TreeMatch : Un algorithme de placement de processus sur architectures multicœurs

Emmanuel Jeannot, Guillaume Mercier, François Tessier

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

TreeMatch : Un algorithme de placement de processus sur architectures multicœurs

Emmanuel Jeannot, Guillaume Mercier et François Tessier

LaBRI - Inria - 200, avenue de la Vieille Tour, 33405 Talence - France
prenom.nom@inria.fr

Résumé
Depuis quelques années, les clusters de nœuds NUMA à processeurs multi-cœurs deviennent très répandus. Programmer efficacement ces architectures est un réel défi compte tenu de leur hiérarchie complexe. Afin d'en tirer pleinement profit, il est nécessaire de prendre en compte cette structure de façon précise et d'y faire correspondre le schéma de communication de l'application. Ce faisant, les coûts de communication sont réduits et l'on observe des gains sur le temps d'exécution total de l'application. Nous présentons ici comment nous utilisons d'un côté le schéma de communication et de l'autre une représentation fidèle de l'architecture pour produire une permutation des processus d'une application donnée, permettant ainsi une réduction des coûts de communication.
Mots-clés : Placement, processus MPI, coûts de communication, optimisation

1. Introduction

La simulation est employée depuis des années comme un moyen de corroborer des résultats théoriques dans des domaines scientifiques où les expériences sont physiquement impossibles (problème de temps ou d'échelle par exemple) ou bien trop coûteuses. Ces applications nécessitent une grande puissance de calcul compte-tenu du volume toujours plus important de données à traiter ou de la précision espérée des résultats. Les machines parallèles sont une réponse à ces contraintes : un maximum de processeurs sont assemblés afin de fournir la puissance nécessaire. Mais exécuter des applications sur ces architectures nécessite des outils spécifiques. L'Interface de Passage de Messages (MPI) est une solution standardisée permettant de distribuer efficacemment les calculs.
Afin de pouvoir exploiter au maximum l'architecture cible, MPI offre des communications hautement optimisées tant pour les échanges intra-nœuds qu'inter-nœuds. Mais les avancées dans le domaine du multi-cœur et du *many-core* rendent cette tâche de plus en plus difficile, notamment à cause de la nature hiérarchique des architectures. Par exemple, les processeurs récents présentent des effets NUMA (Non-Uniform Memory Access) compte tenu de l'organisation de leurs niveaux de mémoire.
Or, plus l'architecture est hiérarchique, plus la façon dont sont répartis les processus d'une application sur les différentes unités de calcul va avoir une influence sur les coûts de communication. Ainsi dans cet article, nous expliquons l'avantage du placement de processus pour les applications parallèles puis nous montrons que des gains substantiels peuvent être obtenus en prenant en considération le schéma de communication et l'architecture cible dans le calcul du placement.
La section 2 expose nos motivations. La section 3 décrit de précédents travaux portant sur le placement de processus. Les modèles que nous considérons sont décrits en section 4 et notre algorithme de placement, appelé TREEMATCH est présenté en section 5. Des expériences qui valident notre approche sont analysées en section 6 tandis que la section 7 conclut cet article.

2. Motivations

Une application utilisant MPI divise le travail en entités appelées *processus MPI*, qui s'exécutent en parallèle sur les différentes unités de calcul de l'architecture cible. Ces processus MPI s'échangent des

données (messages) durant l'exécution. Or, ces échanges ne sont pas nécessairement réguliers : un processus MPI ne communique pas toujours avec tous les autres processus MPI et le volume échangé de données peut varier entre deux messages consécutifs. C'est la représentation de ces différents volumes d'échanges que nous appelons *schéma de communication*. De plus, MPI ne tient pas compte de la complexité de la plate-forme cible alors que la latence ou la bande passante entre deux unités de calcul peuvent fortement varier en fonction de la topologie physique.
En effet, le paysage des architectures parallèles a subit d'importants changements avec l'arrivée des machines multi-cœurs et *many-core*. Jusqu'à l'ère des processeurs multi-cœurs, les architectures parallèles suivaient le principe des grappes de machines, à savoir une multiplication de machines monoprocesseurs. Mais les processeurs multi-cœurs ont fait évoluer cet aspect grâce à leurs mémoires fortement hiérarchiques, amenant par ailleurs des effets NUMA. Désormais, la topologie physique du réseau et la structure multi-cœur interne doivent être considérées si on espère réordonner de façon efficace les processus d'une application MPI.
En termes pratiques, cela signifie que les processus MPI doivent être placés sur des cœurs dédiés. Cette tâche ne fait pas partie des attributions du standard MPI mais dépend plutôt du gestionnaire de processus de l'implémentation MPI ou de son support d'exécution.

