont les lignes représentent chacune une expérience (*i.e.* une application) et les colonnes chaque évènement (*i.e.* un compteur matériel), est multipliée par le vecteur de poids. Puis le vecteur résultat aux dimensions de la sortie à prédire (*i.e.* la politique optimale) est passée dans une fonction sigmoïde pour donner l'hypothèse de résultat h_θ. À l'instar de la régression linéaire, la fonction de coût J_θ (ici *cross-entropy* [107]) est une fonction convexe qui calcule une distance entre la sortie du modèle et les sorties à trouver. La fonction de coût est dérivée le long des paramètres θ, itérativement à chaque exemple, pour en trouver le minimum global, c'est-à-dire la valeur des poids qui minimise cette distance, pour tous les échantillons observés. Dans le cas d'un classificateur de politique par appli-

cation, une sortie y du modèle correspondrait à un vecteur booléen de taille le nombre de politiques et dont les valeurs sont nulles partout sauf sur la politique qui donne le meilleur temps d'exécution. Sur la Figure 6.2, Y serait donc une matrice dont les lignes représentent les applications, et les colonnes les probabilités qu'une politique soit la meilleure pour une application. En pratique, la valeur de sortie de ce genre de modèle n'est pas entière et est interprétée comme une probabilité que la sortie soit la bonne. Ici, comme on a qu'une seule sortie (sensible ou non à la localité), on considère que si la sortie est supérieure à 0,5 alors elle vaut 1 (sensible), sinon 0 (non sensible). Si on a plusieurs sorties, on choisit celle dont la valeur est le maximum des probabilités en sortie. L'avantage de ce modèle, est que les poids définissent l'équation linéaire d'un séparateur (ou hyper-plan) entre les échantillons classés. Donc, en projetant les échantillons sur toutes les paires de paramètres, si on ne voit pas de frontière linéaire entre les échantillons de chaque classe, il y a fort à parier que le modèle ne marchera pas très bien. Par exemple, sur la Figure 6.3 on observe que sur une paire de paramètres $x1$ et $x2$, les points des deux classes observées peuvent être séparés par une droite. Si ce n'était pas le cas, on ne pourrait pas trouver de séparateur sous la forme d'une droite qui permettent de décider de la classe du point. Enfin, les poids du modèle indiquent l'importance des paramètres, et donc les critères d'importance pour choisir la politique de placement.

Machines à vecteurs de support (SVM)

Le modèle de machine à vecteurs de support (*Support Vector Machine*) (SVM) [11] est une généralisation de la régression logistique. Elle fonctionne par le calcul d'une séparation entre les évènements labélisés. Cependant l'algorithme du modèle SVM ne calcule pas un mais deux séparateurs parallèles. Ces deux séparateurs sont appelés vecteurs de support et leur distance ou « marge » est optimisée par l'algorithme. L'utilisation de ces deux hyper-plans permet de traiter les cas où plusieurs frontières équivalentes peuvent être calculées par une régression logistique, mais pour lesquelles de nouveaux exemples proches de cette frontière peuvent être bien ou mal classés selon la frontière. Par exemple, en Figure 6.3 nous représentons sur un plan, des échantillons étiquetés parmi 2 classes (croix bleu ou rond vert) et un séparateur linéaire calculé par une regression logistique (droite noire), avec un séparateur linéaire calculé par une SVM (droite rouge), dont on représente la marge en rouge clair autour du séparateur. Sur cet exemple, nous montrons que la croix entre les deux séparateurs (en gradiant bleu/vert pour montrer qu'il s'agit d'un nouvel échantillon) de la Figure 6.3 est mal classé par rapport à la frontière de la régression logistique, tandis qu'il est bien classé selon la frontière de la SVM. Les séparateurs décrits par le modèle SVM permettent de calculer une frontière entre les marges plus robustes à la généralisation. De plus, cette modélisation permet de réduire la complexité

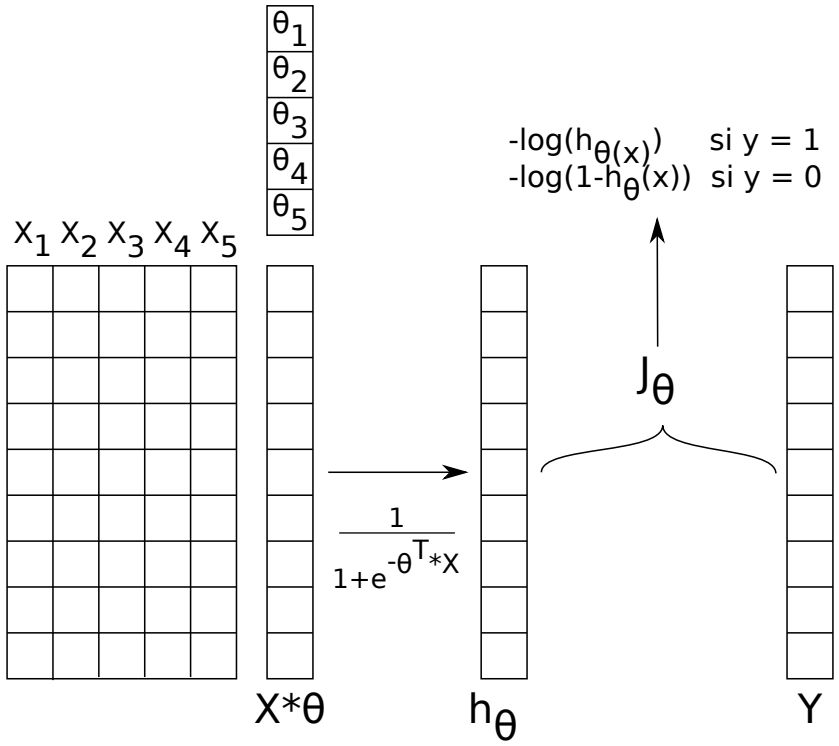


FIGURE 6.2 – Représentation du calcul des sorties h_θ d'une régression logistique et du calcul de la fonction de coût J_θ du modèle en fonction des entrées $(X_1, X_2, \ldots, X_n)$, des poids du modèle $(\theta_1, \theta_2, \ldots, \theta_n)$ et des sorties Y.

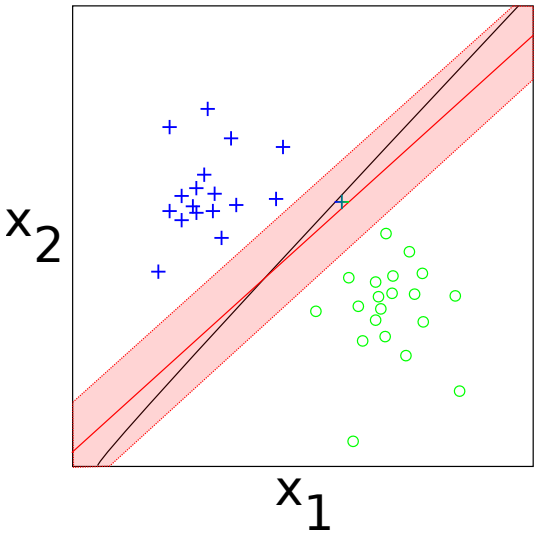


FIGURE 6.3 – Représentation des frontières décrites par une regression logistique (en noir) et par une machine à vecteurs de supoprt (SVM) (en rouge) pour un ensemble d'échantillons caractérisés par les évènements x_1 et x_2 et étiquetés par une croix bleue ou un rond vert.

des calculs lorsqu'il s'agit d'appliquer des transformations non-linéaires sur les paramètres pour transformer une frontière non-linéaire en frontière linéaire (*cf.* sous-sous-section 6.2.3). Dans notre cas, nous utilisons systématiquement ce modèle en conjonction avec une transformation gaussienne qui change les entrées en leur probabilité d'apparition selon une loi normale paramétrée par la moyenne et la déviation de l'ensemble d'apprentissage.

Nous utilisons les trois modèles décrits dans cette sous-section pour classer les applications selon leur sensibilité à la localité dans un premier temps, et leur politique d'exécution préférée dans un second temps. Avant d'entraîner les modèles, nous procédons à une série de transformation des données et de sélection des échantillons décrits ci-dessous.

6.2.3 Pipeline de classification

La chaîne de traitement que nous utilisons contient les éléments suivants dans l'ordre :

- Pré-traitements des données pour aider l'algorithme d'apprentissage à être plus efficace.
- Réduction de dimensionnalité, pour réduire la complexité du modèle et le risque de sur-apprentissage.
- Séparation des données qui servent à paramétrer le modèle des données qui servent à vérifier son efficacité.

- Entraînement des modèles.
- Analyse de la qualité des modèles.

Transformation des entrées

Les compteurs matériels bruts tels que nous les avons collectés ne sont pas directement de bons paramètres pour caractériser les applications. Par exemple, le nombre total de défauts de cache dans le premier niveau de cache dépend de la durée de l'exécution. Si on exécute une application deux fois, le total de défauts de cache sera approximativement doublé mais les propriétés de l'application seront les mêmes vis-a-vis des problèmes de localité. Il paraît donc nécessaire de rapporter cette valeur à une caractéristique intrinsèque à l'exécution de l'application, comme le nombre d'accès mémoire, *i.e.* `PAPI_LD_INS` + `PAPI_SR_INS`. Donc la première étape de transformation des entrées est une re-combinaison des compteurs en valeurs relatives :

- `LST_ratio` = `PAPI_LD_INS` / `PAPI_SR_INS` : Le ratio d'instructions de lecture par rapport au nombre d'instructions d'écriture.
- `LD_INSTANT` = `PAPI_LD_INS` / `PAPI_TOT_CYC` : Le débit de l'application en lecture, en instructions par cycle.
- `SR_INSTANT` = `PAPI_SR_INS` / `PAPI_TOT_CYC` : Le débit de l'application en écriture, en instructions par cycle.
- `L1_MISS_REL` = `PAPI_L1_DCM` / (`PAPI_LD_INS` + `PAPI_SR_INS`) : Le nombre de défauts de cache dans le L1 relativement au nombre d'accès mémoire de l'application.
- `L2_MISS_REL` = `PAPI_L2_DCM` / (`PAPI_LD_INS` + `PAPI_SR_INS`) : Le nombre de défauts de cache dans le L2 relativement au nombre d'accès mémoire de l'application.
- `L3_MISS_REL` = `PAPI_L3_TCM` / (`PAPI_LD_INS` + `PAPI_SR_INS`) : Le nombre de défauts de cache dans le L3 relativement au nombre d'accès mémoire de l'application.
- `NODE_MISS_REL` = (`perf::NODE_STORE_MISSES` + `perf::NODE_LOAD_MISSES`) / (`PAPI_LD_INS` + `PAPI_SR_INS`) : Le nombre de défauts d'accès dans la mémoire locale relativement au nombre d'accès mémoire de l'application.

Selon les algorithmes d'apprentissage il peut être nécessaire de normaliser les données au préalable. C'est par exemple le cas pour l'utilisation de régression logistique, tandis que ce n'est pas nécessaire dans le cas des arbres de décision et algorithmes dérivés. La mise à l'échelle des valeurs entre 0 et 1 permet de couvrir toutes les contraintes de tous les modèles pour notre cas. Cette mise à l'échelle est calculée sur l'ensemble d'apprentissage, puis appliquée à l'identique sur l'ensemble de validation.

Il est possible d'effectuer n'importe quelle transformation non linéaire sur les entrées du modèle avant de l'entraîner. Par exemple, soit un ensemble d'échantillons constitué par deux classes dont le séparateur est une ellipse, c'est-à-dire que dans le plan des deux évènements x_1 et x_2 observés, les points d'une classe sont contenus dans une ellipse d'équation $\frac{x_1^2}{a}+\frac{x_2^2}{b} = c$, tandis que les autres sont à l'extérieur de l'ellipse. On représente pour notre cas d'utilisation une frontière elliptique pour le classement des applications sensibles à la localité en fonction des deux évènements `SR_INSTANT` et `LD_INSTANT` en Figure 6.8a. Dans ce cas x_1 correspond à `SR_INSTANT` et x_2 correspond à `LD_INSTANT`. Sans transformation, la régression logistique donne un séparateur linéaire sous la forme d'une droite d'équation : $\theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2 = 0$, où les paramètres θ sont calculés par la régression pour minimiser l'erreur du classificateur. En multipliant les entrées entre elles on peut à la place chercher un séparateur d'équation : $\theta_0 + \theta_1 \times x_1^2 + \theta_2 \times x_2^2$ avec une régression logistique où $\theta_0 = -c$, $\theta_1 = \frac{1}{a}$ et $\theta_2 = \frac{1}{b}$ et décris l'équation d'une ellipse. Par conséquent, parmi les transformations, nous testons plusieurs degrés de polynôme [2], sur les entrées, avant d'entraîner les classificateurs dans le but de chercher une frontière non linéaire entre les échantillons.

Sorties des modèles

Pour chaque modèle, les entrées sont les paramètres de la prédiction et les sorties sont des vecteurs booléens dont chaque valeur vaut $FAUX$ si ce n'est pas la classe de l'application, $VRAI$ si il s'agit de la bonne classe. On entraîne deux classificateurs pour deux objectifs : déterminer la sensibilité des applications à la localité et déterminer la politique préférée de applications.

