Exposer les caractéristiques des architectures à mémoires hétérogènes aux applications parallèles

Andrès Rubio Proaño

Compas'2020 : Parallélisme / Architecture/ Système/ Temps Réel
MILC - Lyon, France, du 30 juin au 3 juillet 2020

Exposer les caractéristiques des architectures à mémoires hétérogènes aux applications parallèles

Andrès Rubio Proaño*

Inria, LaBRI, Univ. Bordeaux,
200 avenue de la Vieille Tour
33405 Talence - France
andres.rubio@inria.fr

Résumé
La complexité des systèmes de mémoire a considérablement augmenté au cours de la dernière décennie. En conséquence, les supercalculateurs incluent des mémoires à plusieurs niveaux, hétérogènes et non uniformes, avec propriétés significativement différentes. Les développeurs d'applications scientifiques sont confrontés à un énorme défi : exploiter efficacement le système de mémoire pour améliorer les performances et la productivité.
Dans ce travail, nous présentons une interface pour gérer la complexité du système de mémoire, composée d'un ensemble d'attributs des mémoires et d'une API pour exprimer et gérer ces diverses caractéristiques à l'aide de métriques, par exemple la bande passante, la latence et la capacité. Elle permet aux supports exécutifs, aux bibliothèques parallèles et aux applications scientifiques de sélectionner la mémoire appropriée en exprimant leurs besoins pour chaque allocation sans avoir à modifier le code pour chaque plate-forme.

Mots-clés : Mémoire hétérogène, mémoire multi-niveaux, NUMA, hwloc.

1. Introduction

Les architectures des supercalculateurs sont devenues de plus en plus complexes au cours des dernières années pour pouvoir supporter les besoins de calcul en augmentation. Parmi les technologies récentes, après les multicœurs et les architectures hétérogènes, on trouve des systèmes de mémoire à plusieurs niveaux et des mémoires hétérogènes.
Il y a quelques années, l'architecture *Intel Knights Landing* (KNL) [8] pouvait être configurée en huit nœuds NUMA avec un système mémoire à deux niveaux : une mémoire à haute bande passante (HBM [2], *High-Bandwidth Memory*) et une mémoire plus lente mais à plus grande capacité (DDR). En raison de leurs caractéristiques très différentes, le choix entre ces mémoires devient essentiel pour les développeurs, puisque la performance des applications en dépend fortement. Cette complexité se retrouve aujourd'hui dans des architectures généralistes avec par exemple l'émergence des mémoires non-volatiles fournissant une grande capacité mais des performances inférieures à la mémoire habituelle.

*. Travail dirigé par Brice Goglin.
2. La mémoire à haut débit du KNL est en fait une MCDRAM (*Multi-Channel DRAM*), différente du standard HBM mais les performances sont similaires. On utilisera l'acronyme HBM pour simplifier le reste du document.

Dans cet article, nous présentons une interface pour aider à gérer la complexité de ces architectures mémoire. Notre approche consiste à disposer d'un ensemble d'attributs et d'une représentation cohérente de ces différents types de mémoires qui pourraient être utiles aux développeurs d'applications, de bibliothèques parallèles et de supports exécutifs. Nous proposons cette interface comme une extension de `hwloc`, le standard *de facto* pour exposer la localité des ressources matérielles en calcul haute performance. Des premiers résultats sur des applications très différentes confirment l'intérêt de ces attributs pour adapter la mémoire cible aux besoins des applications.

2. Exposition de la localité mémoire aux applications

La plupart des runtimes de calcul parallèles se basent sur `hwloc` [1] pour découvrir les topologie des plates-formes de HPC. [3] Il construit une hiérarchie d'objets basée sur l'inclusion et la localité physique sur un serveur. Depuis `hwloc` 2.0, les objets mémoire sont attachés à la hiérarchie CPU pour montrer quels cœurs sont locaux à un nœud NUMA donné [3], ce qui permet de modéliser les architectures mémoire à plusieurs niveaux.

