Une stratégie efficace pour le placement de processus en environnement multicœur

Jérôme Clet-Ortega

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Une stratégie efficace pour le placement de processus en environnement multicœur

Jérôme Clet-Ortega

Université Bordeaux 1,
LaBRI A29', 351 cours de la Libération,
33405 TALENCE - France
jerome.clet-ortega@labri.fr

Résumé
L'exploitation de ces machines sophistiquées, dites « multicœurs », est un défi constant pour les développeurs des implémentations MPI, qui doivent offrir de très bonnes performances en terme de communication réseau, mais également en terme de communication en mémoire partagée. Une stratégie de placement de processus efficace, en adéquation avec cette structure hiérarchique, est capitale pour atteindre les performances optimales des applications.Ainsi, cet article propose un mécanisme efficace pour agencer les processus MPI sur ces machines multicœurs, se basant sur un outil de découverte de topologie et une analyse fine du modèle de communication applicatif. Pour exemple, les expériences réalisées démontrent l'influence des stratégies de placement sur les performances globales. Nous avons notamment découvert qu'une stratégie d'accroissement des communications intranœud dégrade souvent les performances alors qu'une utilisation réfléchie du cache semble être un facteur plus influent. Une stratégie beaucoup plus sophistiquée, prenant en considération la hiérarchie mémoire, est requise pour observer une amélioration des performances.

Mots-clés : Passage de message, architecture multicœur, placement de processus

1. Introduction

Au cours des dernières années, l'architecture des ordinateurs et notamment des processeurs qui les composent a considérablement progressé. Les grappes de calculs en sont une parfaite illustration : d'un regroupement de machines monoprocesseurs interconnectées par un simple réseau, elles sont passées à des structures complexes et fortement hiérarchiques. Les nœuds sont désormais composés de puces multicœurs partageant plusieurs bancs mémoire. Les progrès effectués par les grands constructeurs tel « *l'HyperTransport* » d'AMD ou encore le « *Quick Path Interconnect* » d'INTEL vont dans ce sens. Selon la position du cœur et celle du banc mémoire, le temps d'accès mémoire est plus ou moins important. Cela est souvent désigné sous le terme d'effet NUMA (*Non-Uniform Memory Access*). L'architecture mémoire est également complexifiée par le développement des différents niveaux de cache. Ce partage de ressources mémoire dépend de l'architecture du processeur et diffère d'un constructeur à l'autre.

Exploiter ces architectures pour des applications parallèles est une chose, cependant les exploiter de façon à en tirer la quintessence est un réel défi. Il est nécessaire d'utiliser une implémentation MPI [10] capable de tirer efficacement parti d'un environnement multicœur. Étant donné que le standard MPI n'est pas dépendant de l'architecture sous-jacente, il revient à l'implémentation MPI de faire le pont entre les performances applicatives et les performances matérielles. Pour preuve, les récentes implémentations telles que OPEN MPI [6] ou MPICH2 [2] offrent un niveau de performances extrêmement satisfaisant sur les architectures multicœurs.

Cependant, afin d'exploiter intégralement les ressources matérielles, les processus MPI doivent être placés avec précaution sur les cœurs de la machine. Cette politique de placement doit être définie à la fois selon le schéma de communication applicatif et selon les caractéristiques de l'architecture sous-jacente. Par exemple, si certains processus de l'application communiquent entre eux plus fréquemment

que d'autres, ils peuvent être regroupés sur le même nœud. De cette façon, le pourcentage de communication intranœud (mémoire partagée) augmente au détriment des communications internœuds (réseau) beaucoup moins rapides. Cela se traduit par une amélioration notable des performances applicatives.
Dans cet article, nous proposons une méthode et des outils que nous avons utilisés pour permettre à une application MPI de tirer meilleur parti d'un environnement multicœur. Nous allons exposer un accroissement des performances, non pas dû à des modifications de l'implémentation MPI, mais dû à un placement réfléchi des processus. La suite de l'article est organisée ainsi : la partie 2 décrit comment le placement est calculé et sur quel ensemble d'algorithmes et d'outils il repose. Les courbes de performances des tests NAS illustrent les résultats expérimentaux de la partie 3. Par la suite, la partie 4 liste les travaux apparentés au domaine du placement de processus MPI et analyse certaines solutions. Une dernière partie est consacrée à la conclusion de cet article et aux perspectives d'avenir.