Définition 6.1. On décide qu'une application est sensible à la localité s'il existe une politique pour laquelle le temps d'exécution varie d'au moins 10% par rapport à la politique par défaut.

Définition 6.2. On définie la politique préférée d'une application comme celle qui permet d'atteindre le temps d'exécution minimum.

Réduction du nombre de paramètres

Lorsqu'un modèle possède autant de paramètres que d'échantillons, il est possible de trouver de manière certaine un ensemble de poids tels que la projection par le modèle de l'ensemble d'apprentissage dans l'ensemble des solutions soit exactement égale à celui de l'ensemble des éléments cibles, c'est-à-dire que le modèle a appris par-cœur. Par exemple le théorême d'interpolation des polynômes assure qu'on peut trouver un polynôme de degré n passant par $n+1$ points, donc entraîner un modèle avec plus de paramètres que d'échantillons

2. *e.g.* x_1, $x_2 \rightarrow x_1$, x_2, $x_1 \times x_2$, x_1^2, x_2^2 au degré 2

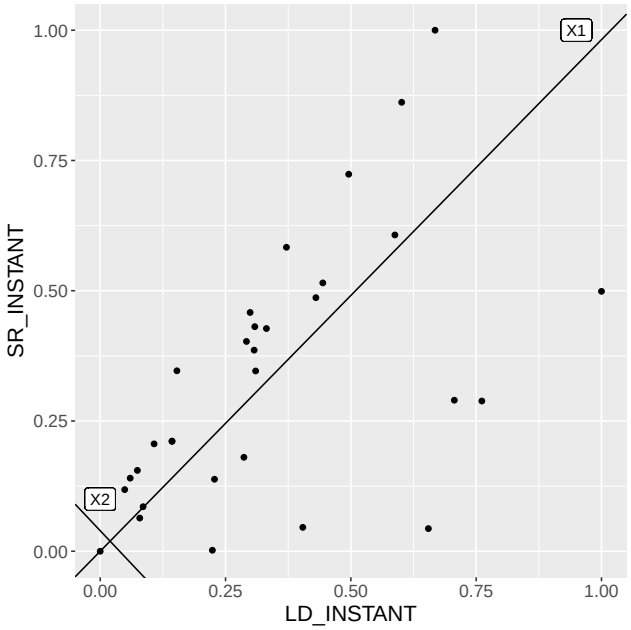


FIGURE 6.4 – Représentation des compteurs normalisés LD_INSTANT et SR_INSTANT pour chaque applications exécutée avec la politique par défaut. Les axes X1 et X2 correspondent aux vecteurs de la base calculée par la décomposition en valeurs singulières des échantillons.

devrait permettre de trouver un résultat exact. Cependant lorsque de nouveaux échantillons sont présentés au modèle, celui-ci échoue à prédire correctement la valeur cible. Ce phénomène est appelé *overfitting* et la réduction de dimensionnalité est une des solutions à ce problème. De plus dans le cas particulier des forêts d'arbres décisionnels, la complexité en temps de l'apprentissage est exponentielle en le nombre de paramètres, et les implémentations usuelles ne permettent pas d'excéder un nombre raisonnable de paramètres. Pour pallier à ce problème on peut réduire le nombre de paramètres en utilisant l'analyse en composantes principales.

Si deux variables sont très corrélées entre elles, on peut en supprimer une sans perdre d'information sur les relations entre les paramètres du modèle et la sortie. La Décomposition en Valeurs Singulières (SVD) [3] est une généralisation de cette méthode. L'algorithme SVD décompose une matrice $X_{n,m}$ d'entrées en un produit $U_{n,n} \times D_{n,m} \times V^T_{m,m}$ tel que $U^T \times X$ correspond à un changement de base de X dans une nouvelle base orthonormée où les vecteurs de la base sont colinéaires avec les droites de corrélations entre les m dimensions de X. Par exemple, en Figure 6.4, nous représentons en $X1$ et $X2$ les axes de la nouvelle base calculée par la SVD de X, *i.e.* des applications selon les compteurs `LD_INSTANT` et `SR_INSTANT`. On observe que la droite $X1$ est celle qui mini-

3. https://fr.wikipedia.org/wiki/Décomposition_en_valeurs_singulières

mise la distance des points à la droite, et $X2$ est orthogonale à $X1$. Dans cette nouvelle base, si on projette les points sur la droite $X1$, *i.e.* si on effectue le changement de base et que l'on garde uniquement la dimension $X1$ des points, on aura réduit le nombre de dimensions de 2 à une, tout en minimisant la perte d'informations. Cette perte d'information correspond à la distance des points à la droite $X1$ et est contenue dans le premier élément de la matrice diagonale D.

Donc la SVD permet de réduire le nombre de dimensions, en minimisant la perte d'information, tout en sachant la quantité d'information perdue selon le nombre d'axes conservés. Dans notre cas, nous supprimons les axes de U qui contiennent le moins d'information jusqu'à un seuil de 10% d'information perdue. Cette technique de décomposition en valeurs singulières présente toutefois le désavantage de grandement réduire l'interprétabilité du modèle. En effet, les poids calculés ne quantifient plus la participation d'un paramètre connu mais plutôt celle de la projection de plusieurs paramètres combinés.

Ensembles d'apprentissage et de validation Selon la quantité d'échantillons à disposition on peut utiliser des techniques simples ou plus raffinés pour sélectionner les données qui servent à positionner les poids du modèles et celles qui servent à vérifier la performance du modèle. Tout d'abord, il convient de mélanger les échantillons aléatoirement, afin de mitiger au maximum les biais issus d'interactions entre ces derniers, ou bien de diminuer les chances de stopper l'apprentissage dans un minimum local [108]. Ceci étant fait, on choisit par exemple les trois premiers quarts des échantillons pour entraîner le modèle et le quart restant pour le valider. Lorsque le nombre d'échantillons est très limité, comme c'est le cas ici (seulement 30 applications, *cf.* Annexe B), on enlève un seul échantillon (une seule application) de l'ensemble d'apprentissage. On réitère le procédé pour chaque application puis on moyenne la performance des 30 modèles entraînés sur 29 applications et validés sur une seule.

6.2.4 Qualité du modèle

Finalement, pour vérifier la qualité d'un modèle, l'apprentissage statistique nous munit de plusieurs métriques génériques, associées aux objectifs des modèles. Dans le cas des classificateurs, on distingue en particulier quatre valeurs directement interprétables :

- le nombre de faux positifs, *i.e.* les échantillons classés à tord dans une catégorie cible.
- le nombre de faux négatifs, *i.e.* les échantillons classés à tord en dehors d'une catégorie cible.
- le nombre de vrai négatifs, *i.e.* les échantillons classés à raison en dehors d'une catégorie cible.

– le nombre de vrai positifs, *i.e.* les échantillons classés à raison dans une catégorie cible.

Lorsqu'on classe des échantillons sur plus de 2 classes, une méthode similaire et visuellement expressive est de calculer une *matrice de confusion* dont les lignes sont la classe réelle des échantillons et les colonnes sont les classes prédites des échantillons.

Dans le cas particulier de la sélection de politique, on n'est pas seulement intéressé par le taux d'erreur du modèle mais également par son impact sur le temps d'exécution, c'est-à-dire si les erreurs concernent plutôt les échantillons peu sensibles à la localité et facilement confondus avec ceux qui ne le sont pas du tout, ou bien ceux qui sont très sensibles à la localité. Pour évaluer ce compromis, nous présentons 3 métriques : le *speedup* [4] moyen des applications : avec la meilleure politique, avec la pire politique, et enfin avec la politique d'exécution décidée par le classificateur.

6.3 Classification d'applications pour la sensibilité à la localité et le choix d'une politique d'exécution

Dans cette section nous mettons en pratique la méthodologie détaillée dans la section précédente. Dans un premier temps nous observons les variations de performances et de localité des applications selon la politique d'exécution. Puis dans un second temps nous construisons et évaluons des modèles pour les classer automatiquement.

6.3.1 Politique de placement et performance des applications

Nous exécutons les 30 applications décrites en Table B.1 sur le système décrit en Figure A.5. Dans la Figure 6.5, nous représentons chaque application pour 2 politiques (partie haute et partie basse de la figure). La partie haute montre le *speedup* des applications par rapport à la politique par défaut (1 thread par cœur, placement *round-robin*, allocation *firsttouch*) avec leur meilleur politique respective. La partie du bas montre le *speedup* par rapport à la politique par défaut, avec la pire politique pour chaque application, lorsque tous les cœurs sont utilisés. Dans ce deuxième cas on omet volontairement les applications qui sont moins rapides lorsqu'elles utilisent moins de cœurs car cela représente une large majorité des applications et car cela cache la variété

4. Temps d'exécution de l'application avec la meilleure des politiques divisé par le temps avec la politique de référence.

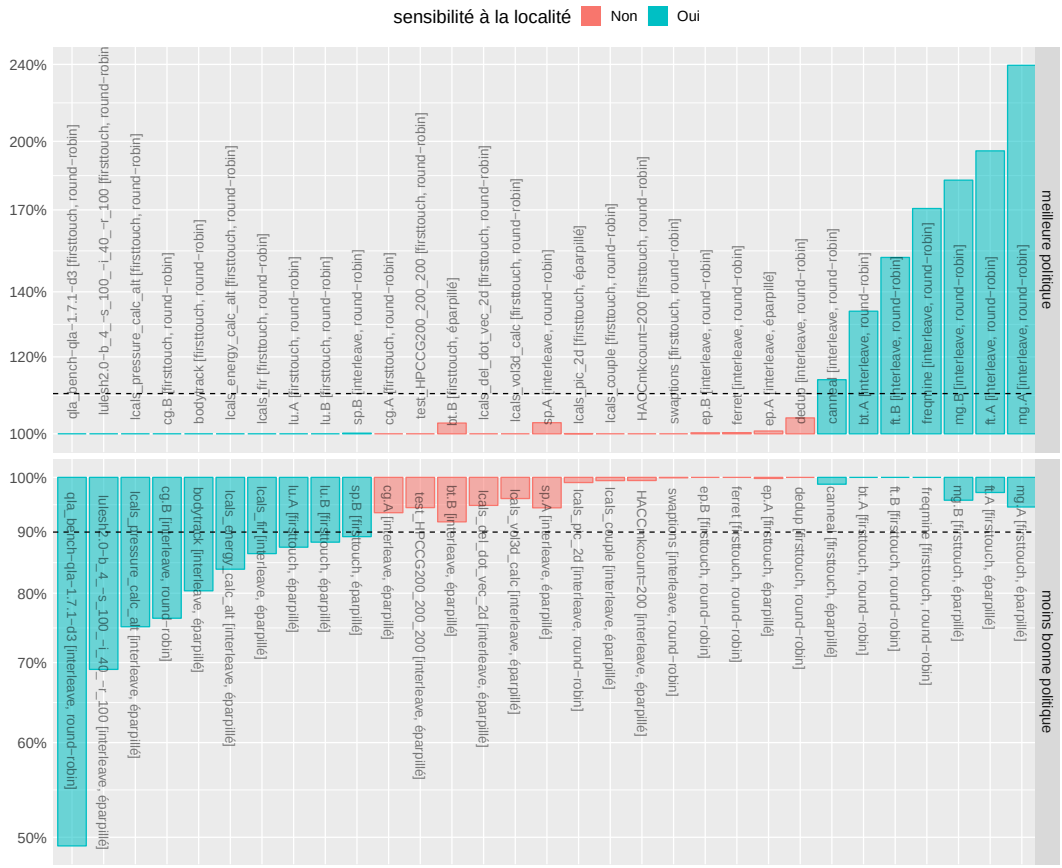


FIGURE 6.5 – *Speedup* des applications en fonction de la meilleure et de la pire politique pour chaque application décrite en Table B.1.

des causes de ralentissement. On représente également à 10% et -10% de *speedup* une barre horizontale pointillée séparant les applications qualifiées comme sensibles à la localité, *i.e.* avec un *speedup* supérieur ou inférieur de 10% et celles qui ne le sont pas, *i.e.* avec un *speedup* entre 10% et -10%. On constate que certaines applications peuvent être ralenties jusqu'à environ 50% et accélérées jusqu'à environ 2,5 fois. Donc le placement des threads et des données à un impact clair sur la performance. De plus, on voit que toutes les applications ne sont pas sensibles aux politiques choisies.

