HAL
open science

Impact du placement des threads de progression pour les collectives non-bloquantes

Hugo Taboada

To cite this version:

Hugo Taboada. Impact du placement des threads de progression pour les collectives non-bloquantes. Compas 2016: conférence d'informatique en Parallélisme, Architecture et Système, Jul 2016, Lorient, France. hal-01355140

HAL Id: hal-01355140

https://hal.science/hal-01355140v1

Submitted on 22 Aug 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compas'2016 : Parallélisme / Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Impact du placement des threads de progression pour les collectives MPI non-bloquantes

Hugo Taboada

CEA, DAM, DIF, F-91297 Arpajon, France / Inria Bordeaux - Sud-Ouest, France
taboadah@ocre.cea.fr / hugo.taboada@inria.fr

Résumé
Les collectives MPI non-bloquantes permettent de recouvrir les communications par du calcul. Un bon taux de recouvrement est obtenu en effectuant les tâches de communication et des tâches de calcul en parallèle. Pour réaliser cela, certaines implémentations utilisent des threads de progression pour gérer les tâches de communication. Ces threads sont en concurrence avec les autres threads de l'application. Dans cet article, nous proposons un placement de ces différents threads en tenant compte de la topologie NUMA de la machine afin d'améliorer le taux recouvrement des communications collectives non-bloquantes.

Mots-clés : Collectives non-bloquantes, Placement, Thread, Thread de progression, MPI

1. Introduction

« Message Passing Interface » (MPI) est une norme définissant une bibliothèque de fonctions permettant la programmation parallèle par le biais d'envois de messages. Ceux-ci peuvent être bloquants ou non-bloquants. Cette dernière technique permet de recouvrir le temps de communication par du calcul. Certaines implémentations utilisent des threads de progression prenant en charge la progression des communications. Ils se retrouvent ainsi en concurrence non seulement avec les threads de calcul mais aussi avec d'autres threads de progression. Les changements de contextes engendrés ainsi que le partage des ressources de calcul peuvent dégrader les performances.
Dans cet article, nous nous concentrons sur les communications collectives non-bloquantes apparues lors de la version 3.0 de la norme MPI [3]. Nous proposons un placement de threads permettant un meilleur recouvrement. Pour cela, nous avons intégré notre solution au support exécutif MPC [11].
Nous ferons d'abord un bref rappel sur les communications MPI. Ensuite nous présenterons une politique de placement de threads mise en place pour assurer un maximum de recouvrement. Nous expliquerons les résultats obtenus, puis nous conclurons.

2. Contexte : les communications MPI non-bloquantes

Les grappes de calculs sont aujourd'hui constituées de plusieurs machines reliées entre elles, appelées des hôtes. Ces hôtes possèdent leur propre système d'exploitation ainsi que leur propre mémoire. Ils peuvent être constitués de plusieurs nœuds NUMA qui constituent un ensemble de cœurs, se partageant une partie de la hiérarchie mémoire. De ce fait, l'accès à cette

mémoire n'est pas uniforme entre 2 nœuds NUMA. Lorsqu'une application utilise plusieurs hôtes, les processus doivent s'envoyer des messages pour communiquer. MPI est une norme qui définit une bibliothèque de fonctions permettant aux différentes tâches MPI de communiquer à travers le réseau.

Il existe plusieurs types de communications : les communications point-à-point permettant à 2 tâches MPI de communiquer, et les communications collectives qui associent plusieurs tâches MPI dans le but d'effectuer une communication où chacune des tâches aura un rôle à jouer. Toutes ces communications peuvent être bloquantes ou non-bloquantes. Lorsqu'un appel bloquant est effectué (MPI_Send, MPI_Recv, MPI_Alltoall, etc.) , la tâche MPI attend que la communication soit terminée pour reprendre son fil d'exécution. À l'inverse, après l'appel à une fonction non-bloquante (MPI_Isend, MPI_Irecv, MPI_Ialltoall, etc.), la tâche MPI continue d'exécuter les instructions suivantes sans attendre que la communication soit réalisée. D'autres fonctions permettent d'attendre la fin de la communication (MPI_Wait) ou de tester si celle-ci a été effectuée (MPI_Test).