3. Travaux Connexes

Le problème du placement optimal de processus MPI a précédemment été étudié. Dans bien des cas, ce problème se réduit à du partitionnement et du plongement de graphe. Ainsi, une approche a été présenté dans [8] où un algorithme utilisant l'heuristique de Kernighan-Lin [2] y est décrit. Ces travaux sont cependant limités à des cas très spécifiques d'architectures matérielles. D'autres algorithmes fournissent également une solution au problème de plongement de graphe. Parmi les partitionneurs qui les implémentent, on citera Chaco [9], METIS [12] (ParMETIS pour sa version parallèle) et SCOTCH [6].
D'autres travaux se basent sur des topologies réelles mais uniquement au niveau du réseau. Les machines Blue Gene en particulier ont servi de cibles pour ces études ([16], ou [3]), de même que des réseaux Infiniband ([14] ou [10]).
MPIPP [5] est un ensemble d'outils permettant d'optimiser le temps d'exécution d'une application parallèle. Pour ce faire, MPIPP utilise un outil externe pour récupérer les informations sur l'architecture matérielle utilisée tandis que nous réalisons cette étape à l'exécution de TREEMATCH. MPIPP se contente également de répartir les processus sur les nœuds mais ne prend pas en compte la structure interne de ces derniers. Les effets NUMA ne sont alors pas considérés.
La technique d'assignation de processus à des cœurs a été étudiée dans des travaux plus récents. On citera par exemple [15] et [13]. Il est aussi important de noter qu'OpenMPI [7] ou MPICH2 [1], des implémentations libres du standard MPI, offrent des options d'assignation de processus à travers ORTE et Hydra, leur gestionnaire de processus. Ces placements sont néanmoins génériques et ne prennent pas en compte le schéma de communication de l'application.

4. Contexte

Pour effectuer un placement efficace des processus sur les cœurs physiques de l'architecture cible, nous avons besoin d'une part d'une représentation précise et fidèle de la plate-forme, et d'autre part du schéma de communication de l'application. En prenant en compte ces informations, l'algorithme TREEMATCH est capable de déterminer un placement des processus MPI de l'application. Ces processus seront ainsi assignés aux cœurs des machines cibles de façon à réduire autant que possible les coûts de communication tout en exploitant au maximum les unités de calcul mises à disposition (un processus par cœur).

4.1. Architecture matérielle

Récupérer une représentation précise de l'architecture matérielle est une tâche non triviale, en particulier lorsqu'il faut être le plus générique possible. Jusqu'à maintenant, aucun outil ne permettait d'obtenir ces informations avec un degré de précision incluant la hiérarchie mémoire (taille des caches, cœurs correspondants) et de façon portable sur un large panel d'architectures. Hwloc [4] (ou Hardware Locality),

développé dans notre équipe, permet d'extraire la topologie d'une machine : nœuds NUMA, sockets, cœurs, hiérarchie mémoire. Dans de précédents travaux [5],[13], l'architecture matérielle était représentée sous forme d'une matrice de vitesses de communication entre les cœurs. L'inconvénient de cette représentation est que l'on considère alors une topologie plate. On perd ainsi de l'information puisque les nœuds NUMA ont la caractéristique d'être fortement hiérarchiques. De ce fait, une représentation arborescente semble plus adaptée. Hwloc utilisant déjà ce type de structures, nous nous sommes donc également orientés vers cette solution. Dans ce type de représentation, la profondeur de l'arbre correspond aux différents niveaux de hiérarchie. Ainsi les nœuds, les niveaux de caches, les processeurs et les cœurs sont aisément reproduits. La répresentation du réseau, elle, n'est pas automatique. Bien entendu, les feuilles de l'arbre correspondent aux cœurs.