2. Calcul d'un placement de processus MPI efficace

Afin d'atteindre un placement efficace des processus MPI, il est nécessaire de réunir des informations sur l'architecture cible et sur le modèle de communication de l'application, puis de les analyser pour déterminer le meilleur placement possible. Ce choix devrait être conditionné par une stratégie elle-même définie par l'utilisateur. Actuellement, ce travail n'englobe pas tous les types d'applications MPI.

2.1. Considérations sur les applications et leur environnement

Dans cet article, nous nous focalisons sur les applications MPI *statiques*. Par *statiques* nous entendons que l'application n'use pas des mécanismes de processus dynamiques offerts par MPI-2. Les applications MPI hybrides qui reposent sur le *multithreading* sont également exclues. De plus, le nombre de processus est considéré comme constant pour une exécution. Les machines et donc tous les cœurs utilisés sont dédiés à l'application. Par ailleurs, pour un cœur donné, un seul processus MPI y est associé. Il est évident que dans un futur proche, la dynamicité entrera en compte. Cependant, notre intérêt pour les modèles de communication statiques n'est pas dénué de sens. En effet, nombre d'applications conservent un même schéma de communication pour différents jeux de données. Dans le cas d'un maillage régulier, par exemple pour une application de propagation d'ondes, un processus communique toujours avec les mêmes processus (voisinage) quel que soit le jeu de données.
Toutes ces hypothèses de départ nous permettent de calculer, avant le lancement de l'application, un couplage statique entre les processus MPI et les cœurs qui ne sera pas modifié en cours d'exécution.

2.2. Collecte des informations sur le matériel

Nous l'avons vu précédemment, les grappes de nœuds NUMA sont organisées de façon hiérarchique. En effet, prenons pour exemple l'architecture d'un processeur AMD OPTERON (Figure 1). Celui-ci est composé de 4 puces contenant chacun 2 cœurs. Un banc mémoire est associé à chaque puce. Un cœur dispose de son propre cache L1 (ne figurant pas sur le schéma) et son propre cache L2, qu'il ne partage pas avec les autres cœurs. La mémoire principale d'un cœur peut être accédée par n'importe quel autre cœur, cependant le temps d'accès croît avec la distance entre le cœur et le banc mémoire. Une carte réseau peut être branchée sur la puce 0 et/ou sur la puce 1. Puisque les cœurs placés sur ces puces sont physiquement proches du bus d'entrées/sorties, nous pouvons espérer obtenir des transactions réseau plus rapides pour les processus s'exécutant sur ces cœurs.
Il existe des outils, tel LIBTOPOLOGY ou PLPA sur LINUX, fournissant les informations qui nous sont nécessaires. Pourtant ces outils ne couvrent pas suffisamment d'architectures ou tout du moins ne délivrent pas d'informations assez précises. Pour y remédier, nous avons utilisé le mécanisme de découverte de topologie implémenté dans la suite d'outils PM^2 [12]. Celui-ci est utilisé pour améliorer l'ordonnancement des threads sur les multiples cœurs d'une machine [14]. À l'heure actuelle, ce mécanisme prend uniquement en charge les processeurs et ne fournit pas d'informations à propos des bus d'entrées/sorties ou des cartes réseau. Lorsque l'on applique la découverte de topologie à l'architecture présentée en figure 1, on obtient les informations de la figure 2.
À partir de ces données, nous générons une structure de données qui sera utilisée par les autres outils pour le calcul du placement. Cette structure de données est un graphe dont les sommets représentent les cœurs et dont les arêtes sont pondérées. Ce graphe est complet et non-orienté. Plus d'éléments mémoires

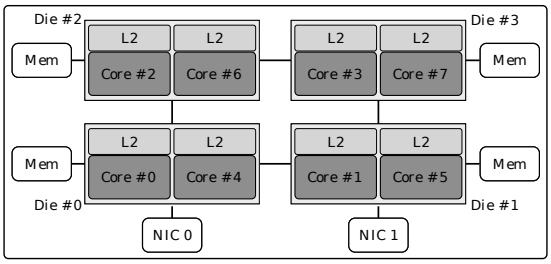


FIG. 1 – Un exemple de processeur hiérarchique : un OPTERON à 8 cœurs.

```
Machine:
 NUMANode + Die: Node#0(8GB) Die#0
  L2Cache + Core + L1Cache + SMTproc : L2#0(1MB) Core#0 L1#0(64kB) CPU#0
  L2Cache + Core + L1Cache + SMTproc : L2#4(1MB) Core#1 L1#4(64kB) CPU#4
 NUMANode + Die: Node#1(8GB) Die#1
  L2Cache + Core + L1Cache + SMTproc : L2#1(1MB) Core#0 L1#1(64kB) CPU#1
  L2Cache + Core + L1Cache + SMTproc : L2#5(1MB) Core#1 L1#5(64kB) CPU#5  ...
```