Les communications non-bloquantes permettent de recouvrir les communications avec du calcul. Néanmoins, pour obtenir un bon taux de recouvrement, il faut effectuer les tâches de communication en parallèle des tâches de calcul. Des techniques matérielles et logicielles existent. Certaines cartes réseaux embarquent un processeur dédié à effectuer les tâches de communication [8]. Lorsque cela n'est pas possible, la gestion de la progression des communications est déléguée aux supports exécutifs MPI de façon logicielle.

La plupart des implémentations MPI (MPICH, Open MPI, etc.) effectuent des appels des fonctions faisant progresser les communications dans toutes leurs routines pour faire progresser les communications mais avec cette méthode, la progression dépend de la fréquence d'appels à la bibliothèque MPI. Les développeurs d'applications peuvent aussi insérer des MPI_Test dans leur boucle de calcul pour faire progresser les communications mais cela demande un effort de plus dans le développement des applications. Pour ne pas avoir à gérer les appels à MPI_Test, d'autres implémentations tel que MPC génèrent des threads de progression pour prendre en charge les tâches de communication. Ces threads se retrouvent alors en concurrence avec les tâches de calculs ainsi qu'avec d'autres threads de progression. La progression dépend de la fréquence d'ordonnancement des threads de progression dans le cas où nous n'avons pas de ressources dédiées à ces threads. Une fréquence d'ordonnancement trop élevée provoque de nombreux changements de contexte qui peuvent être préjudiciables aux bonnes performances de l'application. Néanmoins, selon Torsten Hoefler et Andrew Lumsdaine [9], l'utilisation de threads de progression reste la meilleure option pour obtenir un bon taux de recouvrement.

Les collectives MPI non-bloquantes génèrent plusieurs threads de progression. Leur gestion est primordiale et plusieurs leviers permettent d'améliorer le taux de recouvrement : le placement de ces threads ainsi que leur ordonnancement. Dans cet article, nous nous concentrons sur le placement des threads de progression générés par les collectives MPI non-bloquantes. Nous proposons une méthode de placement pour les tâches MPI ainsi que les threads de progression lorsqu'une machine n'utilise pas tous les coeurs mis à sa disposition. C'est le cas quand une application est limitée par la quantité de mémoire d'une machine. Les tâches MPI ne peupleront pas tous les cœurs de la machine mais utiliseront bien toutes la quantité de mémoire de l'hôte.

Nous avons implémenté cette méthode dans le support exécutif MPC [11]. MPC a été développé pour améliorer l'interaction entre les différents modèles de programmation rencontrés dans les codes scientifiques. Sa spécificité est de proposer une implémentation MPI à base de thread. Les tâches MPI sont donc des threads comme les tâches OpenMP, les threads POSIX ainsi que les threads de progression [6].

3. Étude sur le placement des threads

Notre idée est d'utiliser les cœurs libres pour les threads de progression dans le but de ne pas gêner l'exécution des threads de calcul. Notre approche est de prendre en compte la topologie de la machine sous-jacente pour placer les différents threads mis en jeux. MPC utilise Hardware Locality [5] pour avoir une vision globale de la topologie de la machine. Il connait aussi tous les threads qu'il exécute au sein d'un hôte car il intègre son propre ordonnanceur de threads. Cela nous permet d'avoir un contrôle très fin sur le placement de tous les threads. Nous pouvons donc définir le placement de chaque thread à leur création ou pendant leur exécution. Nous avons commencé par lier les threads de progression aux cœurs sur lesquels les tâches MPI étaient liées, afin de ne pas avoir plusieurs threads de progression perturbant l'exécution du même thread de calcul.