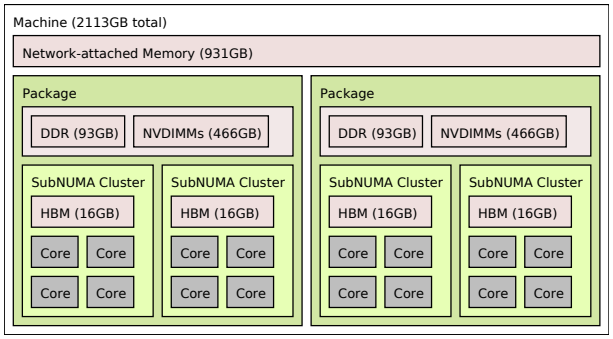

FIGURE 1 – Sortie `lstopo` de `hwloc` sur une plate-forme fictive qui a une mémoire commune connectée au réseau. Chaque package de CPU a des nœuds DDR et NVDIMM NUMA locaux, tandis que HBM est également attaché à chaque cluster SubNUMA.

Quatre types de mémoires sont disponibles sur la Figure 1. [4] Un programme s'exécutant sur un cœur dispose donc de 4 nœuds NUMA locaux pour allouer ses tampons mémoire, d'où la nécessité de fournir un moyen simple de choisir entre eux. Au niveau du processeur (*Package*), `hwloc` expose la mémoire DDR (mémoire classique) avant la NVDIMM (mémoire non-volatile [5]) car la DDR est généralement le nœud d'allocation par défaut. Malheureusement, ce choix ne correspond pas aux besoins de toutes les applications, par exemple pour des raisons de capacité. Il n'existe actuellement aucun moyen simple d'exposer des informations de performances, par exemple latence ou bande passante. `hwloc` pourrait lister la HBM avant la DDR pour guider les applications limitées par la bande passante à utiliser la HBM par défaut. Mais ce n'est pas possible sur cette plate-forme car la HBM et la DDR ne sont pas attachées au même niveau de la hiérarchie CPU.

Avec l'apparition de KNL, des interfaces ont été proposées pour allouer explicitement dans une mémoire rapide ou lente, par exemple memkind [2]. Cependant cette API est spécifique à KNL et propose uniquement le critère de la bande passante pour sélectionner la DDR ou la HBM. Des stratégies un peu plus automatiques ont été proposées en tenant compte des schémas d'accès des applications. Servat [7] et Narayan [5] (MOCA) utilisent une analyse post-mortem des accès mémoire en se basant par exemple sur les compteurs matériels. Cependant ils nécessitent de savoir quels nœuds NUMA sont *rapides* et d'y adapter l'application. C'était facile à cabler en dur dans le code pour KNL en mode *Flat*, mais ça n'est plus possible avec les plates-formes génériques avec NVDIMM, etc. Il faudrait plutôt effectuer ce processus automatiquement pen-

3. `http://www.open-mpi.org/projects/hwloc`.
4. Actuellement les plates-formes du commerce en supportent jusqu'à 3 simultanément.
5. Les barrettes de mémoire non-volatile peuvent être utilisé comme stockage très rapide ou comme mémoire lente mais de très grande capacité.

dant l'exécution et pour n'importe quels types de mémoires.
Nous proposons dans cet article d'exposer explicitement aux applications les caractéristiques des différentes mémoires disponible pour qu'elles puissent selon leurs besoin, choisir la bonne mémoire cible pour chaque allocation.

3. Contrôle des attributs de mémoire

Nous proposons de caractériser les mémoire par un ensemble d'attributs et métriques. Ceux-ci sont utilisés pour ordonner les périphériques de mémoire et cela permet aux développeurs de choisir la meilleure mémoire qui répond à leurs besoins lors de chaque allocation.