FIG. 2 – Extrait des informations générées par l'outil de découverte de topologie de PM^2

sont partagés par deux entités de calcul, plus le poids de l'arête les reliant est élevé. Il est à noter que l'effet NUMA entre également en jeu dans la pondération des arêtes.
La figure 3(a) présente les différentes valeurs obtenues pour l'architecture OPTERON (figure 1). Les arêtes ayant les poids les plus élevés correspondent à celles reliant les cœurs d'une même puce. En effet, ces derniers partagent directement le même banc mémoire. La valeur suivante (en terme de poids d'arête) concerne l'effet NUMA. Par exemple, il est plus coûteux pour le core 0 d'accéder à la mémoire attachée à la puce 3 qu'à celle attachée à la puce 1 et à la puce 2. Si la machine de calcul est composée de plusieurs nœuds, la plus petite valeur est associée aux communications utilisant le réseau.

$$\begin{pmatrix} 0 & 100 & 100 & 10 & 1000 & 100 & 100 & 10 \\ 100 & 0 & 10 & 100 & 100 & 1000 & 10 & 100 \\ 100 & 10 & 0 & 100 & 100 & 10 & 1000 & 100 \\ 10 & 100 & 100 & 0 & 10 & 100 & 100 & 1000 \\ 1000 & 100 & 100 & 10 & 0 & 100 & 100 & 10 \\ 100 & 1000 & 10 & 100 & 100 & 0 & 10 & 100 \\ 100 & 10 & 1000 & 100 & 100 & 10 & 0 & 100 \\ 10 & 100 & 100 & 1000 & 10 & 100 & 100 & 0 \end{pmatrix}$$

(a) Matrice de l'architecture de la figure 2.

$$\begin{pmatrix} 0 & 1000 & 10 & 1 & 100 & 1 & 1 & 1 \\ 1000 & 0 & 1000 & 1 & 1 & 100 & 1 & 1 \\ 10 & 1000 & 0 & 1000 & 1 & 1 & 100 & 1 \\ 1 & 1 & 1000 & 0 & 1 & 1 & 1 & 100 \\ 100 & 1 & 1 & 1 & 0 & 1000 & 10 & 1 \\ 1 & 100 & 1 & 1 & 1000 & 0 & 1000 & 1 \\ 1 & 1 & 100 & 1 & 10 & 1000 & 0 & 1000 \\ 1 & 1 & 1 & 100 & 1 & 1 & 1000 & 0 \end{pmatrix}$$

(b) Matrice du schéma de communication du NAS LU.B.8

Rang MPI	0	1	2	3	4	5	6	7
Numéro de cœur	3	7	4	0	6	2	5	1

(c) Résultat d'association pour le test NAS LU.B.8 sur le processeur AMD OPTERON.

FIG. 3 – Les 3 grandes étapes du calcul du placement.

2.3. Extraction du modèle de communication de l'application

En second lieu, il nous faut établir le schéma de communication de l'application. À l'inverse des données concernant le matériel, qui restent relativement pérennes, les informations liées à l'application sont extrêmement volatiles. Chaque application dispose de son propre modèle de communication, déterminé entre autres par le nombre de processus en lice. Nous avons choisi de prendre la quantité de données échangées entre les processus comme critère discriminant pour l'élaboration du modèle. Il nous faut donc calculer cette quantité pour chaque paire de processus au sein de l'application.