6.3.2 Politique de placement et localité des applications

Pour ces applications, le placement impacte aussi la localité dans le matériel. En effet, en Figure 6.6, nous représentons chaque application exécutée avec tous les cœurs, la politique d'allocation *interleave* et la politique de placement « éparpillé », par un point. En abscisse est représenté le nombre de défauts (absence de la donnée) lors d'une requête dans un niveau de mémoire,

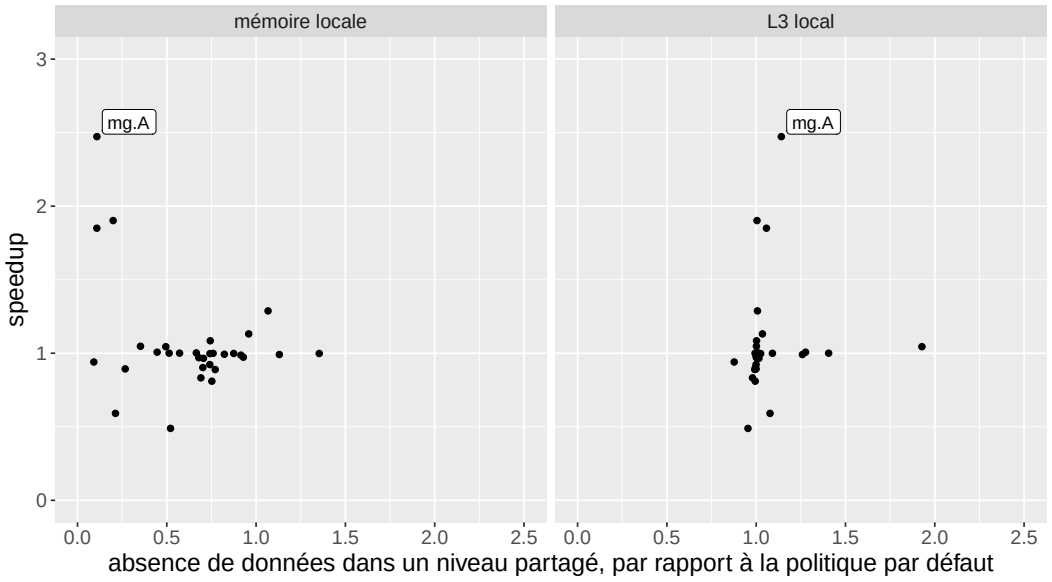


FIGURE 6.6 – Variation des compteurs de défauts (absence de donnée lors d'une requête) des niveaux de mémoire locaux et partagés pour les applications en Table B.1, exécutées sur la plate-forme représentée en Figure A.5. Les variations sont mesurées par rapport à la politique par défaut en utilisant aussi tous les cœurs mais avec un placement de threads « éparpillé » et la politique d'allocation *interleave*.

rapporté au même évènement lorsque l'application est exécutée avec la politique par défaut. De même, en ordonnée, on représente le temps d'exécution rapporté au temps d'exécution de l'application avec la politique par défaut. La Figure est découpée en deux niveaux de mémoire, le cache partagé et la mémoire locale. Dans la partie gauche comme dans la partie droite, on constate que les points sont dispersés autour de du point (1,1). Ce point correspond au fait que le temps d'exécution et la valeur des compteurs ne changent pas lorsqu'on utilise cette politique d'exécution en comparaison de la politique par défaut. Nous notons l'application *mg.A* sur la Figure 6.6, pour observer que la localité et le temps d'exécution sont affectés par la politique d'exécution. En effet, *mg.A* voit son temps d'exécution diminuer pour être environ 2,5 fois plus rapide que sont homologue exécutée avec la politique par défaut. Dans le même temps, *mg.A* fait plus d'accès dans la mémoire locale (nombre de *miss* dans la mémoire inférieur à 1), mais plus de défauts de cache dans le cache partagé. Donc la politique d'exécution (*interleave*, éparpillé) fait changer la valeur des compteurs de défauts dans chacun des niveaux de mémoire partagée, et donc la localité, mais aussi le temps d'exécution.

6.3.3 Classification de la sensibilité à la localité

Pour classer les applications selon leur sensibilité à la localité, on cherche à construire un séparateur entre les points étiquetés « sensibles à la localité » et les autres. La Figure 6.7 est une matrice de représentation de toutes les paires de paramètres (normalisés) décrits en sous-section 6.2.3. Chaque application, exécutée avec la politique par défaut, est représentée par un point sur chaque graphe. Sur chaque colonne et chaque ligne est représenté un compteur. Dans la partie inférieure gauche les applications sont représentées par des points selon deux paramètres (`LD_INSTANT` et `SR_INSTANT`) à chaque graphe, tandis que la partie supérieure droite représente la corrélation (de Spearman [5]) et p-valeur associée pour chaque paire de compteurs.

Sur la Figure 6.7, on recherche des schémas caractéristiques qui permettent d'avoir des indices sur la forme du séparateur *i.e.* l'équation d'un hyper-plan de l'espace vectoriel des paramètres du modèle qui sépare l'espace en classes distinctes. Chacun des échantillons pourrait alors être classé selon le côté du séparateur où il se trouve. Ici, on n'observe pas de frontières évidente permettant de séparer les points verts (sensibles à la localité) des points noirs (insensibles).

Néanmoins, nous proposons d'entraîner et de visualiser l'efficacité d'un séparateur linéaire pour classer les applications. Avec toutes les applications, nous entraînons un séparateur qui classe la sensibilité à la localité en fonction des dimensions `LD_INSTANT` (*i.e.* le débit de l'application en lecture) et `SR_INSTANT` (*i.e.* le débit de l'application en écriture). Ces deux évènements sont préalablement normalisés, et on a effectué une transformation polynomiale (*cf.* sous-sous-Section 6.2.3) de degré 2 en Figure 6.8a et de degré 4 en Figure 6.8b avant la classification. Sur les Figures 6.8 l'abscisse correspond à l'évènement normalisé `LD_INSTANT`, l'ordonnée correspond à l'évènement normalisé `SR_INSTANT`, la couleur correspond à la sensibilité (ou non) à la localité. Les points en triangle représentent les applications, tandis que l'aire de la Figure est colorée selon le résultat du classificateur et permet de déceler la frontière décrite par ce dernier. On peut voir à l'œil nu, avec seulement 2 dimensions, que déjà, au degré 2, le classificateur décrit une ellipse avec peu de faux positifs, et peu de faux négatifs. Le classificateur de degré 4 est encore plus efficace, mais compte déjà un nombre total de 14 paramètres pour une trentaine d'applications, et donc sera probablement victime du problème d'*overfitting* (voir sous-sous-section 6.2.3) qui diminuera la capacité du modèle à généraliser à de nouveaux exemples.

Nous entraînons plusieurs modèles de classification de la sensibilité à la localité, selon la méthodologie décrite en Section 6.2. En Table 6.2, nous présentons les *matrices de confusion* (*cf.* sous-section 6.2.4) pour le modèle linéaire pour plusieurs degrés de transformations polynomiales, en utilisant seulement

5. https://fr.wikipedia.org/wiki/Corrélation_de_Spearman

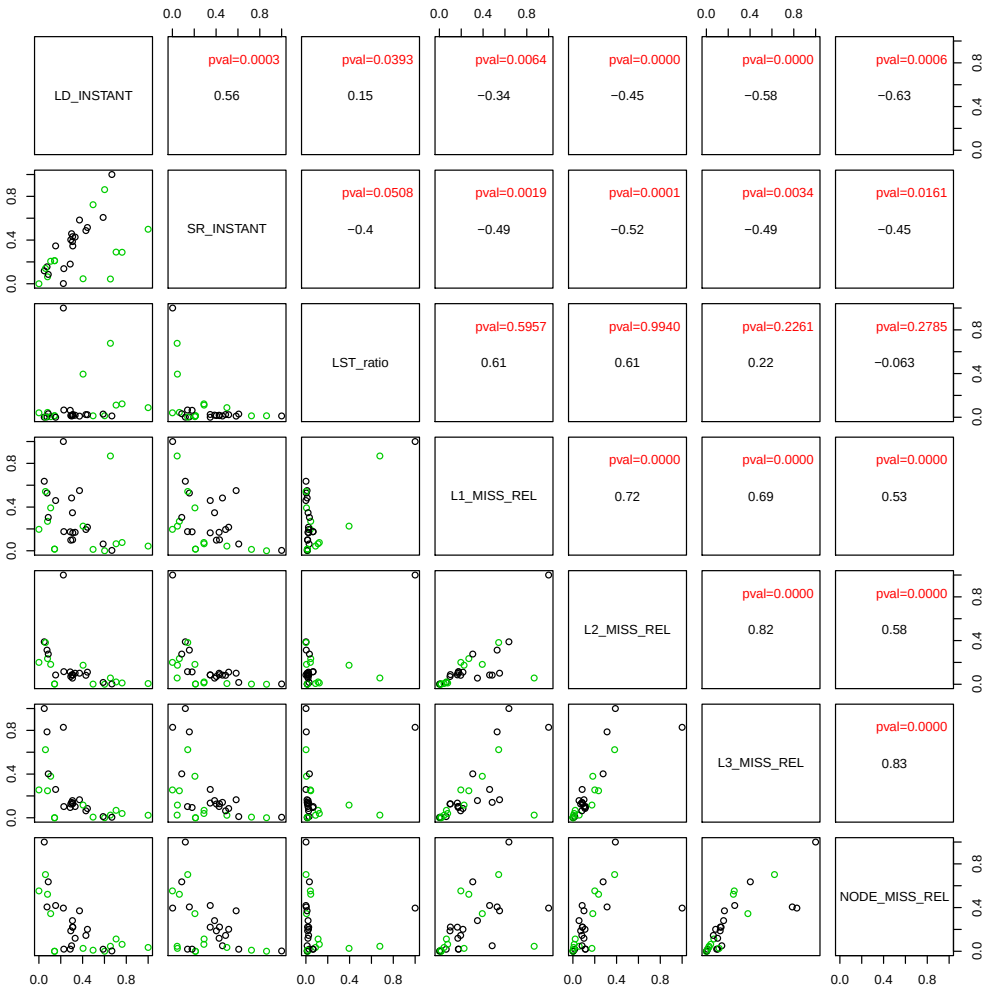
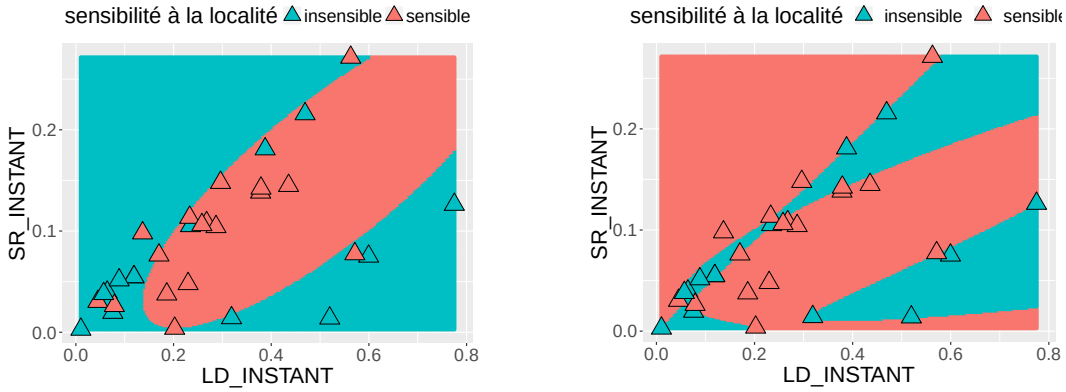


FIGURE 6.7 – Représentation graphique de toutes les paires de paramètres définis en sous-section 6.2.3.



(a) Avec une transformation polynomiale de degré 2

(b) Avec une transformation polynomiale de degré 4

FIGURE 6.8 – Représentation graphique des applications selon deux évènements en abscisse et en ordonnée. Les applications sont colorées selon leur sensibilité à la localité. En parallèle nous représentons les résultats d'un classificateur linéaire, avec une transformation polynomiale sur les entrées, avec les mêmes couleurs selon le résultat de la classification.

les paramètres : `LS_INSTANT` et `SR_INSTANT`. Les résultats en Tables 6.2b, 6.2c sont moins bon que sur la Figure 6.8 car l'élément qui sert à tester le modèle a été retiré de l'ensemble d'apprentissage. Enfin on observe que plus le degré du polynôme est élevé, moins le modèle est efficace, ce qui correspond au phénomène de « sur-apprentissage ». En Table 6.3 nous donnons les matrices de confusion pour chacun des types de modèles (SVM, fôret d'arbres décisionnels, régression logistique), en utilisant tous les paramètres proposés, mais pas de SVD ou de transformation polynomiale. On observe que la régression logistique est le modèle qui fait le moins de faux positifs et faux négatifs, avec toutefois 39% d'erreurs. La réduction de dimensionnalité n'améliore pas les résultats. En conclusion, on ne peut pas se baser sur les évènements proposés pour déterminer de manière fiable la sensibilité d'une application avec des modèles génériques, selon nos conditions d'expérience.

6.3.4 Selection automatique de la politique d'exécution

Dans un second temps, on essaye de déterminer si il est possible de prévoir la politique la plus efficace pour les applications à partir d'une seule exécution avec la politique par défaut. Pour cela, on ne choisit que les applications définies comme sensibles à la localité pour entraîner un classificateur. Pour cette classification on n'utilise pas de régression logistique car l'implémentation du modèle dans l'outil d'analyse que nous utilisons ne permet de classer des variables catégorielles à plus de 2 catégories. En

	sensible	non sensible
sensible	6	7
non sensible	4	14

(a) Pas de transformation polynomiale (37% d'erreur).

	sensible	non sensible
sensible	7	6
non sensible	7	11

(b) Polynôme de degré 2 (43% d'erreur).

	sensible	non sensible
sensible	5	8
non sensible	12	6

(c) Polynôme de degré 4 (67% d'erreur).