4.2. Schéma de communication

Le schéma de communication est une représentation sous forme de matrice $n \times n$ des communications entre processus MPI (cf. Fig. 1), où n est le nombre de processsus de l'application. Nous avons déterminé trois types de métriques :

- msg : le nombre de messages échangés entre les processus.
- size : la quantité de données échangées.
- avg : la taille moyenne d'un message obtenue à partir des deux mesures précédentes.

Dans la grande majorité des cas, à contexte équivalent (données en entrées, nombre de processus) les schémas de communication restent inchangés.

Afin d'extraire ces informations, nous avons dû tracer les communications, tant point-à-point que collectives, dans les couches bas niveaux des implémentations MPI considérées (MPICH2 et OpenMPI). Ce type d'observation des communications nous permet notamment de décomposer les messages collectifs.

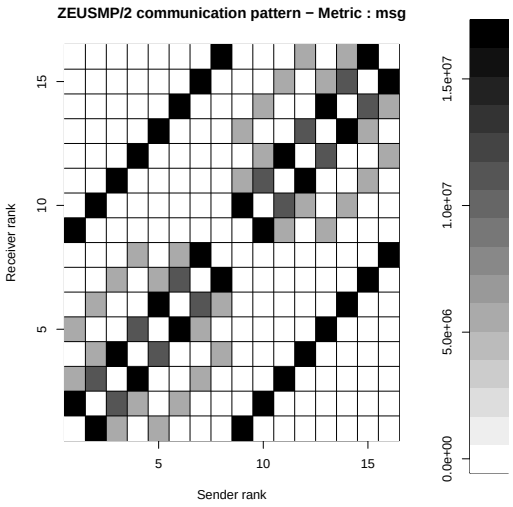


FIGURE 1 – Exemple de matrice de communication. Le niveau de gris donne le nombre de messages échangés par paire de processus.

Enfin, il est à noter que la collecte de ces informations nécessite une exécution préliminaire de l'application. En cas de changements dans le nombre de processus ou des données fournies en entrée, cette exécution doit être à nouveau effectuée.

5. L'algorithme TreeMatch

Nous présenterons dans cette section notre algorithme de placement de processus TREEMATCH. Cet algorithme propose une stratégie pour les petits cas (jusqu'à 128 processus) dont la complexité est exponentielle et une stratégie pour les cas plus important qui s'exécute en temps polynomial [1].

Notre algorithme de placement a pour but d'assigner chaque processus MPI à une unité de calcul afin de réduire les coûts de communication inter-processus. Cet algorithme, contrairement aux autres solutions proposées, permet de prendre en compte la complexité de la plate-forme qui accueille l'application (réseau, niveaux de cache, etc). Nous considérons l'arbre représentant la topologie comme équilibré (toutes les feuilles sont à la même profondeur) et symétrique (les nœuds de même profondeur ont la même arité). Cette limitation est finalement assez fidèle à la réalité où les nœuds d'un même cluster sont souvent tous identiques.

L'algorithme général est présenté en Algorithme 1. Afin d'expliquer par l'exemple le fonctionnement de TREEMATCH, nous allons nous appuyer sur la figure 2. Ici, la topologie est représentée par un arbre ayant 4 niveaux de profondeur et 12 feuilles (cœurs). Le schéma de communication des processus MPI est quant à lui représenté sous forme d'une matrice de taille 8×8 (nous avons donc 8 processus). L'algorithme débute à la profondeur 3. À ce niveau de profondeur, nous appellerons l'arité du niveau su-

1. Par manque de place, ces optimisations ne sont pas présentées dans cet article.

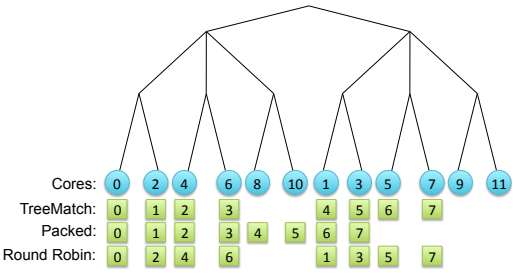
périeur k. Dans notre exemple, $k = 2$ est un diviseur de l'ordre $p = 8$ de notre matrice m. L'algorithme se poursuit ainsi ligne 6, c'est-à-dire à l'appel à `GroupProcesses`.