Plusieurs outils de traces et d'analyse, comme MPE (MPI PARALLEL ENVIRONMENT) sont déjà disponibles, cependant ils ne fournissent pas l'ensemble des informations nécessaires. Par exemple, MPE peut tracer tous les appels de fonction MPI de l'application. Il est aisé de deviner le modèle de communication en se focalisant sur les appels point à point. Il s'agit là d'une méthode simple qui nécessite uniquement de configurer l'implémentation MPI avec MPE et de relier l'application avec les bonnes bibliothèques. Pourtant, cette approche présente deux aspects limitants : d'une part le volume des traces générées peut être conséquent, d'autre part les communications collectives ne sont pas prises en compte. Modifier l'implémentation MPI directement pour recueillir l'information voulue a donc semblé une solution plus valable. De cette façon, la taille des traces engendrées est considérablement réduite et les communications collectives peuvent être prises en charge aisément.
En vue d'obtenir une information générique, non spécifique à l'implémentation, seules les données utilisateur MPI sont prises en compte. Les coûts induits par les mécanismes internes sont ignorés. De la même façon que pour la représentation des architectures, nous utilisons un graphe non-orienté avec des arêtes pondérées. Ici, le poids de ces arêtes équivaut à la quantité de données échangées entre les processus MPI. De ce fait, le sens de circulation des données n'apparait pas. Cependant il ne s'agit pas d'un facteur limitant puisque notre critère concerne toutes les paires de processus et non pas chaque processus. La figure 3(b) présente la matrice du schéma de communication pour un test NAS (LU.B.8).

2.4. Étape finale : placement

Nous avons obtenu, dans un premier temps, le graphe de la machine cible, décrivant la « distance » entre chaque entité de calcul. Par la suite, un second graphe a été généré pour présenter le schéma de communication de l'application qui affiche la quantité de données échangées entre les processus MPI. La dernière étape consiste à réaliser un plongement du graphe des communications dans le graphe de la machine, c'est-à-dire faire correspondre chaque processus MPI à un des cœurs de la machine.
Afin de résoudre ce problème NP-complet, nous avons utilisé le logiciel SCOTCH [8]. Celui-ci applique la théorie des graphes, grâce à une approche « diviser pour régner », pour résoudre des problèmes de calculs scientifiques courants tels le partitionnement de graphes ou de maillages, la renumérotation par blocs de matrices creuses, ou encore le placement statique. C'est ce dernier point qui nous intéresse. SCOTCH utilise des algorithmes de bi-partitionnement récursif pour résoudre ce problème [7] et permet de calculer des correspondances statiques pour des graphes de plus de 2^{16} sommets (le standard MPI précise que le rang des processus doit être codé sur 16 bits). Toutes les applications MPI peuvent donc être sujettes à notre placement, quel que soit le nombre de processus impliqués. Il est important de préciser que le temps de calcul du placement est largement inférieur à celui de l'exécution de l'application (quelques secondes pour une exécution de 15 mins avec 64 processus).
Sur la figure 3(c), le tableau affiche la correspondance statique calculée par SCOTCH pour le test NAS (LU.B.8) présenté précédemment, sur l'architecture OPTERON. Via cette correspondance statique, une commande spécifique pour chaque couple *(Application, Machine cible)* peut être produite. Nous utilisons ainsi la commande *numactl* pour affecter chaque processus au cœur lui étant associé.

3. Évaluation

Nous publions ici les résultats de l'application de notre méthode à différents tests NAS. L'environnement de calcul est tout d'abord décrit, puis les courbes des tests sont affichées et interprétées.

3.1. Plate-forme expérimentale

Les expérimentations ont été réalisées sur une grappe de 10 nœuds qui fait partie de la plate-forme GRID5000 [1]. Les nœuds ont la même architecture que celle de la figure 1 : chaque nœud dispose de 4 puces (AMD OPTERON 2218 - 2,6Ghz), chacune ayant deux cœurs. Les nœuds possèdent 32Go de mémoire (8Go par puce). Un réseau d'interconnexion Myrinet 10G est attaché aux puces 0 des nœuds.
Nous avons utilisé certains des tests NAS afin de démontrer l'intérêt et la pertinence de notre stratégie de placement. La première étape a été de lancer chaque test pour les tracer et en déduire leur schéma de communication respectif (cf 2.3). Étant donné que notre politique de placement se base sur la quantité de données échangées, nous nous focalisons sur les tests où cette valeur est significative : les tests de classe C et D pour 64 processus. De la même façon, seuls les tests ayant un modèle de communication irrégulier