3.1. Placement des tâches MPI

Nous avons ajouté un algorithme de placement des tâches MPI au sein du support exécutif MPC pour permettre un placement des threads de progression par nœud NUMA. Celui-ci prend en compte la topologie au sein même d'un hôte en tenant compte des effets NUMA.
Notre démarche est d'abord de compter le nombre de nœuds NUMA au sein d'un hôte. Nous avons ensuite divisé le nombre de tâches MPI par le nombre de nœuds NUMA afin de savoir combien de tâches MPI nous allons devoir placer par nœud NUMA. Sachant cela, il nous a suffit de placer les tâches MPI de façon à les écarter au maximum (politique scatter) au sein d'un nœud NUMA. Ce placement est défini par l'algorithme 1.

Algorithme 1 : Algorithme de placement des tâches MPI

Entrées : rang_mpi, nb_tâche_mpi, nb_cpu, nb_numa_par_hôte, cpu_par_numa

$\mathtt{id_numa} \leftarrow \lfloor \frac{\mathtt{rang_mpi} * \mathtt{nb_numa_par_hôte}}{\mathtt{nb_tâche_mpi}} \rfloor$;

$\mathtt{id_local} \leftarrow \mathtt{rang_mpi} - \lceil \frac{\mathtt{id_numa} * \mathtt{nb_tâche_mpi}}{\mathtt{nb_numa_par_hôte}} \rceil$;

$\mathtt{nb_tâche_mpi_dans_local_numa} \leftarrow \lceil \frac{\mathtt{id_numa}+1) * \mathtt{nb_tâche_mpi}}{\mathtt{nb_numa_par_hôte}} \rceil - \lceil \frac{\mathtt{id_numa}) * \mathtt{nb_tâche_mpi}}{\mathtt{nb_numa_par_hôte}} \rceil$;

$\mathtt{pos_local} \leftarrow \lfloor \frac{\mathtt{id_local} * \mathtt{cpu_par_numa}}{\mathtt{nb_tâche_mpi_dans_local_numa}} \rfloor$;

$\mathtt{pos_global} \leftarrow \mathtt{pos_local} + (\mathtt{id_numa} * \mathtt{cpu_par_numa})$;

retourner *pos_global*

Les figures 1a et 1b représentent les différents placements engendrés par les algorithmes de placement de threads au sein de MPC sur un hôte de 8 cœurs avec 2 nœuds NUMA. Nous y voyons la répartition des tâches MPI sur les cœurs. Chaque ligne représente le placement des tâches MPI pour un nombre de tâches MPI donné.
La figure 1a illustre le placement de MPC par défaut. Il s'agit d'un placement qui écarte les tâches MPI au maximum au sein d'un hôte pour permettre de peupler les cœurs libres avec des threads OpenMP. Cet algorithme ne prend pas en compte les différents nœuds NUMA au sein d'un hôte.
Sur la figure 1b, nous avons le placement que nous avons mis en place pour tenir compte des nœuds NUMA. Nous plaçons les threads de façon à avoir un nombre de tâches MPI équilibré entre les nœuds NUMA. Cela permet non seulement de placer les threads de progression par nœud NUMA mais aussi de répartir la charge de calcul sur les nœuds NUMA et de bénéficier au maximum des effets de caches.

Nombre de tâches MPI	Identifiant des coeurs 0	1	2	3	4	5	6	7
1	0							
2	0				1			
3	0			1			2	
4	0		1		2		3	
5	0		1		2		3	4
6	0		1		2	3	4	5
7	0		1	2	3	4	5	6
8	0	1	2	3	4	5	6	7

Noeud NUMA 0 — Noeud NUMA 1 — Hôte

(a) Default

Nombre de tâches MPI	Identifiant des coeurs 0	1	2	3	4	5	6	7
1	0							
2	0				1			
3	0		1		2			
4	0		1		2		3	
5	0	1	2		3		4	
6	0	1	2		3	4	5	
7	0	1	2	3	4	5	6	
8	0	1	2	3	4	5	6	7

Noeud NUMA 0 — Noeud NUMA 1 — Hôte

(b) Numa

FIGURE 1 – Placement des rangs MPI sur un hôte de 8 cœurs avec 2 nœuds NUMA en fonction du nombre de tâches MPI.