Algorithm 1: The TREEMATCH Algorithm

```
Input: T// L'arbre représentant la topologie
Input: m // La matrice de communication
Input: D // La profondeur de l'arbre
1 groups[1..D − 1]=∅ // Comment les nœuds sont groupés à chaque niveau
2 foreach depth← D − 1..1 do // On commence à partir des feuilles
3     p ← order of m
      // On étend la matrice de communication si nécessaire
4     if p mod arity(T, depth − 1) ≠ 0 then
5         m ←ExtendComMatrix(T,m,depth)
6     groups[depth]←GroupProcesses(T,m,depth)// On groupe les processus par affinité
7     m ←AggregateComMatrix(m,groups[depth]) // On aggrège les communications des groupes de processus
8 MapGroups(T,groups) // Construction de la permutation
```

Proc	0	1	2	3	4	5	6	7
0	0	1000	10	1	100	1	1	1
1	1000	0	1000	1	1	100	1	1
2	10	1000	0	1000	1	1	100	1
3	1	1	1000	0	1	1	1	100
4	100	1	1	1	0	1000	10	1
5	1	100	1	1	1000	0	1000	1
6	1	1	100	1	10	1000	0	1000
7	1	1	1	100	1	1	1000	0

(a) Matrice de communication



(b) Arbre de la topologie (Les carrés représentent les processus assignés en fonction de différents algorithmes)

FIGURE 2 – Exemple de données en entrée de l'Algorithme TREEMATCH

Function `GroupProcesses(T,m,depth)`

```
Input: T //Arbre de la topologie
Input: m // Matrice de communication
Input: depth // Niveau courant
1 l ←ListOfAllPossibleGroups(T,m,depth)
2 G ←GraphOfIncompatibility(l)
3 return IndependentSet(G)
```

Function `AggregateComMatrix(m,g)`

```
Input: m // La matrice de communication
Input: g // Liste des groupes de processus
       (virtuels)
1 n ← NbGroups(g)
2 for i ← 0..(n − 1) do
3     for j ← 0..(n − 1) do
4         if i = j then
5             r[i, j] ← 0
6         else
7             r[i, j] ← ∑_{i1∈g[i]} ∑_{j1∈g[j]} m[i1, j1]
8 return r
```

La fonction `GroupProcesses` génère dans un premier temps la liste de tous les groupes de processus possibles. La taille de ces groupes est donnée par l'arité k du niveau supérieur dans l'arbre ($k = 2$ dans notre exemple). Par exemple, on pourra grouper le processus 0 avec les processus de 1 à 7, puis le processus 1 avec les processus de 2 à 7, etc. Plus généralement, nous aurons $\binom{8}{2} = 28$ groupes de processus possibles. Comme nous avons $p = 8$ processus et que nous les groupons par paires ($k = 2$), nous souhaitons trouver $p/k = 4$ groupes qui n'ont aucun processus en commun. Afin d'y parvenir, nous créons un graphe des incompatibilités selon la contrainte suivante : deux groupes sont dit *incompatibles* s'ils partagent le même processus. Par exemple, le groupe (2,5) est incompatible avec le groupe (5,7) puisque le processus 5 ne peut pas être assigné à deux unités de calcul. Dans ce graphe, les sommets représentent donc les groupes, tandis que les arêtes déterminent les incompatibilités. Dans la littérature, un tel graphe correspond au complémentaire du graphe de Kneser. L'ensemble de groupes que nous

cherchons est donc un ensemble indépendant de ce graphe. Or, une propriété intéressante du complémentaire du graphe de Kneser est que dès lors que k divise p, les ensembles indépendants maximaux sont de taille p/k. Ainsi, un algorithme glouton trouvera toujours un ensemble indépendant de la taille souhaitée.