3.1. Bande passante

Depuis les architectures comme KNL qui incluent la mémoire haute capacité (DDR) et la mémoire à forte bande passante (HBM) [8], les applications sensibles à la bande passante doivent bénéficier d'une capacité logicielle à allouer dans la mémoire la plus appropriée.
Sur un système avec HBM, DDR et mémoire non volatile (NVDIMM), par exemple, nous proposons donc d'ordonner les mémoires comme suit lorsque la bande passante est le critère prioritaire :

$$\mathrm{HBM}_{\mathrm{BP}} > \mathrm{DDR}_{\mathrm{BP}} > \mathrm{NVDIMM}_{\mathrm{BP}} \tag{1}$$

3.2. Latence

Bien qu'apparemment liées, la bande passante et la latence peuvent ne pas être corrélées dans la pratique. Par exemple, sur KNL, la HBM et la DDR ont des latences assez similaires [6] alors que leurs bandes passantes sont très différentes. De plus, les applications de type *Pointer Chasing* bénéficient beaucoup plus d'une faible latence que d'une forte bande passante.
Notre proposition d'ordonner les différentes mémoire conduirait donc, quand la latence est le critère prioritaire, à l'ordre suivant :

$$\mathrm{DDR}_{\mathrm{Lat}} \simeq \mathrm{HBM}_{\mathrm{Lat}} > \mathrm{NVDIMM}_{\mathrm{Lat}} \tag{2}$$

Il faut noter ici que cet ordre compare la priorité des mémoires pour des allocations sensibles à la latence, et pas la latence elle-même (plus elle est **faible**, plus la mémoire est prioritaire, donc à gauche dans notre équation).
Dans notre exemple, l'application ne saura pas si elle doit allouer sur la DDR ou la HBM puisque leur priorité sont similaires. Mais elle pourra regarder d'autres critères comme la capacité pour faire son choix.

3.3. Capacité

La capacité des différentes mémoires est évidemment un critère important pour choisir où allouer, en particulier quand les mémoires sont petites, par exemple 16Go de HBM sur KNL. Notre proposition d'ordre par capacité est évidemment :

$$\mathrm{NVDIMM}_{\mathrm{Cap}} > \mathrm{DDR}_{\mathrm{Cap}} > \mathrm{HBM}_{\mathrm{Cap}} \tag{3}$$

Si plusieurs applications s'exécutent sur la même machine, leur comportement dynamique pourrait de plus imposer de considérer la capacité disponible plutôt que la capacité totale.

3.4. Localité

Nous envisageons enfin un attribut basé sur la localité qui peut être utile pour décider d'allocations partagées entre plusieurs tâches. En effet, les performances de l'échange de données entre

6. Mais la latence de la HBM dépend de la charge.

deux cœurs peuvent être améliorées si le tampon intermédiaire est alloué dans une mémoire proche. Cela pourrait par exemple servir pour l'allocation de fenêtres de mémoire partagée MPI selon la taille du communicateur (`MPI_Win_allocate_shared`).
Dans la Figure 1, la HBM a une forte localité pour quatre cœurs car elle n'est attachée qu'à un *SubNUMA Cluster* du processeur. La DDR et la NVDIMM sont locales à deux fois plus de cœurs (intégralité du processeur) et la mémoire connectée au réseau est partagée par la machine entière. Cet attribut décrit donc en fait la hiérarchie du sous-système mémoire à plusieurs niveaux :

$$\mathrm{HBM}_{\mathrm{Loc}} > \mathrm{DDR}_{\mathrm{Loc}} = \mathrm{NVDIMM}_{\mathrm{Loc}} > \mathrm{Network}_{\mathrm{Loc}} \tag{4}$$

4. Exposer les attributs mémoire aux applications

Nous proposons d'étendre l'interface de programmation de `hwloc`[7]. Comme indiqué dans la Figure 2, il s'agit d'une part d'exposer les ordres des mémoires selon les critères présentés à la section précédente, et d'autre part de mettre à disposition les métriques elles-mêmes, par exemple la latence des mémoires.