TABLE 6.2 – Matrices de confusion du modèle linéaire pour la prédiction de la sensibilité à la localité, avec plusieurs degrés de transformations polynomiales, sans SVD et avec les compteurs : SR_INSTANT, LD_INSTANT.

	sensible	non sensible
sensible	5	8
non sensible	5	13

(a) régression logistique (43% d'erreur).

	sensible	non sensible
sensible	4	9
non sensible	7	11

(b) arbres de décision (53% d'erreur).

	sensible	non sensible
sensible	1	12
non sensible	5	13

(c) machines à vecteur de support (57% d'erreur).

TABLE 6.3 – Matrices de confusion des différents types de modèle, pour la prédiction de la sensibilité à la localité, sans transformation polynomiale, et sans SVD, avec tous les compteurs.

remplacement, nous utilisons un réseau de neurones artificiels avec seulement une couche cachée de 4 neurones. Pour la modélisation, l'utilisation d'une transformation polynomiale (de degré 2 ou 3) avec une décomposition en valeurs singulières augmente le nombre de paramètres du modèle de 7 à 17. En revanche, sans transformation polynomiale, la SVD conserve les 7 paramètres. Comme il y a exactement 17 applications sensibles à la localité, les modèles sont en théorie capables d'estimer exactement le modèle passant par tous les échantillons d'apprentissage, en revanche il reste intéressant d'observer si ceux-là sont capable de généraliser à de nouveaux exemples, *i.e.* l'application sortie de l'ensemble d'apprentissage. En Table 6.4 on montre le *speedup* moyen de l'exécution de toutes les applications avec la pire politique, la meilleure politique, et celle prédite par le classificateur. On constate que les politiques proposées par le classificateur sont en moyenne au moins aussi bonne que la politique par défaut, car le *speedup* moyen de la politique prédite, par rapport à la politique par défaut est systématiquement meilleur que 1. On observe que l'utilisation de la SVD ne change pas beaucoup l'efficacité du modèle, et que la transformation polynomiale améliore l'efficacité des modèles. Le modèle le plus efficace pour cette tâche est le réseau de neurones, avec la transformation polynomiale degré 2 et la SVD. C'est aussi le modèle le moins interprétable. Ce modèle permet d'obtenir une amélioration du temps d'exécution de 21% en moyenne contre 29% dans le meilleur des cas.

En conclusion, déterminer si une application est sensible à la localité et sélectionner la meilleure politique n'est pas trivial. Ici nous avons montré sur une plate-forme classique avec des applications représentatives du domaine que l'observation de certains compteurs matériels n'est pas suffisante pour déterminer en une exécution si l'application sera sensible ou pas à la politique de placement des threads et d'allocation des données. En revanche, nous pouvons obtenir un classificateur de politique préférée qui permet d'approcher l'accélération fournie par la meilleure politique. Donc en théorie, si une application est sensible à la localité, on peut choisir une politique convenable, et si elle ne l'est pas, on peut faire un choix sans conséquence (bonne ou mauvaise) sur la durée d'exécution. Cependant, les paramètres proposés sont caractéristiques du matériel et pas seulement des applications, donc on n'atteint pas encore l'objectif idéal, qui est de trouver des caractéristiques propres aux applications qui peuvent être transmises à travers le langage par les développeurs, et qui permettraient de déterminer à l'avance le placement optimal des tâches et des données. Dans le Chapitre 3, nous présentons quelques métriques caractéristiques d'applications issues de leur schémas de communications. Nous projetons de compléter cette étude avec de telles caractéristiques, pour améliorer la précision des modèles proposés et ceci avec des caractéristiques propres aux applications. Cette conclusion mérite toutefois d'être nuancée par rapport à la représentativité et la variété des applications utilisées en entrée des mo-

modèle	pire	prédiction	meilleure	polynôme	SVD
Random Forest	0.87	1.14	1.29	Non	Non
Random Forest	0.87	1.08	1.29	Non	Oui
Random Forest	0.87	1.15	1.29	2	Oui
Random Forest	0.87	1.01	1.29	3	Oui
SVM	0.87	1.00	1.29	Non	Non
SVM	0.87	1.01	1.29	Non	Oui
SVM	0.87	1.11	1.29	2	Oui
SVM	0.87	1.08	1.29	3	Oui
réseau de neurones	0.87	1.13	1.29	Non	Non
réseau de neurones	0.87	1.15	1.29	Non	Oui
réseau de neurones	0.87	1.21	1.29	2	Oui
réseau de neurones	0.87	1.11	1.29	3	Oui

TABLE 6.4 – *Speedup* moyen de l'exécution des applications sensibles à la localité en comparaison avec la politique par défaut. « Modèle » indique le type de classificateur. « Pire » politique indique le *speedup* moyen avec la plus mauvaise politique. « Prédiction » indique le *speedup* moyen avec la politique proposée par le classificateur. « Meilleure » politique indique le *speedup* moyen avec la meilleure politique. « polynôme » indique si on a appliqué une transformation polynomiale, et le degré du polynôme. « SVD » indique si on a appliqué une décomposition en valeurs singulières avant d'entraîner le classificateur.

dèles. En effet, il reste probable que les applications utilisées dans ce chapitre, ne soient pas en mesure d'explorer de manière exhaustive toute les limites du matériel, et donc que les modèles ne puissent pas prendre la bonne décision dans un contexte nouveau.

Chapitre 7

Conclusion et perspectives

7.1 Conclusion

7.1.1 Contexte

La science et l'industrie utilisent le Calcul Haute Performance (CHP) pour résoudre des problèmes nécessitant d'immenses ressources de calcul. Parmi les acteurs qui utilisent le CHP, on trouve l'aéronautique, l'automobile, l'aérospatial, la médecine, l'armée, *etc.* Les besoins en ressources de calcul des applications sont croissantes, et la puissance de calcul des systèmes de CHP augmente de manière exponentielle (*cf.* Figure 1.2). Cependant la croissance en puissance de calcul des plates-formes généralistes va de pair avec leur complexité d'utilisation. En particulier, la hiérarchie mémoire de ces systèmes est extrêmement large et profonde (*cf.* Chapitre 2), et jusqu'ici abstraite aux yeux de l'utilisateur comme un espace d'adressage plat au sein d'un nœud (*cf.* Chapitre 3). Pourtant nous montrons que la performance des accès à la mémoire peut varier de plusieurs ordres de magnitude selon l'utilisation que l'on en fait (*cf.* Chapitre 2). De plus l'écart entre la performance des accès à la mémoire et la performance de calcul brut tend à s'agrandir [105] à grande vitesse. Par conséquent, sur les plates-formes généralistes, la vitesse d'exécution des applications dépendra de la qualité de l'utilisation de la mémoire.

Les technologies contemporaines ne sont pas à même de maintenir l'accroissement global de la performance des plates-formes de calcul [105]. Les technologies émergentes (Mémoire sur la puce du processeur (*In-Package Memory*) (IPM), mémoire attachée dans le réseau, RAM non-volatile (*Non-Volatile Dual Inline Memory Module*) (NVDIMM)) nécessitent actuellement d'exposer une partie de cette complexité à l'utilisateur pour en tirer des performances raisonnables. Or, il est clair que les problèmes d'optimisation des applications sur des machines complexes ne sont pas solubles par les utilisateurs qui conçoivent des applications dans une science déjà compliquée. Alors que les systèmes évoluent les applications sont de plus en plus nombreuses et complexes. Elles nécessi-

teront un coût en développement supplémentaire, croissant avec la complexité de la machine, pour les réadapter, si on ne peut pas obtenir automatiquement de la performance sur les nouveaux systèmes de calcul. Donc, la localité des données, *i.e.* la bonne utilisation de la hiérarchie mémoire, et l'abstraction de la complexité des machines resteront des problèmes critiques à résoudre pour soutenir la course à la performance. Les challenges actuels et à venir dans ce domaine, sont rythmés par les innovations qui rendent le matériel plus efficace, mais aussi plus complexe, et concernent l'identification des goulets d'étranglement et l'exposition du bon niveau d'abstraction des mécanismes d'optimisation à toutes les couches de la pile logicielle.

7.1.2 Contributions

Dans ce manuscrit nous explorons ces axes de recherche à l'échelle de la mémoire partagée, pour des systèmes administrés par un seul système d'exploitation mais incluant possiblement plusieurs processeurs ou un seul processeur embarquant plusieurs grappes de cœurs et plusieurs mémoires. Nous avons analysé les applications et les machines jusque dans leur plus basses couches d'abstraction logicielle pour les caractériser et comprendre ce qui fait une bonne localité et l'exposer de manière synthétique à l'utilisateur final.

Une extension du Cache-Aware Roofline Model (CARM) Nous avons proposé une extension (LARM) d'un modèle (CARM) de machine et d'application pour comprendre et exposer les enjeux de la localité aux utilisateurs (*cf.* Chapitre 4). Ce modèle permet de représenter les applications de manière homogène dans un repère borné par les performances des composants de la machine, et d'évaluer le potentiel des applications à atteindre différents seuils de performance correspondants à des goulets d'étranglement de la machine. De plus, nous avons également construit un modèle de bande passante hybride, caractérisant le débit des accès à la mémoire lorsque plusieurs mémoires sont utilisées simultanément en lecture et en écriture. Ce dernier modèle permet de caractériser plus finement la limite de bande passante de la machine selon des contraintes de l'application, et permet entre autre d'améliorer le LARM avec des informations encore plus pertinentes sur les limites des plates-formes.

Un outil de modélisation de la localité Nous avons proposé un outil pour accélérer le processus de modélisation de la localité en associant des métriques à une représentation structurée de la plate-forme (*cf.* Chapitre 5). En particulier, nous avons utilisé ce modèle pour reconstruire une métrique synthétique de localité basée sur la contention du cache.

Une technique de sélection automatique de politique d'exécution Enfin nous avons proposé une approche statistique pour la sélection automa-

tique d'une stratégie de placement de tâches et de données afin d'améliorer les performances des applications (*cf.* Chapitre 6). En observant plusieurs compteurs matériels lors de l'exécution d'un ensemble d'applications, nous entraînons un détecteur de sensibilité des applications à la localité et un classificateur d'application par politique d'exécution préférée. Cette approche permet de sélectionner automatiquement la politique d'exécution d'une application et d'améliorer le temps d'exécution en moyenne de 21% (contre 29% pour la meilleure politique) par rapport à la politique généralement utilisée par défaut.

7.2 Perspectives

Notre travail nous a permis d'acquérir une connaissance profonde du système (machine, application), des compromis qui s'y opèrent, des moyens de les observer et des moyens de les exposer. À court terme nous projetons de développer certains axes des travaux déjà étudiés, tandis qu'à moyen terme, nous envisageons que ces recherches permettront de développer des solutions pour la pile logicielle qui exploitera le matériel émergent.

7.2.1 À court terme

Modélisation du schéma d'accès à la mémoire Les modèles de performance des accès mémoire permettent d'anticiper le temps d'exécution des applications et le coût du déplacement des données. A fortiori de tels modèles donnent des paramètres clés aux supports d'exécution pour ordonnancer les applications de manière efficace. Dans le Chapitre 4, nous modélisons le débit de données maximum reçu par les cœurs en fonction du placement des données dans la mémoire et du ratio de lecture et d'écriture. Pour maximiser le débit, les données sont parcourues de manière contiguë et continue. Cependant, lorsque ce n'est pas le cas, on aimerait aussi pouvoir prévoir la performance des accès à la mémoire. Une perspective à court terme serait donc de modéliser pour chaque mémoire la performance des accès à cette dernière selon le schéma d'accès aux données. Par exemple, on pourrai modéliser la bande passante mémoire en fonction de la répartition du décalage entre chaque déréférencement d'adresse. Comme on perd la notion d'ordre des accès on peut par exemple ordonner aléatoirement les décalages. Dans un premier temps on observerai la répartition des décalages d'adresses virtuelles entre chaque requête des applications, qui décrivent le schéma d'accès aux données de celle-ci. On modéliserai la loi de répartition des décalage d'adresses, dont les paramètres changent d'une application à l'autre. On pourrait alors explorer les paramètres de la loi et la bande passante qu'ils permettent d'obtenir avec des bancs de test. On obtiendrai finalement une bande passante moyenne réalisable, paramétrée par une loi, c'est-à-dire paramétrée par un modèle de schéma d'accès aux données des

applications, que l'on pourrait faire correspondre à une nouvelle application.

Modélisation de la sensibilité à la localité avec des paramètres d'application Un des buts que nous poursuivons est d'inclure dans le langage de programmation des directives qui permettent de déduire le meilleur placement dynamiquement à l'exécution. Les directives sont des caractéristiques statiques de l'application, qui restent aujourd'hui encore à déterminer. Dans le Chapitre 6, nous construisons un classificateur pour décider ou non si une application est sensible à la localité. Ces classificateurs sont entraînés avec des valeurs issues de compteurs matériels, témoins de la réponse du matériel à l'exécution de l'application. Cependant, afin de construire des directives statiques, il faut pouvoir déduire un placement optimal à partir de caractéristiques statiques des applications. Nous proposons donc d'étendre la liste des paramètres que nous utilisons à des paramètres de l'application décris au Chapitre 3, et dont une partie est actuellement mise en œuvre dans un outil d'instrumentation décris en sous-section **??**.