Cependant, tous ces ensembles n'ont pas la même qualité. Dans notre exemple, regrouper le processus 0 et le processus 5 n'est pas bénéfique puisqu'ils n'échangent que très peu de données. En effet, si ces processus étaient groupés, nous aurions beaucoup de communication à répercuter sur le niveau supérieur de la topologie (sur une communication réseau plus coûteuse par exemple). L'idée est donc de pondérer les sommets du graphe en fonction du volume de communication total du groupe réduit du volume de communication interne au groupe. Par exemple, en se basant sur la matrice de communication m, la somme des communications du processus 0 est de 1114 et celle du processus 1 est de 2104 pour un total de 3218. En groupant ces deux processus, on réduit ainsi cette valeur de 2000 (les deux processus s'échangeant chacun un volume de données de 1000). Le poids du sommet correspondant au groupe (0,1) est ainsi $3218 - 2000 = 1218$. Plus la valeur est faible, plus le groupe est intéressant. Malheureusement, trouver un ensemble indépendant de poids minimal est un problème NP-difficile. Par conséquent, nous utilisons une heuristique pour trouver un «bon» ensemble indépendant :

- plus petite valeur d'abord : on trie les sommets par ordre croissant de poids et on crée de façon gloutonne un ensemble indépendant maximal ;
- plus petite valeur maximale : on trie les sommets par ordre croissant de poids et on détermine un ensemble indépendant maximal de façon à ce que le sommet de poids maximum soit minimisé ;
- plus grand poids des voisins d'abord : on trie les sommets de façon décroissante en fonction du poids moyen des sommets voisins. Puis, on détermine un ensemble indépendant maximal en commençant par les sommets dont cette valeur est la plus importante [11].

Dans notre exemple, sans tenir compte de l'heuristique utilisée, nous trouvons l'ensemble indépendant de poids minimal suivant : (0,1),(2,3),(4,5),(6,7). Cette liste est affectée au tableau group[3], ligne 6 de l'algorithme TREEMATCH. Cela signifie par exemple que les processus 0 et 1 serons assignés à des feuilles partageant le même parent.

Virt. Proc	0	1	2	3
0	0	1012	202	4
1	1012	0	4	202
2	202	4	0	1012
3	4	202	1012	0

(a) Matrice agrégée (profondeur 2)

Virt. Proc	0	1	2	3	4	5
0	0	1012	202	4	0	0
1	1012	0	4	202	0	0
2	202	4	0	1012	0	0
3	4	202	1012	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

(b) Matrice étendue

Virt. Proc	0	1
0	0	412
1	412	

(c) Matrice agrégée (profondeur 1)

FIGURE 3 – Évolution de la matrice de communication à différentes étapes de l'algorithme

L'étape suivante consiste à créer des groupes en profondeur 2 de l'arbre. Afin d'y parvenir, il convient d'agréger la matrice m en fonction des communications restantes. La fonction `AggregateComMatrix` permet cela.

Dans notre exemple, les groupes de processus (0,1) et (2,3), que nous appellerons processus virtuels, échangent un volume de communication égal à 1012. Cette valeur est mise en $r[0,1]$ (voir figure 3(a)). La matrice r est de taille 4×4 (4 groupes) et vient écraser la matrice m (ligne 7 de l'algorithme TREEMATCH). La matrice m correspond désormais au schéma de communication entre paires de processus. La suite de l'algorithme consiste à créer récursivement des groupes de groupes jusqu'à remonter à la racine de l'arbre.

Dès lors que l'algorithme se positionne sur un nœud de l'arbre dont l'arité ne divise pas l'ordre (e.g. k ne divise pas p) de la matrice courante, il convient d'ajouter des groupes artificiels. Ces groupes ont un volume de communication nul. La fonction `ExtendComMatrix` permet d'effectuer cette étape. Dans notre exemple, la matrice ainsi obtenue est présentée en figure 3(b). L'itération suivante va générer la matrice de la figure 3(c). Cela correspond au dernier tour de boucle. La seule possibilité étant de créer le

Function `ExtendComMatrix(T,m,depth)`

Input: T `//Arbre de la topologie`
Input: m `// La matrice de communication`
Input: depth `// Niveau courant`
1 p ← order of m
2 k ←arity(T,depth+1)
3 **return** `AddEmptyLinesAndCol(m,k,p)`

groupe (0,1) qui est affecté à group[1].
L'algorithme se poursuit alors ligne 8 où les processus vont être assignés aux ressources. Pour ce faire, nous utilisons le tableau des groupes qui décrit leur hiérarchie. Un parcours de ce tableau permet de déterminer le placement de chaque processus. Par exemple, le groupe 0 (resp. 1) de group[1] est assigné à la partie gauche (resp. droite) de l'arbre. Si un groupe correspond à un groupe artificiel, aucun processus ne sera assigné dans le sous-arbre correspondant. À la fin de l'exécution, les processus 0 à 7 sont respectivement assignés aux feuilles (cœurs) 0,2,4,6,1,3,5,7 (voir figure 2(b)), ce qui s'avère être le placement optimal. En effet, on peut aisément voir que les processus qui échangent un volume important de données sont groupés dans le même sous-arbre, minimisant ainsi les coûts de communications.