3.2. Placement des threads de progression

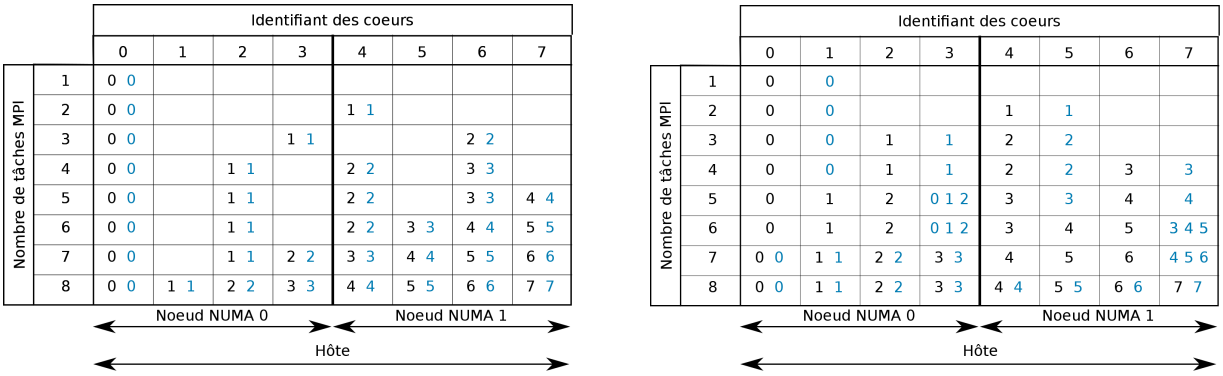
Le placement des threads de progression s'effectue en fonction de celui des tâches MPI. La figure 2 reprend les placements des tâches MPI donnés sur la figure 1. En bleu, nous affichons le placement des threads de progression générés par leur tâche MPI.

La figure 2a illustre le placement des threads de progression lorsqu'il sont liés sur les mêmes cœurs que les tâches MPI qui les ont générés. Ce placement est le placement des threads de progression par défaut dans MPC. La figure 2b décrit le placement résultant de l'algorithme 2 que nous avons mis en place pour permettre une meilleure progression. Nous plaçons les threads de progression sur les cœurs libres répartis au sein du même nœud NUMA. De cette manière, les threads de progression bénéficierons des effets de caches. S'il ne reste aucun cœur de libre, les threads de progression seront liés sur les mêmes cœurs que les tâches MPI.

Algorithme 2 : Algorithme de placement des threads de progression

Entrées : nb_numa_par_hôte, cpu_par_numa, nb_tâche_mpi_dans_local_numa, pos_global
si $\mathtt{nb_tâche_mpi_dans_local_numa} \geq \mathtt{cpu_par_numa}$ **alors**
  $\mathtt{pos_thread_progression} \leftarrow \mathtt{pos_global}$;
sinon
  $\mathtt{esp_moy_cpu_libres} \leftarrow \frac{\mathtt{cpu_par_numa}}{\mathtt{cpu_par_numa} - \mathtt{nb_tâche_mpi_dans_local_numa}}$;
  $\mathtt{pos_thread_progression} \leftarrow \lceil (\lfloor \frac{\mathtt{pos_global}}{\mathtt{esp_moy_cpu_libres}} \rfloor + 1) * \mathtt{esp_moy_cpu_libres} \rceil - 1$;
fin
retourner *pos_thread_progression*

Nous voyons sur la troisième ligne de la figure 2b, que pour 3 tâches MPI, ces tâches sont réparties sur les nœuds NUMA et les threads de progression associés sont sur les cœurs libres sur ces mêmes nœuds NUMA. Le but de notre méthode est de placer les threads de progression sur le même nœud NUMA que leurs tâches MPI associées. Pour 7 tâches MPI, nous avons donc un comportement similaire. Sur le premier nœud NUMA, il n'y a pas de cœur de libre alors les threads de progression sont liés sur les mêmes cœurs que les tâches MPI. En revanche, sur