Modélisation de la sensibilité à la localité avec des exemples de bancs de tests Les classificateurs proposés au Chapitre 6 sont entraînés avec un sous-ensemble des applications et testés avec un nouvel exemple. Cette méthode d'apprentissage avec des applications présente plusieurs inconvénients. D'abord en utilisant une application complète, on moyenne un comportement potentiellement dynamique. Si une application comporte des phases très différentes qui requièrent des politiques d'exécution différentes, l'apprentissage déduit le placement dans un cas très particulier de phases moyennées. Donc il faudrait découper les applications à un grain plus fin pour caractériser les phases individuellement. Ensuite, le nombre d'exemples pour l'apprentissage est extrêmement réduit et probablement peu représentatif de toute les configurations d'utilisation du système de calcul qui peuvent apparaître dans de nouveaux exemples. Par conséquent, une perspective à court terme serait de proposer une large suite de bancs de test, similaire à SpecCPU[51], un peu plus bas niveau, pour stresser différentes parties du matériel et entraîner les classificateurs sur un grand nombre d'exemples représentatifs. Par exemple on pourrait prendre un sous-ensemble des bancs de test décrits en Sections 4.3 et 4.2.2, et ceux proposés plus haut, pour modéliser les accès à la mémoire, selon l'intensité arithmétique des applications, *etc.* Enfin on valide les classificateurs sur les applications. La qualité du classificateur serait même un bon estimateur de la pertinence des *benchmarks* utilisés et des paramètres choisis, pour caractériser la performance des applications sur la plate-forme cible.

7.2.2 À moyen terme

Migration de données en mémoire rapide Certains processeurs contemporains (Knights Landing (KNL), Unité de calcul graphique (*Graphic Processing Unit*) (GPU)) sont équipés de plusieurs types de mémoire avec des caractéristiques différentes. Dans un avenir proche, cette architecture mémoire devrait perdurer. Celle-ci impose de choisir entre différents compromis lorsqu'il s'agit d'allouer ou de migrer des données. Actuellement, on voit s'opposer des solutions logicielles et des solutions matérielles qui satisfont à des compromis qui siéent à différentes applications. Un cas d'utilisation d'une mémoire rapide et de petite taille est le préchargement de données. Nous envisageons que certaines applications pour lesquelles le développeur connaît facilement les données à précharger, une solution logicielle surpassera les solutions de type cache matériel. Cependant, il y a des compromis compliqués à prendre en compte, (*e.g.* est-ce qu'il vaut mieux attendre des données qui ne sont pas complètement pré-chargées ou commencer le calcul en mémoire lente ?) qui doivent être résolus par une analyse fine des temps d'accès à la mémoire.

Politiques d'exécution des applications et paradigmes de programmation Le placement par politique est extrêmement confortable car il ne nécessite pas une mise en œuvre complexe pour coopérer avec les applications. Du point vue de la pile logicielle, tout se passe dans le support d'exécution (et éventuellement le système d'exploitation) et ne nécessite aucune transformation du code de l'application. En comparaison, les solutions qui se basent sur une analyse profonde des applications [35] ou une transformation du matériel [34], sont moins attirantes car elles nécessitent une analyse extrêmement lourde voire irréalisable pour de grosses applications, ou bien des transformations du matériel au bon vouloir des constructeurs, pour des performances parfois comparables aux solutions par politique. En effet, le problème du placement de tâches est un problème multicritères et NP-complet [89, 63]. Donc, il n'est pas exclu que des solutions simples puissent couvrir une vaste majorité des besoins si le paradigme de programmation oriente le développement pour que celles-ci fonctionnent bien. Au Chapitre 6 nous montrons que le choix d'une politique statique pour toute la durée d'une application permet déjà d'améliorer en moyenne le temps d'exécution jusqu'à 29% pour les paradigmes en équipe de threads (*fork-join*). Cette conclusion sur le placement des tâches et des données par politique vaut pour des paradigmes de programmation en tâches [89]. A moyen terme l'identification des facteurs de la localité dans les paradigmes existants (éventuellement munis d'extensions pour la localité) permettrai de définir et de choisir automatiquement des politiques d'exécution génériques.

Gestion de la RAM non-volatile Prochainement les processeurs seront équipés de RAM non-volatile adossée à une RAM volatile classique, offrant une plus grande capacité que cette dernière et des performances à mi-chemin entre celles des SSD sur le bus PCIe et celles de la RAM classique. Selon les usages prévus pour cette mémoire il sera peut-être nécessaire d'adapter les interfaces. Par exemple on peut l'utiliser comme un niveau de cache supplémentaire avant d'accéder au disque dur, ou encore pour sauvegarder le contenu de la RAM en cas de panne. Au même titre que la mémoire sur la puce du processeur, la gestion de ce niveau de mémoire supplémentaire pourrai être automatique. En revanche, il fournirai probablement de meilleures performances si l'utilisateur est capable de transmettre les bonnes informations au support d'exécution pour l'utiliser au bon moment. La recherche du bon niveau d'abstraction à soumettre à l'utilisateur passe par la compréhension de ses besoins et une caractérisation précise des compromis de performance de ce type de matériel.

Annexes

Annexe A

Caractéristiques des plates-formes

Pour les expériences menées dans cette thèse, nous utilisons des nœuds de calcul de la Plateforme Fédérative pour la Recherche en Informatique et Mathématiques (PlaFRIM) [92].

A.1 Xeon Phi(TM) Knights Landing (KNL)

Ce système présente la particularité de posséder une mémoire rapide (MCDRAM de 16Go) sur la puce du processeur et d'une mémoire RAM classique. Les 64 cœurs de ce processeur, cadencés à 1,3GHz, sont groupés par paquet de 2 cœurs partageants une tuile de cache L2, et ces tuiles sont interconnectées à travers un maillage cartésien en deux dimensions. La machine est configurable au démarrage pour être vue logiciellement comme un groupe de 4, 2 ou un seul domaine(s) NUMA (*Sub-NUMA-Cluster* (SNC)), chacun composés d'une mémoire MCDRAM et d'une mémoire RAM. Une représentation, telle que vue par l'application, du premier domaine NUMA de la machine découpée en 4 domaines NUMA est fournie en Figure A.3. La plate-forme peut aussi être configurée pour déléguer au matériel la gestion de la MCDRAM comme un dernier niveau de cache (mode *cache*), ou bien la couper en deux parties (mode *hybrid*), dont une est adressable et l'autre est gérée par le matériel, ou enfin la laisser entièrement disponible comme une mémoire supplémentaire (mode *flat*). Enfin, on peut sélectionner une politique de routage des données sur le maillage, privilégiant la localité au sein des domaines NUMA (mode *quadrant*) ou homogène sur tout le réseau.

Processeur	Xeon Phi(TM) KNL
Figure	A.3
Fréquence (GHz)	1,3
Nombre de cœurs	64
Topologie d'interconnexion des cœurs	maillage 2D
Nombre de *Socket*	1
Nombre de domaines NUMA	4
Technologie d'interconnexion des *sockets*	N/A
Mode *Cluster*	SNC4, flat, quadrant
Hyper-Threading	Oui
Turbo Boost	Oui

FIGURE A.1 – Caractéristiques et configuration du système Xeon Phi(TM) KNL

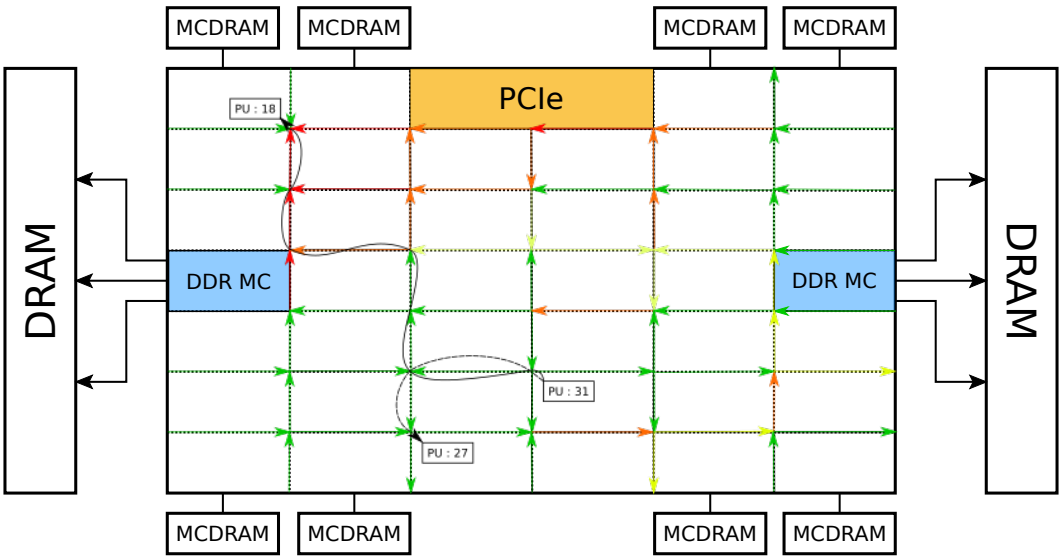


FIGURE A.2 – Schéma d'un routage de données hypothétique sur le réseau d'interconnexion d'un processeur KNL. Les flèches vertes indiquent des liens non-saturés, les flèches jaunes indiquent des liens partiellement saturés, et les flèches rouges indiques des liens saturés.

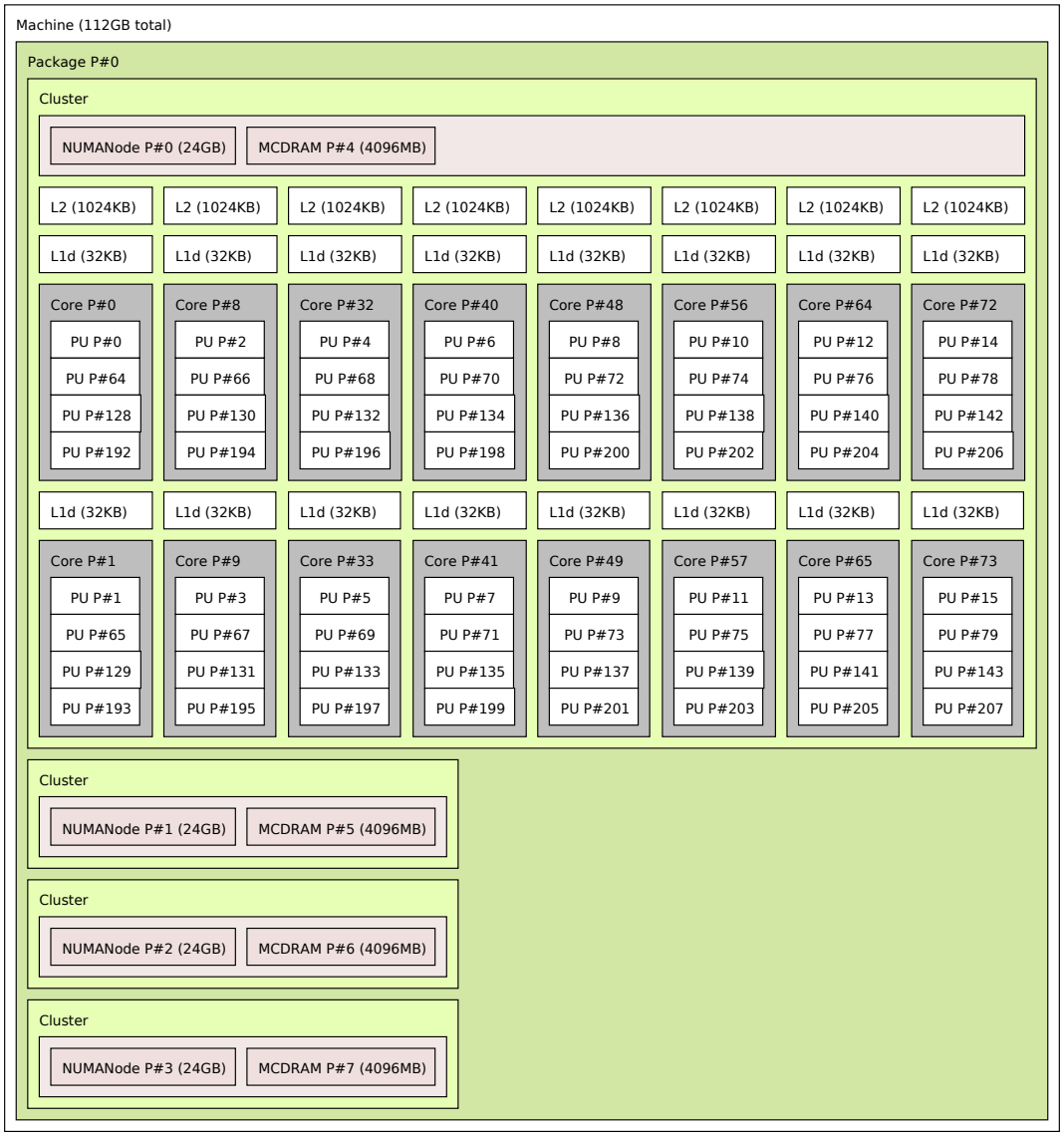


FIGURE A.3 – Topologie du premier domaine NUMA accompagné des mémoires des autres domaines du système Xeon Phi(TM) KNL.