6. Expériences

6.1. Conditions d'expérimentation

Nos expériences ont été effectuées sur la plate-forme PlaFRIM : 64 nœuds (2 processeurs quad-cœurs *Intel Xeon Nehalem*) sous Linux 2.6.27, interconnectés par réseau Infiniband. Enfin, nous avons utilisé ici Open MPI (version 1.5.4) et sur Hwloc (version 1.4.1).
L'application utilisée est ZEUS-MP/2 [2] qui fait de la mécanique numérique des fluides et est développée par le Laboratoire d'Astrophysique Numérique de l'Université de Californie à San Diego.
Le placement de processus étant assimilable à un problème de partitionnement de graphes, nous avons comparé notre solution a des solutions reconnues dans ce domaines tel que SCOTCH [6], ParMETIS [12], Chaco [9] et MPIPP [5]. Nous avons également inclut deux politiques de placement gloutonnes. *Packed* correspond à une identité logique entre processus et cœurs (processus 0 assigné au cœur logique 0, etc) et *Round Robin* correspond à l'identité physique (processus 0 assigné au cœur physique 0, etc). La figure 2 illustre ce type de placement.

6.2. Temps de calcul de la permutation

Nous avons mesuré les temps de calcul de permutation pour chaque partitionneur de graphes et TREEMATCH. On notera malgré tout l'absence de MPIPP5 [3] compte tenu du temps déjà très important de MPIPP1 (MPIPP5 étant environ 5 fois plus lent que ce dernier). Ces mesures sont des permutations calculées à partir de schémas de communication contenant entre 64 et 16384 processus et une topologie modélisant 128 commutateurs, reliant chacun 16 nœuds. On considère que chaque nœud est doté de 2 processeurs à 4 cœurs. Chaque mesure est une moyenne de 10 exécutions effectuées sur une machine à processeur INTEL NEHALEM cadencé à 2,66 GHz. Seuls les temps de calculs effectifs ont été mesurés. Les entrées/sorties (charger le graphe et écrire la solution sur le disque) ont été exclues. Enfin, on notera l'absence de Chaco au delà de 512 sommets dans le graphe de communication. Ceci s'explique par des erreurs lors de l'exécution au-delà de cette limite.