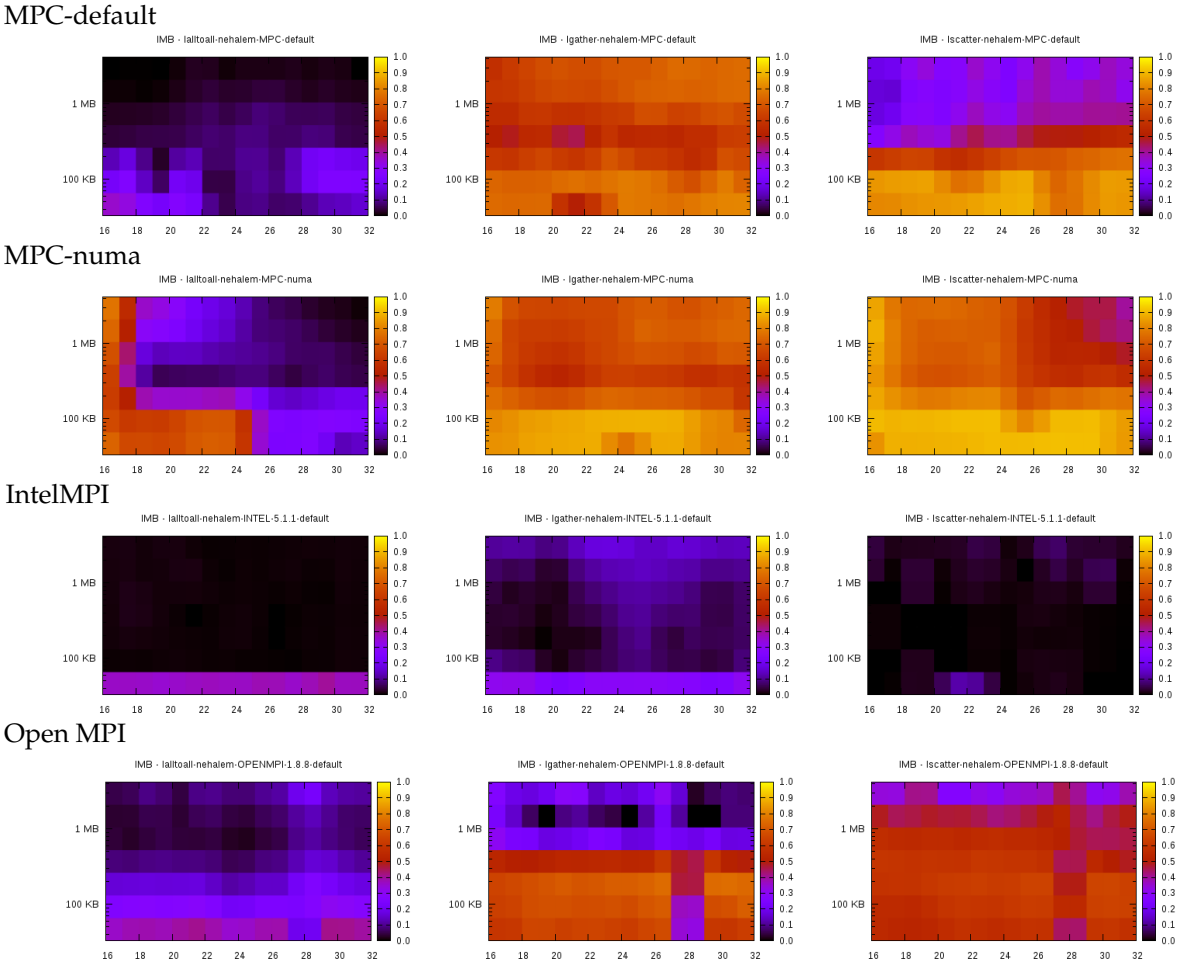
(a) Default (b) Numa

FIGURE 2 – Placement des rangs MPI en noir et des threads de progression en bleu sur un hôte de 8 cœurs avec 2 nœuds NUMA en fonction du nombre de tâches MPI.

le second nœud NUMA, un cœur est libre alors tous les threads de progression de ce nœud NUMA vont y être liés. Avec notre méthode, un thread de progression ne peut jamais être lié sur un nœud NUMA différent de la tâche MPI qui l'a généré.