Ordonner les mémoires selon un/des attributs :
`get_ordering(attribute, [attribute])`

Obtenir la meilleure mémoire pour un attribut :
`get_best_target(attribute)`

Obtenir la valeur d'un attribut pour une mémoire :
`get_value(attribute, target)`

FIGURE 2 – Résumé des fonctionnalités principales de l'API.

La plupart de ces fonctions tiennent en fait compte des cœurs qui vont accéder à la mémoire (l'*initiator*). En effet, la bande passante d'une HBM (mais pas sa capacité) sera différente si elle est accédée par un cœur d'un autre processeur. Une fois l'initiateur connu et les mémoires qui lui sont locales (les *targets*), `get_ordering()` construit un ordre de ces mémoires en les classant selon l'attribut sélectionné.
La fonction `get_best_target()` permet de déterminer rapidement la meilleure cible pour une allocation depuis un cœur donné. Mais si cette allocation échoue (par exemple par manque de capacité disponible), il faudra regarder l'ordre des autres mémoires pour décider où réessayer.
Enfin `get_value()` permet de connaître les attributs exacts de chaque mémoire (par exemple sa latence en nanosecondes), ce qui peut par exemple permettre à une application d'ordonner elle-même les mémoires selon des critères plus complexes.

5. Découvrir les valeurs des attributs

Dans cette section, nous décrivons comment obtenir les valeurs de performances utilisées dans la section précédente pour ordonner les mémoires.

5.1. Informations matérielles

`hwloc` va utiliser la table *Heterogeneous Memory Attributes Table* (HMAT) introduite dans la révision 6.2 de la spécification ACPI[8] qui devrait être disponible sur les prochaines plates-formes pour décrire des hiérarchies de mémoire complexes.
Cette table peut exposer les latences et les bandes passantes théoriques entre tous les initiateurs (ensembles de cœurs) et toutes les cibles de mémoire (nœuds NUMA). Nous avons contribué à l'utilisation de ces tables à partir de Linux 5.2 pour qu'elles soient exposées aux applications

7. L'interface détaillée est en cours de développement dans `https://github.com/bgoglin/hwloc/blob/mmms/include/hwloc/memattrs.h` et devrait être disponible dans `hwloc` 2.3 mi-2020.
8. `https://uefi.org/specifications`

dans le système de fichiers virtuels `sysfs`. Cependant cela se limite actuellement aux performances des accès locaux. Il est donc pour le moment impossible de comparer une DDR locale avec une HBM d'un autre processeur.

5.2. Mesures expérimentales

En attendant que la table ACPI HMAT soit effectivement implémentée dans toutes les plates-formes, nous pouvons mesurer expérimentalement les valeurs de nos attributs. Par ailleurs, cela permettra toujours de contourner les tables ACPI incomplètes ou erronées comme c'est souvent le cas.

Par exemple, les performances de la DDR sur Intel *Cascade Lake* ont été mesurées empiriquement à environ 80 Go/s et 285 ns de latence tandis que les performances de la NVDIMM étaient de 10 Go/s et 860 ns de latence [9]. Bien que ces valeurs dépendent du nombre de threads accédant à mémoire, le modèle d'accès, etc., ils sont suffisants pour ordonner les mémoires.

Quand `hwloc` ne pourra récupérer les valeurs HMAT dynamiquement dans `sysfs`, il utilisera les résultats de mesures expérimentales préalablement stockées dans un fichier XML.

6. Cas d'utilisation : Allocateurs de mémoire contrôlés

Notre API peut être utilisée par les supports exécutifs et bibliothèques parallèles pour fournir des allocateurs de mémoire et des politiques de placement respectant finement les affinités et besoins des tâches de calcul. Nous avons expérimenté cette idée en implémentant notre propre allocateur que nous avons testé dans l'application Graph500 [4] (version 3.0.0) qui utilise des accès mémoire irréguliers [6]. Les performance sont mesurées par une moyenne harmonique des arêtes traversées par seconde (TEPS).

TABLE 1 – Performances sur Graph500 en arêtes traversées par seconde.