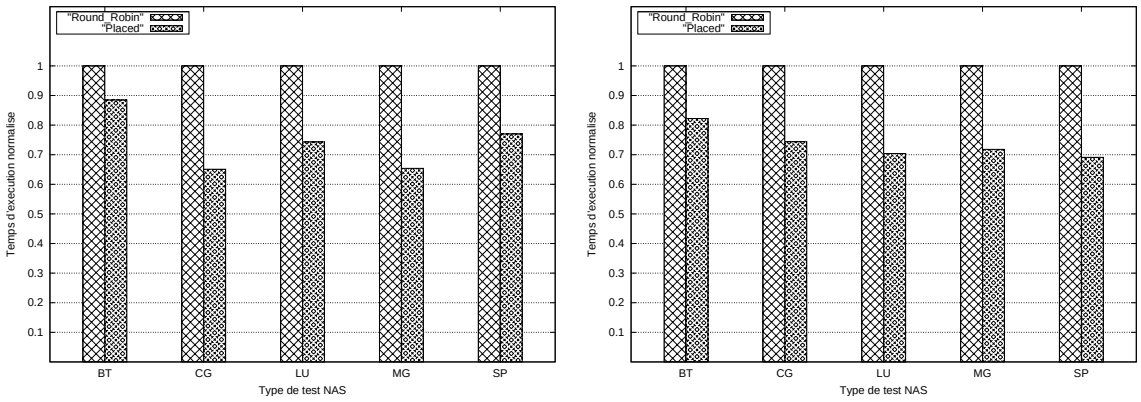
sont évalués. En effet, ils apparaissent être les plus touchés par une politique de placement de processus particulière en environnement multicœur. Les tests NAS suivants satisfont ces conditions : BT, CG, LU, MG et SP. Pour les tests BT, LU, MG et SP , la quantité totale de données échangées entre deux processus varie singulièrement d'une paire à l'autre. En ce qui concerne CG, cette valeur est approximativement la même. Cependant certains processus ne communiquent pas avec les autres.

3.2. Comparaison de deux implémentations MPI

L'un des points forts de notre approche est qu'elle est indépendante de l'application, tout comme les données collectées. Pour valider ce point, nous comparons les résultats expérimentaux de deux implémentations MPI. La première, MPICH2-NEMESIS, repose sur un système de communication intranœud très efficace grâce à la mémoire partagée [3]. La seconde implémentation est MPICH2-MX [11]. Le système de communication intranœud est aussi très efficace car basé sur un mécanisme interne au noyau.

Deux stratégies de placement sont comparées sur les figures 4 et 5. D'une part, nous avons une politique de *Round-Robin* où le processus numéroté i est placé sur le nœud numéro $n \equiv i \mod 8$ (chaque nœud possède 8 cœurs). Cette méthode de placement est régulièrement utilisée pour des applications MPI où le pattern de communication est inconnu, ou tout du moins ne fait pas l'objet de préoccupations particulières. C'est au système d'exploitation de sélectionner sur quel cœur sera affecté chaque processus. La seconde stratégie, dénommée *Placed*, correspond à la méthode décrite dans la section précédente.

En ce qui concerne MPICH2-NEMESIS, la différence entre les deux stratégies est notable, que ce soit pour la classe C ou pour la classe D. Comme on le voit sur les diagrammes, la stratégie *Placed* améliore de 25% les performances de l'application. Dans le cas de MPICH2-MX, la différence est encore plus importante. Le temps d'exécution est 34% plus court avec la politique *Placed* qu'avec la première.



(a) Class C - 64 Processus

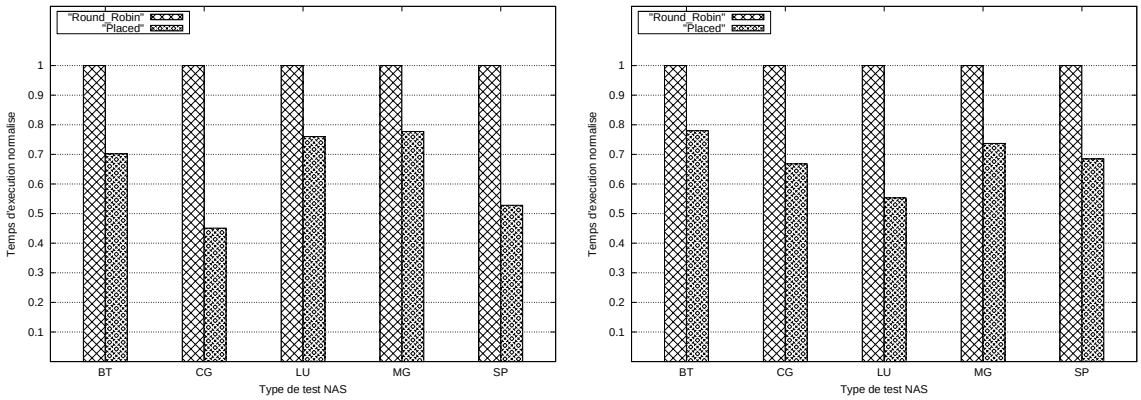
(b) Class D - 64 Processus

FIG. 4 – Comparatif de stratégies de placement de processus pour MPICH2-Nemesis :MX.