Processeur	Haswell Xeon E5-2680 v3
Figure	A.5
Fréquence (GHz)	2,5
Nombre de cœurs	24
Topologie d'interconnexion des cœurs	anneau
Nombre de *Socket*	2
Nombre de domaines NUMA	4
Technologie d'interconnexion des *sockets*	QPI
Mode *Cluster*	*Cluster-on-Die*
Hyper-Threading	Non
Turbo Boost	Oui

FIGURE A.4 – Caractéristiques et configuration du système Haswell Xeon E5-2680 v3 *bi-socket*

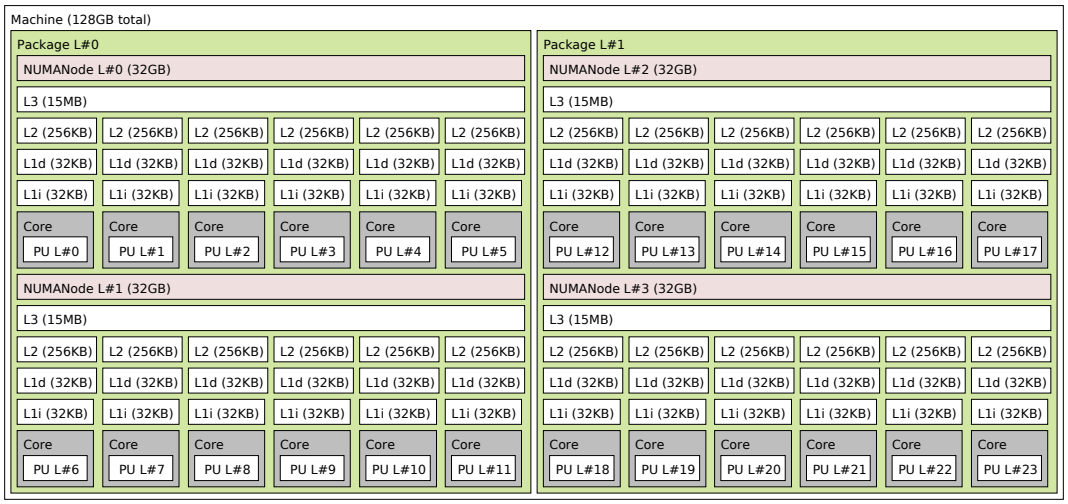


FIGURE A.5 – Topologie du système Haswell Xeon E5-2680 v3 *bi-socket*

A.2 Haswell Xeon E5-2680 v3 bi-*socket*

Processeur	Broadwell Xeon E5-2650L v4
Figure	A.7
Fréquence (GHz)	1,7
Nombre de cœurs	28
Topologie d'interconnexion des cœurs	anneau
Nombre de *Socket*	2
Nombre de domaines NUMA	4
Technologie d'interconnexion des *sockets*	QPI
Mode *Cluster*	*Cluster-on-Die*
Hyper-Threading	Non
Turbo Boost	Non

FIGURE A.6 – Caractéristiques et configuration du système Broadwell Xeon E5-2650L v4 *bi-socket*

A.3 Broadwell Xeon E5-2650L v4 bi-*socket*

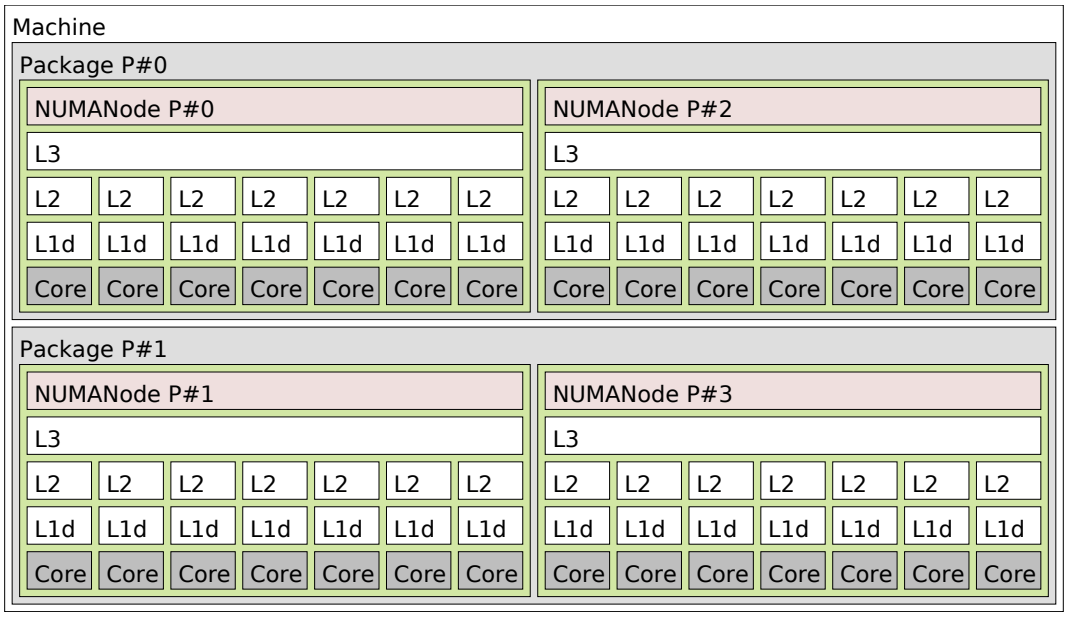


FIGURE A.7 – Topologie du système Broadwell Xeon E5-2650L v4 *bi-socket*

A.4 Skylake Intel(R) Xeon(R) Gold 6140 bi-*socket*

Processeur	Skylake Intel(R) Xeon(R) Gold 6140
Figure	A.9
Fréquence (GHz)	2,3
Nombre de cœurs	36
Topologie d'interconnexion des cœurs	maillage 2D
Nombre de *Socket*	2
Nombre de domaines NUMA	4
Technologie d'interconnexion des *sockets*	UPI
Mode *Cluster*	*Sub-NUMA Cluster*-2 (SNC2)
Hyper-Threading	Non
Turbo Boost	Non

FIGURE A.8 – Caractéristiques et configuration du système Skylake Intel(R) Xeon(R) Gold 6140 *bi-socket*

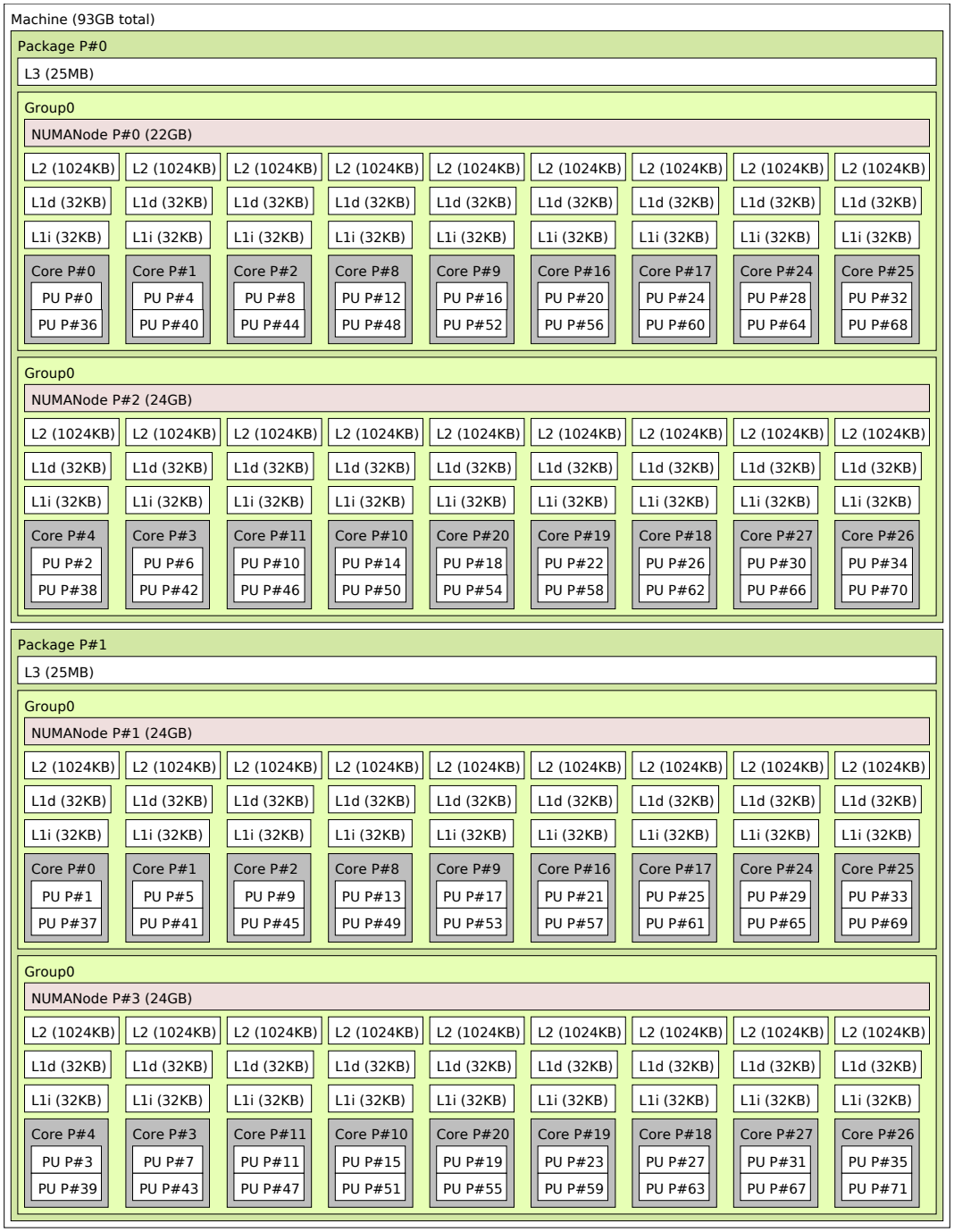


FIGURE A.9 – Topologie du système Skylake Intel(R) Xeon(R) Gold 6140 *bi-socket*

Annexe B

Applications utilisées pour le placement de threads et de données (Chapitre 6)

B.1 Coral *benchmarks*

La suite d'applications Coral [19] est une initiative du *Department of Energy* (DoE), destinée à comparer la performance des systèmes de calcul dans la résolution de problèmes issus de laboratoires américains (Oak Ridge, Argonne et Livermore), dans le but de répondre aux appels d'offres du DoE pour la commande systèmes de calcul haute performance. Nous utilisons les applications issues de cette suite écrites en C et utilisants le paradigme OpenMP [23], résumées en Table B.1 et décrites dans la liste ci-après.

- lcals [55] : est une suite de noyaux de calculs représentatifs d'une partie des problèmes traités par le DoE. Ces noyaux sont intensifs dans leur utilisation de la mémoire. Parmi les noyaux disponibles nous sélectionnons uniquement les noyaux parallèles, *i.e.* fir, del_dot_vec_2d, energy_calc_alt, vol3d, couple, pressure_calc_alt et pic_2d. Pour chaque mesure, les éléments à exécuter sont activés puis désactivés dans le fichier `main.cxx` avant de recompiler l'application. Les mesures sont prises exactement entre les appels aux fonctions `TIMER_START` et `TIMER_STOP` de l'application.
- lulesh2.0 [64] : est une application d'hydrodynamique lagrangienne tridimensionnelle. Dans le Chapitre 6 lulesh est exécutée avec les paramètres `-b 4 -s 100 -i 40 -r 100`. Les points de mesure de lulesh sont pris dans la fonction `main` du fichier `lulesh.cc`, exactement entre les fonctions de mesure de temps `gettimeofday` autour de la boucle d'appel à la fonction `LagrangeLeapFrog(*locDom)`.

– hpccg [52] : est un acrnonyme pour *High Performance Conjugate Gradient*, qui est un solveur de gradient conjugué pour matrices pré-conditionnées, et est présenté comme pouvant être utilisé pour vérifier le bon fonctionnement des plates-formes de CHP. La mesure des caractéristiques de l'application est prise autour de l'appel à `HPCCG( A, b, x, max_iter, tolerance, niters, normr, times)` dans le fichier `main.cpp`. L'application est exécutée avec les paramètres `200 200 200`.
– MILCmk [81] : est une suite de noyaux utilisés dans des codes de physique nucléaire. L'application propose deux bancs de tests, un pour les performances en calcul double précision (qla_bench-qla-1.7.1-d3) et une pour les performances en calcul simple précision (qla_bench-qla-1.7.1-f3). Nous n'utilisons que la première, en limitant les exécutions dans le code à une seule taille d'ensemble de travail par thread de `nmax = 256*1024*16`, dans le fichier `qla_bench.c`. Dans ce même fichier nous prenons les mesures autour de l'inclusion `#include "benchfuncs.c"` du code principal de l'application.
– HACCmk : Est un code de cosmologie composé d'une boucle découpée en une phase de préparation des données puis une phase de calcul. Dans le fichier `main.c` de ce code nous prenons nos mesures autour de la boucle principale `for ( n = 400; n < N; n = n + 20 )`.