Graph Size	DDR	NVDIMM
2.15 Go	3.423	2.056
4.29 Go	3.459	2.067
8.59 Go	3.481	2.084
17.18 Go	3.343	2.107
34.36 Go	2.990	1.044

(a) Xeon : 16 processus MPI sur un seul processeur en utilisant ses DDR et NVDIMM locales.

Graph Size	HBM	DDR
2.15 Go	0.418	0.415
4.29 Go	0.402	0.396

(b) KNL : 16 processus MPI sur un *SubNUMA Cluster* en utilisant ses HBM et DDR locales.

Nous réalisons des expériences sur un serveur composé de 2 Xeon *Cascade Lake* 6230 (20 coeurs chacun) avec 192Go de DDR et 768Go de NVDIMM chacun, et sur un serveur KNL en mode SNC-4 Flat (processeur découpé entre 4 clusters, avec chacun une DDR de 24Go et une HBM de 4Go). Comme Graph500 impose 2^n tâches par défaut, nous avons utilisé 16 processus. Notre allocateur mémoire expérimental `mem_alloc(...,[attribut])` est utilisé avec différents critères pour déplacer les allocations. Les résultats sont présentés dans les tables 1a et 1b.

Sur le Xeon, la DDR fournit des résultats entre 1,5 et 3 fois plus élevés. Cela permet de confirmer que cette application devra spécifier en priorité la latence ou la bande passante comme critère dans notre allocateur.

Sur le KNL, la DDR fournit des résultats similaires à la HBM. Comme la latence des deux mémoires est similaire tandis que la bande passante est très différente (90Go/s contre 350 environ), ce résultat montre que le critère de bande passante n'est en fait pas adapté pour cette allocation

(le gain qu'il procure est trop faible pour justifier de gaspiller la capacité de la HBM).
Ces résultats confirment ce qui était attendu : l'application Graph500 est plutôt limitée par la latence car elle effectue des accès mémoire avec dépendances et donc peu pipelinables. Sur une application limitée par la bande passante, par exemple lors d'accès mémoire pipelinés comme dans le test Stream, c'est par contre le critère de bande passante qu'il faudra indiquer à notre allocateur.
Ainsi, avec notre allocateur et les critères de bande passante et latence, nous sommes capables ici d'avoir une allocation sur une mémoire adaptée pour ces deux architectures très différentes. Combiné avec le critère de capacité, ce travail nous a permis ici de dynamiquement adapter l'allocation selon les besoins de l'application et selon la mémoire effectivement disponible.