3.3. Analyse des politiques de placement

La politique *Placed* regroupe autant que faire se peut les processus communiquant entre eux, sur le même nœud. Comme nous le voyons sur le tableau 1, le pourcentage de communication intranœud (utilisant la mémoire partagée) augmente par rapport au pourcentage de communication internœud. L'accroissement des communications intranœuds n'est peut-être pas l'unique responsable de ce gain de performances. D'autres facteurs telle la hiérarchie mémoire peuvent en être à l'origine.

En vue de répondre à ces interrogations nous avons effectué une comparaison avec deux autres stratégies de placement appliquées aux tests NAS. La première est quasiment identique à *Round-Robin*. L'unique différence est que chaque processus est fixé sur un cœur particulier (par l'interface *numactl*). Ainsi, en cas de réordonnancement lors de l'exécution, le système d'exploitation ne pourra pas changer le processus d'entité de calcul. L'utilisation du cache de niveau 2 devient donc plus efficace. Toutefois,



(a) Class C - 64 Processus (b) Class D - 64 Processus

FIG. 5 – Comparatif de stratégies de placement de processus pour MPICH2-MX.

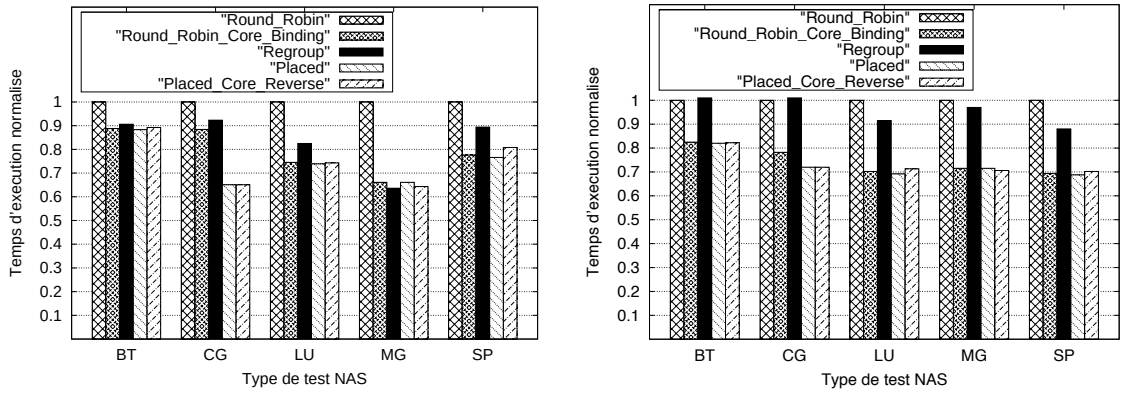
	BT		CG		LU		MG		SP	
	C	D	C	D	C	D	C	D	C	D
Round-Robin	56%	56%	33%	33%	66%	66%	62%	70%	55%	55%
Placed	69%	69%	85%	85%	82%	82%	73%	80%	69%	69%

TAB. 1 – Taux de communications intranœud pour les tests NAS.

le rapport {communications intranœud/communications internœud} reste le même. De l'autre côté, la seconde stratégie concentre sur le même nœud les processus MPI dialoguant entre eux (autant que possible), ce qui équivaut à la politique *Placed*. En revanche, les processus ne sont pas fixés sur des cœurs dédiés. C'est donc le système d'exploitation qui va décider du placement (comme la première stratégie *Round-Robin*). Le ratio de communication est donc optimal, néanmoins la hiérarchie mémoire ou encore le facteur NUMA ne sont plus pris en compte. L'impact sur l'utilisation du cache est clairement négatif.
Les résultats de ces tests sont présentés sur la figure 6. Les stratégies *Round-Robin* et *Placed* sont les mêmes que dans la partie 3.2. Les deux nouvelles stratégies décrites ci-dessus sont étiquetées *Round-Robin Core Binding* et *Regroup*. Comme nous pouvons l'observer sur ces résultats, améliorer le taux de communication intranœud n'est pas suffisant pour améliorer les performances globales. Cela produit même l'effet inverse pour certains cas. Globalement, notre stratégie *Placed* offre de meilleures performances que la nouvelle stratégie *Round-Robin Core Binding*. Ces résultats suggèrent que l'utilisation efficace du cache a une influence bénéfique sur les tests de performance. Une stratégie de placement fructueuse requiert de prendre en compte l'élément architectural.
Pourtant, nous pensons que l'effet NUMA de l'architecture OPTERON utilisée pour nos expérimentations n'est pas suffisamment important pour affecter notablement les performances applicatives. De même, le cache de niveau 2 n'est pas partagé par les cœurs d'une même puce. Ces deux points tendent à penser que notre stratégie se révélerait encore plus efficace sur des architectures plus complexes.