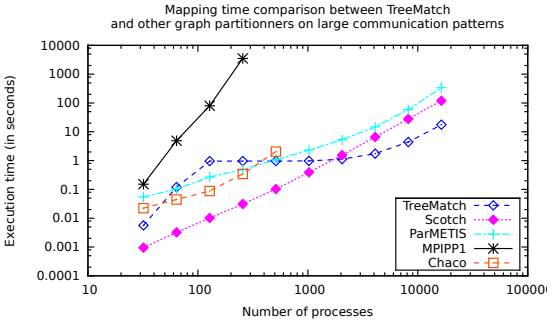


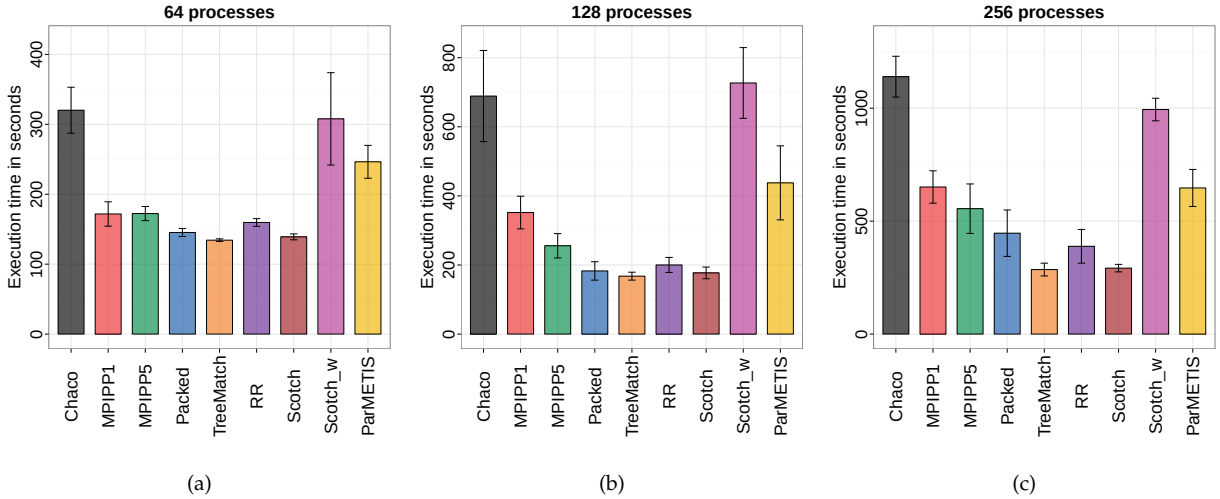
FIGURE 4 – Temps moyen de calcul d'une permutation pour différente méthodes de placement.

2. http ://lca.ucsd.edu/portal/codes/zeusmp2/
3. MPIPP est un algorithme randomisé. MPIPP1 utilise une seule graine. MPIPP5 exécute MPIPP1 5 fois avec 5 graines différentes et garde le meilleur résultat.

Sur les résultats présentés en figure 4 on peut noter les très bons résultats de Scotch sur des cas inférieurs à 2048 processus. Sur un graphe à 128 sommets, TREEMATCH met 957 ms tandis que Scotch trouve une solution en 10 ms. Cependant, au delà de 128 processus, TREEMATCH change de stratégie afin de conserver un bon compromis entre le temps de calcul et la qualité de la solution trouvée. On notera à ce propos qu'au delà de 2048 processus, TREEMATCH devient la solution la plus rapide avec un temps sept fois plus court que Scotch pour 16384 processus.

6.3. Comparaison des temps d'exécution

Comme expliqué précédemment, ces expériences ont été effectuées sur la plate-forme PlaFRIM. L'application utilisée est ZEUS-MP/2. La configuration utilisée à l'avantage d'augmenter la taille du problème en fonction du nombre de processus. Nous avons choisi de retenir pour ces expériences la métrique dont les temps d'exécution sont les plus bas. Il se trouve que la métrique *msg* est la plus efficace quel que soit le nombre de processus et d'itérations considérés. Les résultats sont présentés en figures 5(a), 5(b) et 5(c), respectivement pour des exécutions sur 64, 128 et 256 processus.



(a) (b) (c)

FIGURE 5 – Temps moyens d'exécution de ZeusMP/2 en faisant varier le nombre de processus (moyennes sur 10 exécutions entrelacées, metrique *msg*, 3000 itérations)

Le schéma de communication de cette application est très irrégulier. Ainsi, l'intérêt d'un placement efficace est d'autant plus important. On peut voir que TREEMATCH offre dans tous les cas des temps d'exécution moyens inférieurs aux autres solutions. Quand l'application est exécutée sur 256 processus, TREEMATCH offre même une réduction de l'ordre de 30% par rapport à Round Robin (RR), le placement par défaut de MPI. On remarque que les résultats de Scotch sont très proches de ceux de TREEMATCH. Malgré tout, Scotch utilise une réprésentation arborescente de l'architecture cible dont les arêtes sont pondérées (on peut se baser sur la latence ou la bande passante par exemple). En cas de poids élevés (de l'ordre de la centaine), le placement calculé devient improductif comme le montrent les temps de «Scotch_w» («w» pour *weighted*). TREEMATCH ne tient pas compte de ces poids mais considère la longueur du chemin dans l'arbre. On constate également que plus le nombre de processus augmente plus l'écart entre TREEMATCH et les autres solutions devient important. Ainsi, sur 64 processus, la différence entre notre solution et Packed sur 3000 itérations est de 5% à notre avantage tandis que sur 256 processus, ce gain avoisine les 37%. Concernant les techniques de partitionnement de graphes on voit que les différences sont importantes par rapport à Chaco ou ParMETIS.