7. Conclusion et perspectives

Nous avons présenté une interface pour gérer la complexité des nouveaux systèmes de mémoire composés de mémoires hétérogènes à plusieurs niveaux. Elle permet d'effectuer des allocations pendant l'exécution pour n'importe quel type de mémoire, sans que le développeur n'ait à connaître les nœuds NUMA *rapides* et ni que l'application n'ait à être modifiée ou lancée différemment selon la plate-forme.
Notre approche se concentre sur la spécification d'un certain nombre d'attributs de mémoire et d'une API pour interroger et classer les périphériques mémoire. Ces attributs représentent des caractéristiques de haut niveau relativement faciles à utiliser : la bande passante, la latence, la capacité et la localité. En faisant remonter ces infos aux applications, nous montrons qu'il est possible d'implémenter des allocateurs permettant aux applications de spécifier leurs besoins pour chacune des allocations. C'est un travail qui permet d'améliorer la productivité et obtenant facilement les mêmes performances que lors d'une optimisation manuelle des allocations pour chaque plate-forme.
Parmi les défis qui nous restent à traiter se trouve d'abord la gestion des nombreuses mémoires disponibles, locales ou non. Sur un serveur à 4 processeurs Xeon avec des NVDIMMs, il est possible d'avoir 8 nœuds NUMA DDR (chaque processeur peut être configuré en 2 *SubNUMA Clusters* comme sur la Figure 1) et 4 NVDIMMs (un par processeur). Exposer des métriques pour tous ces nœuds ne sera pas forcément très pratique pour les applications car seuls les nœuds locaux sont utiles en général (1 DDR et 1 NVDIMM). Mais si l'application est irrégulière et les nœuds locaux sont pleins, dans quel nœud distant vaut-il mieux allouer, sachant qu'ils d'être pleins eux aussi dans un futur proche ?
Ensuite il nous faut réfléchir à la précision des valeurs que l'on veut manipuler. Si la DDR et la HBM ont des latences proches, faut-il tout de même exposer leurs valeurs précises et différentes, sachant qu'elles sont difficiles à mesurer et peuvent varier avec la charge ? Faut-il également distinguer les valeurs en lecture et écriture puisqu'elles sont significativement différentes pour la NVDIMM ? Par ailleurs, nous allons devoir prendre en compte les nouveaux caches situés devant les mémoires (*Memory-side Cache*) comme on en trouve sur nos plates-formes lorsqu'elles sont configurées différemment. [9] Ces caches rendent les performances théoriques plus difficiles à exprimer et donc plus difficiles à exposer pour aider les allocateurs.
Enfin il nous faudra gérer d'autres types de mémoire pouvant être exposés par certains périphériques (GPUs NVIDIA sur architecture POWER, disques NVMe, mémoire attachée au réseau, etc.). Ces mémoires ont des contraintes différentes, en terme de cohérence ou disponibilité par exemple, qui pourraient nécessiter l'ajout d'attributs supplémentaires dans notre interface.

9. KNL en mode Cache, Xeon en mode *2-Level-Memory*.

Bibliographie

1. Broquedis (F.), Clet-Ortega (J.), Moreaud (S.), Furmento (N.), Goglin (B.), Mercier (G.), Thibault (S.) et Namyst (R.). – hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications. – In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pp. 180–186, Pisa, Italia, février 2010. IEEE Computer Society Press.
2. Cantalupo (C.), Venkatesan (V.), Hammond (J. R.) et Hammond (S.). – User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies, 2015.
3. Goglin (B.). – Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications. – In *International Symposium on Memory Systems*, *MEMSYS'16*, MEMSYS'16, Washington, DC, 2016. ACM.
4. Murphy (R. C.), Wheeler (K. B.), Barrett (B. W.) et Ang (J. A.). – Introducing the Graph500. *Cray Users Group (CUG)*, vol. 19, 2010, pp. 45–74.
5. Narayan (A.), Zhang (T.), Aga (S.), Narayanasamy (S.) et Coskun (A. K.). – MOCA : Memory Object Classification and Allocation in Heterogeneous Memory Systems. – In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium*, Vancouver, BC, Canada, mai 2018. IEEE.
6. Peng (I. B.), Gioiosa (R.), Kestor (G.), Vetter (J. S.), Cicotti (P.), Laure (E.) et Markidis (S.). – Characterizing the performance benefit of hybrid memory system for HPC applications. *Parallel Computing*, vol. 76, 2018, pp. 57–69.
7. Servat (H.), Pena (A.), Llort (G.), Mercadal (E.), Hoppe (H.-C.) et Labarta (J.). – Automating the Application Data Placement in Hybrid Memory Systems. – In *Proceedings of the IEEE International Conference on Cluster Computing*, Hawaii, USA, septembre 2017.
8. Sodani (A.), Gramunt (R.), Corbal (J.), Kim (H.-S.), Vinod (K.), Chinthamani (S.), Hutsell (S.), Agarwal (R.) et Liu (Y.-C.). – Knights Landing : Second-Generation Intel Xeon Phi Product. *IEEE Micro*, vol. 36, n2, mars 2016, pp. 34–46.
9. van Renen (A.), Vogel (L.), Leis (V.), Neumann (T.) et Kemper (A.). – Persistent Memory I/O Primitives. *arXiv e-prints*, vol. abs/1904.01614, avril 2019.