4. Travaux apparentés

En vue de profiter pleinement des ressources multicœurs, il est extrêmement important de veiller à un placement des processus efficace. En observant le standard MPI, on remarque qu'un ensemble de routines, destinées à créer et manipuler des topologies, est disponible. Cependant, peu d'applications utilisent ces routines [9]. Et pour cause, toutes les implémentations MPI ne fournissent pas des mécanismes de mise en place et de contrôle de topologie efficaces. Notre solution se révèle beaucoup plus générique et convient à n'importe quel type d'application MPI. Pour l'heure, elle n'est pas dépendante



(a) Class C - 64 Processus

(b) Class D - 64 Processus

FIG. 6 – Comparatif de stratégies de placement de processus pour MPICH2-Nemesis :MX.

de MPI, plutôt du modèle de passage de messages. Une application n'utilisant pas MPI peut s'y référer à condition que les données nécessaires soient récupérées.
L'application de la théorie des graphes au problèmes de placement de processus MPI sur des architectures cibles n'est pas inédite. Les constructeurs IBM [5] et HEWLETT-PACKARD [4] fournissent des implémentations MPI ayant recours à de telles méthodes. Ce sujet est également abordé dans l'article [9]. Malgré tout, pour chacune de ces solutions l'algorithme qui établit la correspondance est dépendant de l'implémentation MPI concernée. Notre proposition repose sur une partie d'un outil externe (SCOTCH) entièrement conçu pour le calcul de graphes. Celui-ci peut travailler sur de très grands graphes , dont le nombre de sommets dépasse largement le nombre de processus MPI utilisé dans n'importe quelle application MPI. De cette façon, le passage à l'échelle et les performances sont garantis.
Comme expliqué en partie 2.2, toutes les informations obtenues sur le matériel le sont grâce à l'outil de découverte de topologie de PM^2. Jusqu'il y a peu, ce mécanisme était complétement intégré dans cette pile logicielle. Il était donc difficile de l'utiliser pour une application externe. De multiples applications, comme les gestionnaires de processus, expriment un profond besoin de connaître précisément le matériel de calcul sous-jacent. De plus, les applications dites hybrides (utilisant à la fois MPI et OPENMP par exemple) pourraient également tirer parti de cette connaissance [13]. C'est pourquoi nous avons commencé à externaliser cet outil pour le rendre accessible à tous ces types d'applications.
En dernier lieu, il nous faut noter que pour établir le schéma de communication de l'application visée, une première exécution de celle-ci est nécessaire. De plus l'exécutable doit être compilé avec une version de l'implémentation MPI modifiée. D'autres travaux dans ce domaine utilisent ce même modèle. En outre, le constructeur HEWLETT-PACKARD met en avant le fait que la collecte de la quantité de données échangées entre deux processus MPI peut être une avancée pour les outils de trace [4]. Cette information pourrait permettre de rassembler les paires de processus et les organiser. Toutefois, il n'est toujours raisonnable, voire possible, de faire cette première exécution. Des alternatives peuvent être envisagées. S'appuyer sur les connaissances du programmeur de l'application pourrait en être une mais ce n'est pas toujours possible. Ce genre d'information pourrait éventuellement être déterminé à la compilation. Nous pourrions envisager de passer une option au compilateur *mpicc* qui générerait alors la ligne de commande adéquate. Ce sont ces solutions que nous aimerions explorer dans un futur proche.