7. Conclusion

Dans cet article, nous avons présenté notre algorithme TREEMATCH capable de trouver un placement de processus MPI efficace en fonction du schéma de communication de l'application et de l'architecture

sous-jacente. Afin de valider cet outil, nous avons effectué une série d'expériences sur une application de mécanique des fluides numérique en se comparant aux autres méthodes de placement. Les résultats comparés à ceux de Chaco, ParMETIS, MPIPP ou SCOTCH sont très encourageants. En général, plus le schéma de communication est irrégulier, plus nos gains sont importants. En particulier face à des placements dits gloutons comme *Packed* ou *RR*. Ces résultats obtenus sur ZeusMP/2 valident nos précédents travaux effectués sur les NAS (NASA Parallel Benchmarks).
Nos futurs travaux concernant TREEMATCH vont suivre plusieurs orientations. Entre autres, nous prévoyons d'améliorer la collecte du schéma de communication, par exemple au travers d'analyse statique du code de l'application. De plus, nous envisageons d'évaluer notre algorithme de placement sur davantage d'applications.

Bibliographie

1. Argonne National Laboratory. – MPICH2. – http ://www.mcs.anl.gov/mpi/2004.
2. B. W. Kernighan and S. Lin . – An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, vol. 49, n2, février 1970, pp. 291–307.
3. Balaji (P.), Gupta (R.), Vishnu (A.) et Beckman (P. H.). – Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems. *Computer Science - R&D*, vol. 26, n3-4, 2011, pp. 247–256.
4. 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)*. – Pisa, Italia, février 2010.
5. Chen (H.), Chen (W.), Huang (J.), Robert (B.) et Kuhn (H.). – MPIPP : an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. *In : ICS*, éd. par Egan (G. K.) et Muraoka (Y.). pp. 353–360. – ACM.
6. François Pellegrini. – Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. *In : Proceedings of SHPCC'94, Knoxville*. IEEE, pp. 486–493.
7. 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 Woodall (T. S.). – Open MPI : Goals, concept, and design of a next generation MPI implementation. *In : Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pp. 97–104. – Budapest, Hungary, September 2004.
8. Hatazaki (T.). – Rank reordering strategy for MPI topology creation functions. *In : Recent Advances in Parallel Virtual Machine and Message Passing Interface*, éd. par Alexandrov (V.) et Dongarra (J.), pp. 188–195. – Springer Berlin / Heidelberg, 1998. 10.1007/BFb0056575.
9. Hendrickson (B.) et Leland (R.). – *The Chaco User's Guide : Version 2.0*. – Rapport technique n SAND94–2692, Sandia National Laboratory, 1994.
10. Ito (S.), Goto (K.) et Ono (K.). – Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments. *Computer & Fluids*, avril 2012.
11. Kako (A.), Ono (T.), T. (H.) et Halldorsson (M. M.). – Approximation Algorithms for the Weighted Independent Set Problem. *In : LNCS*. pp. 341–350. – SPRINGER-VERLAG.
12. Karypis (G.) et Kumar (V.). – *METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. – Rapport technique, 1995.
13. Mercier (G.) et Clet-Ortega (J.). – Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. *In : EuroPVM/MPI*. pp. 104–115. – Espoo, Finland, 2009.
14. Rashti (M. J.), Green (J.), Balaji (P.), Afsahi (A.) et Gropp (W.). – Multi-core and Network Aware MPI Topology Functions. *In : EuroMPI*, éd. par Cotronis (Y.), Danalis (A.), Nikolopoulos (D. S.) et Dongarra (J.). pp. 50–60. – Springer.
15. Rodrigues (E.), Madruga (F.), Navaux (P.) et Panetta (J.). – Multicore aware process mapping and its impact on communication overhead of parallel applications. *In : Proceedings of the IEEE Symposium on Computers and Communications*, pp. 811–817.
16. Yu (H.), Chung (I.-H.) et Moreira (J. E.). – Blue Gene system software - Topology mapping for Blue Gene/L supercomputer. *In : SC*. p. 116. – ACM Press.