B.2 Simulation Numérique Aéronautique (*Numerical Aerodynamic Simulation*) (NAS) *benchmarks*

La suite d'applications NAS [5] est destinée à évaluer les performances des supercalculateurs. Cette suite est composée d'un ensemble d'applications parallèles codées C ou en Fortran et parallélisées avec le paradigme OpenMP [1]. Pour cette suite comme pour les applications Coral, nous insérons des sondes directement autour de la région d'intérêt principale identifiée dans les application exécutées avec les classes [2] A et B, avec une version implémentée en langage C [3] pour toutes les applications. Ces applications représentent des sous-problèmes présents dans les codes d'aéronautique et aérospatial :

– lu : est un code factorisation LU pour résoudre des systèmes d'équations. Pour cette application, les sondes sont placées dans le fichier `lu.c` autour de l'ensemble de fonctions `ssor(); error(); pintgr();`.

1. Les NAS existent également en MPI.
2. La classe d'une application NAS est déterminée par l'utilisateur à la compilation et agit sur la taille du problème traité, indépendemment des caractéristique du système.
3. https ://github.com/benchmark-subsetting/NPB3.0-omp-C

- cg : est une application de calcul de gradiant conjugué. Dans cette application, les sondes sont placées dans le fichier `cg.c` autour de la boucle qui appel la fonction `conj_grad(colidx, rowstr, x, z, a, p, q, r/*, w*/, &rnorm)`.
- ep : est une application caractérisée par un haut niveau de parallélisme (*embarassingly parallel*), *i.e.* elle passe facilement à l'échelle sur des systèmes très larges. Les sondes pour cette application sont placées dans le fichier éponyme de l'application entre les appels aux fonction de mesure de temps `timer_start(1)` et `timer_stop(1)`.
- mg : est un parcours de maillage multi-grille. Dans le code de mg, nous plaçons des sondes entre les appels aux fonctions de mesure de temps `timer_start(T_BENCH)` et `timer_stop(T_BENCH)`.
- sp : est un solveur scalaire penta-diagonal. Dans le code de sp, nous plaçons des sondes autour de la boucle d'appel à la fonction `adi()`.
- bt : est un solveur par bloc tri-diagonal. Dans le code de bt, nous plaçons des sondes entre les appels aux fonctions de mesure de temps `timer_start(1)` et `timer_stop(1)`.
- ft : est une transformée de fourrier discrète. Dans le code de ft, nous plaçons des sondes autour de la boucle qui précède la vérification des résultats, et dans laquelle les fonctions `evolve()`, `fft()` et `checksum()` sont appelées.

B.3 *Princeton Application Repository for Shared-Memory Computers* (PARSEC) *benchmarks*

PARSEC est un ensemble d'applications représentatif des applications émergentes dans le domaine du calcul intensif. Ces applications ont la particularité d'être conçues pour être exécutées en mémoire partagée uniquement. Les applications sont livrées avec des outils de compilation et une bibliothèque d'instrumentation de la région d'intérêt de chacune d'entre elles. Donc nous mesurons les métriques de performance à partir des fonctions marquants le début et la fin de la *Region of Interest* dans la bibliothèque embarquée. Lorsqu'il est possible, nous compilons les applications avec la configuration `gcc-openmp` sinon, avec la configuration `gcc-pthreads`. Une description rapide du travail effectuée par ces applications est donnée dans le tableau B.1.

application	paramètres	suite	description
ferret	input native	Parsec	Recherche de similarités
swaptions	input native	Parsec	Évaluation de cote boursière
freqmine	input native	Parsec	Exploration de données
canneal	input native	Parsec	Routage dans les semi-conducteurs
bodytrack	input native	Parsec	Tracking visuel de personnes
dedup	input native	Parsec	Compression de données
fir	NA	Coral (lcals)	Collection de noyaux...
del_dot_vec_2d	NA	Coral (lcals)	en boucle, représentative...
energy_calc_alt	NA	Coral (lcals)	des applications exécutées...
vol3d	NA	Coral (lcals)	au Department of Energy
couple	NA	Coral (lcals)	
pressure_calc_alt	NA	Coral (lcals)	
pic_2d	NA	Coral (lcals)	
lulesh2.0	-b 4 -s 100 -i 40 -r 100	Coral	Stencil, shocks hydroliques
hpccg	200 200 200	Coral	Gradiant Conjugué
MILKmk	nmax = 256*1024*16	Coral	Physique nucléaire (double précision)
HACCmk	count=200	Coral	Cosmologie (n-cores)
lu	classe A classe B	NAS	Factorisation LU
cg	classe A classe B	NAS	Gradiant conjugué
ep	classe A classe B	NAS	Extrêmement parallèle, ... peu de communications
mg	classe A classe B	NAS	Parcours de maillage ... multi-grille
sp	classe A classe B	NAS	Solveur scalaire ... penta-diagonal
bt	classe A classe B	NAS	Solveur de bloc ... tri-diagonal
ft	classe A classe B	NAS	Transformée de... Fourrier discrète

TABLE B.1 – Tableau récapitulatif des applications utilisées pour caractériser le passage à l'échelle, le placement de tâches et le placement de données.

Bibliographie

Références bibliographiques

[1] Intel 3d-xpoint non-volatile memory. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html, 2018. *Accédé le : 2018/06/15.*

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams et Katherine A. Yelick : The landscape of parallel computing research : A view from berkeley. Rapport technique, TECHNICAL REPORT, UC BERKELEY, 2006.

[3] InfiniBand Trade Association *et al.* : The infiniband architecture specification. *http ://www. infinibandta. org/specs/*, 2000.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst et Pierre-André Wacrenier : Starpu : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2):187–198, 2011.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan et S. K. Weeratunga : The nas parallel benchmarks. Rapport technique, The International Journal of Supercomputer Applications, 1991.

[6] Iñigo Barandiaran : The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8), 1998.

[7] N. Beckmann et D. Sanchez : Modeling cache performance beyond lru. *In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, March 2016.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh et Kai Li : The parsec benchmark suite : Characterization and architectural implications. *In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[9] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood et Robert C Zak : Intel® omni-path architecture : Enabling scalable, high performance fabrics. *In High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 1–9. IEEE, 2015.

[10] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova et Ali Kamali : A case for numa-aware contention management on multicore systems. *In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM.

[11] Bernhard E Boser, Isabelle M Guyon et Vladimir N Vapnik : A training algorithm for optimal margin classifiers. *In Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault et Jack J Dongarra : Parsec : Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[13] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault et Raymond Namyst : hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications. *In* IEEE, éditeur : *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, février 2010.

[14] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier et Raymond Namyst : ForestGOMP : an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming*, 2010.

[15] P. Caheny, L. Alvarez, S. Derradji, M. Valero, M. Moretó et M. Casas : Reducing cache coherence traffic with a numa-aware runtime approach. *IEEE Transactions on Parallel and Distributed Systems*, 29(5): 1174–1187, May 2018.

[16] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra et J. F. Méhaut : A machine learning-based approach for thread mapping on transactional memory applications. *In 2011 18th International Conference on High Performance Computing*, pages 1–10, Dec 2011.

[17] Georgios Chatzopoulos, Rachid Guerraoui, Tim Harris et Vasileios Trigonakis : Abstracting multi-core topologies with mctop. *In Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 544–559, New York, NY, USA, 2017. ACM.

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

[19] The coral benchmarks codes. https://asc.llnl.gov/CORAL-benchmarks/, 2018. *Accédé le : 2018/04/16*.

[20] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat et Parkson Wong : Overview of the mpi-io parallel i/o interface. *In Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[21] Eduardo H.M. Cruz, Matthias Diener, Marco A.Z. Alves et Philippe O.A. Navaux : Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. *Journal of Parallel and Distributed Computing*, 74(3):2215 – 2228, 2014.

[22] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla et Philippe OA Navaux : An efficient algorithm for communication-based task mapping. *In Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 207–214. IEEE, 2015.

[23] Leonardo Dagum et Ramesh Menon : Openmp : an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[24] *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*. IEEE Computer Society, 2016.

[25] Arnaldo Carvalho De Melo : The new linux'perf'tools. *In Slides from Linux Kongress*, volume 18, 2010.

[26] Nicolas DENOYELLE : Moniteurs hiérarchiques de performance, pour gérer l'utilisation des ressources partagées de la topologie. *In Compas*, Lorient, France, juillet 2016.

[27] Nicolas DENOYELLE, Brice GOGLIN, Aleksandar ILIC, Emmanuel JEANNOT et Leonel SOUSA : Modeling large compute nodes with heterogeneous memories with cache-aware roofline model. *In* Stephen JARVIS, Steven WRIGHT et Simon HAMMOND, éditeurs : *High Performance Computing systems - Performance Modeling, Benchmarking, and Simulation - 8th International Workshop, PMBS 2017*, volume 10724 de *Lecture Notes in Computer Science*, pages 91–113, Denver (CO), United States, novembre 2017. Springer.

[28] Nicolas DENOYELLE, Brice GOGLIN et Emmanuel JEANNOT : A Topology-Aware Performance Monitoring Tool for Shared Resource Management in Multicore Systems. *In* SPRINGER, éditeur : *Proceedings of Euro-Par 2015 : Parallel Processing Workshops*, Lecture Notes in Computer Science, Vienna, Austria, août 2015.

[29] Nicolas DENOYELLE, Brice GOGLIN et Emmanuel JEANNOT : Modeling Non-Uniform Memory Access on Large Compute Nodes with the Cache-Aware Roofline Model. *IEEE Trans. Parallel Distrib. Syst.*, 19, 2019. **Soumission en cours**.

[30] Nicolas DENOYELLE, Aleksandar ILIC, Brice GOGLIN, Leonel SOUSA et Emmanuel JEANNOT : Automatic Cache Aware Roofline Model Building and Validation Using Topology Detection. *In NESUS Third Action Workshop and Sixth Management Committee Meeting*, volume I, Sofia, Bulgaria, octobre 2016. Jesus Carretero.

[31] Saïd DERRADJI, Thibaut PALFER-SOLLIER, Jean-Pierre PANZIERA, Axel POUDES et François Wellenreiter ATOS : The bxi interconnect architecture. *In High-Performance Interconnects (HOTI), 2015 IEEE 23rd Annual Symposium on*, pages 18–25. IEEE, 2015.

[32] Tanima DEY, Wei WANG, Jack W. DAVIDSON et Mary Lou SOFFA : Characterizing multi-threaded applications based on shared-resource contention. *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 76–86, Washington, DC, USA, 2011. IEEE Computer Society.

[33] M. DIENER, E. H. M. CRUZ et P. O. A. NAVAUX : Locality vs. balance : Exploring data mapping policies on numa systems. *In 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 9–16, March 2015.

[34] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse et Hans-Ulrich Heiíss : Communication-aware process and thread mapping using online communication detection. *Parallel Comput.*, 43(C): 43–63, mars 2015.

[35] Matthias Diener, Eduardo H.M. Cruz, Laércio L. Pilla, Fabrice Dupros et Philippe O.A. Navaux : Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation*, 88-89:18 – 36, 2015.

[36] Chen Ding et Yutao Zhong : Predicting whole-program locality through reuse distance analysis. *In Acm Sigplan Notices*, volume 38, pages 245–257. ACM, 2003.

[37] Jack J Dongarra, Piotr Luszczek et Antoine Petitet : The linpack benchmark : past, present and future. *Concurrency and Computation : practice and experience*, 15(9):803–820, 2003.

[38] Jack J Dongarra, Hans W Meuer, Erich Strohmaier *et al.* : Top500 supercomputer sites, 1994.

[39] Ulrich Drepper : What every programmer should know about memory, 2007.

[40] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik et Parkson Wong : Theory and practice in parallel job scheduling. *In Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer, 1997.

[41] Christine Fricker, Philippe Robert et James Roberts : A versatile and accurate approximation for lru cache performance. *In Proceedings of the 24th International Teletraffic Congress*, page 8. International Teletraffic Congress, 2012.

[42] Sabela Ramos Garea et Torsten Hoefler : Modelling communications in cache coherent systems. *Technical Report*, 2013.

[43] Fabien Gaud, Baptiste Lepers, Justin R. Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize et Mark Roth : Challenges of memory management on modern NUMA systems. *Commun. ACM*, 58(12):59–66, 2015.

[44] J. Gaur, M. Chaudhuri, P. Ramachandran et S. Subramoney : Near-optimal access partitioning for memory hierarchies with multiple heterogeneous bandwidth sources. *In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, Feb 2017.

[45] Aurélien Géron : *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017.

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

[47] Brice Goglin : Towards the structural modeling of the topology of next-generation heterogeneous cluster nodes with hwloc. Research report, Inria, novembre 2016.

[48] William D Gropp, William Gropp, Ewing Lusk et Anthony Skjellum : *Using MPI : portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[49] Daniel Hackenberg, Daniel Molka et Wolfgang E. Nagel : Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. *In Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, New York, NY, USA, 2009. ACM.

[50] Tomas Hansson, Chris Oostenbrink et WilfredF van Gunsteren : Molecular dynamics simulations. *Current opinion in structural biology*, 12(2):190–196, 2002.

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

[52] Mike Heroux : Hpccg microapp, 2007.

[53] M. D. Hill et M. R. Marty : Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[54] Tin Kam Ho : Random decision forests. *In Document analysis and recognition, 1995., proceedings of the third international conference on*, volume 1, pages 278–282. IEEE, 1995.