5. Conclusion

Exploiter plus efficacement les diverses architectures fortement hiérarchiques et multicœurs du monde du calcul parallèle, tel est l'objectif de la méthode que nous avons décrite dans cet article. Grâce à la collecte d'informations sur le matériel sous-jacent et au modèle de communication de l'application recueilli, nous réalisons une correspondance entre les numéros de rangs des processus MPI et les numéros des cœurs de différents nœuds. Au final, une ligne de commande spécifique est générée. Notre

méthode n'est pas dépendante d'une quelconque implémentation MPI et utilise des logiciels libres et open-source. Nous avons analysé différentes stratégies de placement appliquées à des tests NAS. Il s'est avéré qu'augmenter le taux de communication intranœud n'est pas suffisant pour rendre les applications plus performantes. Une gestion réfléchie du cache fait obligatoirement partie d'un gain en termes de performances. Pour l'heure les résultats sont mitigés : notre stratégie de placement améliore les performances des applications MPI mais devrait être meilleure pour des architectures plus complexes. Nous espérons pouvoir éprouver notre méthode sur des architectures beaucoup plus hiérarchiques.
De nombreux travaux entrent dans nos perspectives. Comme vu à la section 4, l'élaboration d'un outil indépendant de découverte de topologie matérielle a déjà commencé. Les informations concernant les bus d'entrées/sorties ou encore les processeurs graphiques viendront s'y greffer par la suite. Par ailleurs, notre panel d'applications MPI doit être agrandi en relâchant certaines des contraintes fixées pour cet article. Nous nous sommes concentrés sur les applications où le nombre d'entités de calcul reste constant pendant l'exécution. Ainsi, à la fois les applications créant de nouveaux processus MPI et celles qui sont multithreadés sont exclues. Puisque la programmation hybride, mélangeant le modèle multithread et celui du passage de messages, est envisagée afin de programmer plus efficacement les architectures multicœurs, nous devons étudier le placement des processus MPI lorsque des sections parallèles OPENMP entrent en jeu. Une autre voie intéressante pourrait être de raffiner la collecte des données de communication, pour considérer une dimension *temporelle* en plus de la dimension *spatiale*. En effet, la quantité de données échangées entre deux processus peut varier durant l'exécution. Il nous faut donc faire ressortir du schéma des périodes de temps et refaire la correspondance *{processus,cœur}* pour chacune des périodes. Quelle est donc la granularité qui sera la plus profitable à l'application ?

Bibliographie

1. « Grid'5000 », 2009. `http://www.grid5000.fr`.
2. ARGONNE NATIONAL LABORATORY. « MPICH2 ». http ://www.mcs.anl.gov/mpi/2004.
3. Buntinas (D.), Mercier (G.) et Gropp (W.). « Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem ». *Parallel Computing, Selected Papers from EuroPVM/MPI 2006*, 33(9) :634–644, septembre 2007.
4. Solt (D.). « A profile based approach for topology aware MPI rank placement », 2007. `http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt`.
5. Duesterwald (E.), Wisniewski (R.W.), Sweeney (P.F.), Cascaval (G.) et Smith (S.E.). « Method and System for Optimizing Communication in MPI Programs for an Execution Environment », 2008. `http://www.faqs.org/patents/app/20080288957`.
6. Gabriel (E.), Fagg (G.E.), Bosilca (G.), Angskun (T.), Dongarra (J.J.), Squyres (J.M.), Sahay (V.), Kambadur (P.), Barrett (B.), Lumsdaine (A.), Castain (R.H.), Daniel (D.J.), Graham (R.L.) et Timothy S. WOODALL. « Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation ». Dans *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, septembre 2004.
7. Pellegrini (F.). « Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs ». Dans *Proceedings of SHPCC'94, Knoxville*, pages 486–493. IEEE, mai 1994.
8. Pellegrini (F.). « SCOTCH *and* LIBSCOTCH 5.1 *User's Guide* ». ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800, août 2008. `http://www.labri.fr/perso/pelegrin/scotch/`.
9. Larsson Träff (J.). « Implementing the MPI process topology mechanism ». Dans *Supercomputing '02 : Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
10. MESSAGE PASSING INTERFACE FORUM. « MPI-2 : Extensions to the Message-Passing Interface ». http ://www.mpi-forum.org/docs/mpi-20.ps, July 1997.
11. MYRICOM. « MPICH2-MX », 2009. `http://www.myri.com/scs/download-mpichmx.html`.
12. Namyst (R.), Denneulin (Y.), Geib (J.-M.) et Méhaut (J.-F.). « Utilisation des processus légers pour le calcul parallèle distribué : l'approche PM2 ». *Calculateurs Parallèles, Réseaux et Systèmes répartis*, 10(3) :237–258, July 1998.
13. Rabenseifner (R.), Hager (G.) et Jost (G.). « Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes ». Dans *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, pages 427–436, Weimar, Germany, février 2009.
14. Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). « Building Portable Thread Schedulers for Hierarchical Multiprocessors : the BubbleSched Framework ». Dans *EuroPar*, Rennes,France, 8 2007. ACM.