[55] RD Hornung : Livermore compiler analysis loop suite. Rapport technique, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[56] Classement hpcg500. http://www.hpcg-benchmark.org/index.html, 2018. *Accédé le : 2018/06/05*.

[57] Aleksandar Ilic, Frederico Pratas et Leonel Sousa : Cache-aware Roofline model : Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014.

[58] Chapitre 9 : Evènement matériels de performance des processeurs intel. https://software.intel.com/sites/default/files/managed/7c/f1/253669-sdm-vol-3b.pdf, 2018. *Accédé le : 2018/08/13.*

[59] Intel 2017 annual report. https://s21.q4cdn.com/600692695/files/doc_financials/2017/annual/Intel_Annual_Report_Final-3.20.pdf, 2017. *Accédé le : 2018/05/25.*

[60] Intel roofline model, inclus dans le logiciel intel advisor. https://software.intel.com/en-us/articles/intel-advisor-roofline, 2018. *Accédé le : 2018/07/19.*

[61] Sverre Jarp, Ryszard Jurga et Andrzej Nowak : Perfmon2 : a leap forward in performance monitoring. *In Journal of Physics : Conference Series*, volume 119, page 042017. IOP Publishing, 2008.

[62] Emmanuel Jeannot, Esteban Meneses, Guillaume Mercier, François Tessier et Gengbin Zheng : Communication and topology-aware load balancing in charm++ with treematch. *In Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[63] Emmanuel Jeannot et Guillaume Mercier : Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures. *In* Pasqua D'Ambra, Mario Rosario Guarracino et Domenico Talia, éditeurs : *Europar*, volume 6272 de *Lecture Notes on Computer Science*, pages 199–210, Ischia, Italy, août 2010. Springer.

[64] Ian Karlin, Jeff Keasler et JR Neely : Lulesh 2.0 updates and changes. Rapport technique, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[65] Howard C Kirsch : Method and circuit for reducing dram refresh power by reducing access transistor sub threshold leakage, may 2005. US Patent 6,888,769.

[66] Laszlo B Kish : End of moore's law : thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149, 2002.

[67] Andi Kleen : A NUMA API for LINUX. *Novel Inc*, 2005.

[68] Alvin R Lebeck, Xiaobo Fan, Heng Zeng et Carla Ellis : Power aware page allocation. *ACM Sigplan Notices*, 35(11):105–116, 2000.

[69] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho et Chong Sang Kim : Lrfu : A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 12:1352–1361, 2001.

[70] Charles E Leiserson : Fat-trees : universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10): 892–901, 1985.

[71] Baptiste Lepers, Vivien Quéma et Alexandra Fedorova : Thread and memory placement on numa systems : Asymmetry matters. *In USENIX Annual Technical Conference*, pages 277–289, 2015.

[72] Marcel Lesieur : *Turbulence in fluids : stochastic and numerical modelling*. Nijhoff Boston, MA, 1987.

[73] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall et Leonid Oliker : Roofline model toolkit : A practical tool for architectural and program analysis. *In* Stephen A. Jarvis, Steven A. Wright et Simon D. Hammond, éditeurs : *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 129–148, Cham, 2015. Springer International Publishing.

[74] Oscar G Lorenzo, Tomás F Pena, José C Cabaleiro, Juan C Pichel et Francisco F Rivera : Using an extended Roofline Model to understand data and thread affinities on NUMA systems. *Annals of Multicore and GPU Programming*, 1(1):56–67, 2014.

[75] C. A. Mack : Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, May 2011.

[76] Zoltan Majo et Thomas R. Gross : Memory management in numa multicore systems : Trapped between cache contention and interconnect overhead. *SIGPLAN Not.*, 46(11):11–20, juin 2011.

[77] Site web de maqao mentionnant l'outil de profilage d'application basé sur des compteurs matériels. http://www.maqao.org/, 2018. *Accédé le : 2018/08/13*.

[78] Outil de mesure de performance des accès mémoire. https://github.com/bputigny/mbench, 2013. *Accédé le : 2018/06/25*.

[79] John D. McCalpin : Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, décembre 1995.

[80] The memkind library. http://memkind.github.io/memkind, 2018. *Accédé le : 2018/07/04.*

[81] Documents de description de l'application milcmk. https://asc.llnl.gov/CORAL-benchmarks/Summaries/MILCmk_Summary_v1.0.pdf, 2018. *Accédé le : 2018/08/10.*

[82] S. Mittal et J.S. Vetter : A survey of software techniques for using non-volatile memories for storage and main memory systems. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1–1, 2015.

[83] Philip J Mucci, Shirley Browne, Christine Deane et George Ho : Papi : A portable interface to hardware performance counters. *In Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[84] Richard C Murphy, Kyle B Wheeler, Brian W Barrett et James A Ang : Introducing the graph 500. *Cray User's Group (CUG)*, 19:45–74, 2010.

[85] Chitra Natarajan, Bruce Christenson et Fayé Briggs : A study of performance impact of memory controller features in multi-processor server environment. *In Proceedings of the 3rd Workshop on Memory Performance Issues : In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, pages 80–87, New York, NY, USA, 2004. ACM.

[86] Andrew NG : Coursera machine-learning mooc. https://fr.coursera.org/learn/machine-learning, 2018. *Accédé le : 2018/07/31.*

[87] Jaroslaw Nieplocha, Robert J Harrison et Richard J Littlefield : Global arrays : A portable" shared-memory" programming model for distributed memory computers. *In Supercomputing'94., Proceedings*, pages 340–349. IEEE, 1994.

[88] François Pellegrini : Scotch and PT-Scotch Graph Partitioning Software : An Overview. *In* Olaf Schenk Uwe Naumann, éditeur : *Combinatorial Scientific Computing*, pages 373–406. Chapman and Hall/CRC, 2012.

[89] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra et P. Beckman : Exploring data migration for future deep-memory many-core systems. *In 2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 289–297, Sept 2016.

[90] Aleksey Pesterev, Nickolai Zeldovich et Robert T. Morris : Locating cache performance bottlenecks using data profiling. *In Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 335–348, New York, NY, USA, 2010. ACM.

[91] Utilisation de la simulation et des machines parallèles pour des applications pharmaceutiques. https://str.llnl.gov/july-2014/lightstone, 2018. *Accédé le : 2018/06/19*.

[92] Page web de la plateforme fédérative pour la recherche en informatique et mathématiques (plafrim). https://www.plafrim.fr, 2018. *Accédé le : 2018/08/10*.

[93] Divulgation des regitres spécifique pour désactiver les préchargeurs de données. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, 2018. *Accédé le : 2018/07/04*.

[94] Bertrand Putigny, Brice Goglin et Denis Barthou : A Benchmark-based Performance Model for Memory-bound HPC Applications. *In International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italie, juillet 2014. IEEE.

[95] M. K. Qureshi et Y. N. Patt : Utility-based cache partitioning : A low-overhead, high-performance, runtime mechanism to partition shared caches. *In 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432, Dec 2006.

[96] Siegfried Raasch et Michael Schröter : Palm–a large-eddy simulation model performing on massively parallel computers. *Meteorologische Zeitschrift*, 10(5):363–372, 2001.

[97] S. Ramos et T. Hoefler : Capability models for manycore memory systems : A case-study with xeon phi knl. *In 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 297–306, May 2017.

[98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson et John D. Owens : Memory access scheduling. *SIGARCH Comput. Archit. News*, 28(2):128–138, mai 2000.

[99] Erven Rohou : Tiptop : Hardware Performance Counters for the Masses. *In 41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 404–413, Pittsburgh, PA, United States, septembre 2012.

[100] Erven ROHOU et David GUYON : Sequential performance : Raising awareness of the gory details. *Procedia Computer Science*, 51:1393–1402, 12 2015.

[101] Robert SCHREIBER et Jack J DONGARRA : Automatic blocking of nested loops. Rapport technique, University of Tennessee Knoxville, TN, USA, 1990.

[102] Derek L SCHUFF, Benjamin S PARSONS et Vijay S PAI : Multicore-aware reuse distance analysis. *In Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[103] Philip SCHWAN *et al.* : Lustre : Building a file system for 1000-node clusters. *In Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[104] H. SERVAT, A. J. PEÑA, G. LLORT, E. MERCADAL, H. C. HOPPE et J. LABARTA : Automating the application data placement in hybrid memory systems. *In 2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 126–136, Sept 2017.

[105] John SHALF, Sudip DOSANJH et John MORRISON : Exascale computing technology challenges. *In Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag.

[106] Hongzhang SHAN, Katie ANTYPAS et John SHALF : Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. *In Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 42. IEEE Press, 2008.

[107] John SHORE et Rodney JOHNSON : Axiomatic derivation of the principle of maximum entropy and the principle of minimum cross-entropy. *IEEE Transactions on information theory*, 26(1):26–37, 1980.

[108] Réponse à la question : Pourquoi doit-on mélanger les échantillons en entrée d'un algorithme de machine learning ? https://stats.stackexchange.com/questions/245502/shuffling-data-in-the-mini-batch-training-of-neural-network/311318#311318, 2018. *Accédé le : 2018/08/2.*

[109] Santhosh SRINATH, Onur MUTLU, Hyesoon KIM et Yale N PATT : Feedback directed prefetching : Improving the performance and bandwidth-efficiency of hardware prefetchers. *In High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 63–74. IEEE, 2007.

[110] Tarek M Taha et Scott Wills : An instruction throughput model of superscalar processors. *IEEE Transactions on Computers*, 57(3):389–403, 2008.

[111] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot et F. Isaila : Topology-aware data aggregation for intensive i/o on large-scale supercomputers. *In 2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 73–81, Nov 2016.

[112] Vivek Tiwari, Sharad Malik et Andrew Wolfe : Power analysis of embedded software : a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[113] Classement top500. https://www.top500.org/lists/, 2018. *Accédé le : 2018/06/13.*

[114] Classement top500 de juin 2018. https://www.top500.org/list/2018/06/, 2018. *Accédé le : 2018/09/04.*

[115] Georgios Tournavitis, Zheng Wang, Björn Franke et Michael F.P. O'Boyle : Towards a holistic approach to auto-parallelization : Integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.*, 44(6):177–187, juin 2009.

[116] Masaki Unno, Shuichi Aono et Hideki Asai : Gpu-based massively parallel 3-d hie-fdtd method for high-speed electromagnetic field simulation. *IEEE Transactions on Electromagnetic Compatibility*, 54(4):912–921, 2012.

[117] Oreste Villa, Daniel R Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero *et al.* : Scaling the power wall : a path to exascale. *In High Performance Computing, Networking, Storage and Analysis, SC14 : International Conference for*, pages 830–841. IEEE, 2014.

[118] John Von Neumann : First draft of a report on the edvac. *IEEE Annals of the History of Computing*, (4):27–75, 1993.

[119] Zheng Wang et Michael FP O'Boyle : Mapping parallelism to multicores : a machine learning based approach. *In ACM Sigplan notices*, volume 44, pages 75–84. ACM, 2009.

[120] Douglas Brent West *et al.* : *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[121] Samuel Williams, Andrew Waterman et David Patterson : Roofline : An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, avril 2009.

[122] Jeffrey R. Wilson et Kent A. Lorenz : *Short History of the Logistic Regression Model*, pages 17–23. Springer International Publishing, Cham, 2015.

[123] Wm. A. Wulf et Sally A. McKee : Hitting the memory wall : Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, mars 1995.

[124] Sergey Zhuravlev, Sergey Blagodurov et Alexandra Fedorova : Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, mars 2010.

Publications

[125] Nicolas Denoyelle : Moniteurs hiérarchiques de performance, pour gérer l'utilisation des ressources partagées de la topologie. *In Compas*, Lorient, France, juillet 2016.

[126] Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Emmanuel Jeannot et Leonel Sousa : Modeling large compute nodes with heterogeneous memories with cache-aware roofline model. *In* Stephen Jarvis, Steven Wright et Simon Hammond, éditeurs : *High Performance Computing systems - Performance Modeling, Benchmarking, and Simulation - 8th International Workshop, PMBS 2017*, volume 10724 de *Lecture Notes in Computer Science*, pages 91–113, Denver (CO), United States, novembre 2017. Springer.

[127] Nicolas Denoyelle, Brice Goglin et Emmanuel Jeannot : A Topology-Aware Performance Monitoring Tool for Shared Resource Management in Multicore Systems. *In* Springer, éditeur : *Proceedings of Euro-Par 2015 : Parallel Processing Workshops*, Lecture Notes in Computer Science, Vienna, Austria, août 2015.

[128] Nicolas Denoyelle, Brice Goglin et Emmanuel Jeannot : Modeling Non-Uniform Memory Access on Large Compute Nodes with the Cache-Aware Roofline Model. *IEEE Trans. Parallel Distrib. Syst.*, 19, 2019. **Soumission en cours**.

[129] Nicolas Denoyelle, Aleksandar Ilic, Brice Goglin, Leonel Sousa et Emmanuel Jeannot : Automatic Cache Aware Roofline Model Building and Validation Using Topology Detection. *In NESUS Third Action*

Workshop and Sixth Management Committee Meeting, volume I, Sofia, Bulgaria, octobre 2016. Jesus Carretero.