4. Résultats

Afin de mesurer l'impact du placement des threads de progression, nous utilisons les Intel MPI Benchmarks (IMB) NBC [1] conçus pour mesurer le taux de recouvrement des communications collectives non-bloquantes. Nous avons lancé la suite de benchmarks sur 4 hôtes possédant 2 processeurs quad-cores Intel Xeon X5560 à 2.80 GHz qui constituent chacun 2 nœuds NUMA avec un réseau InfiniBand. Nous testons toutes les collectives MPI non-bloquantes avec la configuration par défaut de MPC que nous nommerons « default » ainsi qu'avec la configuration que nous avons mis en place pour améliorer le recouvrement des communications que nous nommerons « numa ». Pour chaque collective, nous faisons varier le nombre de tâches impliquées dans celle-ci ainsi que la taille des données à utiliser lors de ces communications.
Le temps de calcul est du même ordre de grandeur que le temps des communications lors de l'utilisation des IMBs. Ceci est dû à l'utilisation de la fonction IMB_cpu_exploit [2] qui génère un temps de calcul proche du temps de communication. Plusieurs temps sont mesurés dans ce benchmark. Le temps de communication seul, noté « t_pure », le temps de calcul seul, noté « t_CPU » et le temps de la communication en concurrence avec un temps de calcul généré par la fonction IMB_cpu_exploit, noté « t_ovrl ». Le taux de recouvrement calculé dans les IMBs est donné par la formule suivante : $taux = max(0, min(1, (t_pure + t_CPU - t_ovrl)/min(t_pure, t_CPU))$.
Un taux de 1 correspond à un taux de recouvrement de 100%. À l'inverse, un taux de 0 signifie que nous n'avons aucun recouvrement.
Les résultats des tests que nous avons effectués démontrent l'importance du placement des threads de progression. La figure 3 illustre les résultats des tests sur les collectives non-bloquantes Igather (tous vers 1), Iscatter (1 vers tous) et Ialltoall (tous vers tous) avec MPC, IntelMPI et Open MPI. Pour chaque collectives, nous avons le nombre de tâches MPI en abscisse et la taille des données (en octets) utilisée pour les communications en ordonnée. La couleur représente le taux de recouvrement des communications. La couleur jaune représente un taux de recouvrement de 100% tandis que le noir correspond à aucun recouvrement.



(a) Ialltoall (b) Igather (c) Iscatter

FIGURE 3 – Taux de recouvrement des collectives Ialltoall, Igather et Iscatter sur 32 cœurs en fonction du nombre de tâches MPI et de la taille des messages en octets sur 4 hôtes de 8 cœurs

Sur les résultats correspondant à « MPC-default » et « MPC-numa », le seuil de 24 tâches correspond à un cœur de libre pour chaque nœud NUMA dans notre cas test. Le calcul n'est donc jamais perturbé par les communications. Au-delà de 24 tâches, les threads de progression sont liés sur le même cœur que les tâches MPI qui les ont générés s'il n'y a plus de cœurs de libres au sein de leur nœud NUMA.

Les seuils observés lorsque nous dépassons une taille 128 kio pour MPC et 64 kio pour IntelMPI correspondent à un changement d'algorithme. C'est le moment où est mis en œuvre le protocole de « rendez-vous ». Au-delà de ce seuil nous avons un taux de recouvrement inférieur. Néanmoins, lorsque nous avons un cœur dédié à chaque thread de progression, nous pouvons observer un très bon taux de recouvrement malgré le changement d'algorithme pour le cas « MPC-numa ». De plus, le taux de recouvrement est de l'ordre de 80% lorsque nous avons des ressources dédiées aux threads de progression et que nous sommes avec des tailles de messages inférieures au seuil de changement de protocole.

Nous avons aussi comparé nos résultats avec d'autres bibliothèques MPI. Les résultats correspondant à « IntelMPI » et « Open MPI » sur la figure 3 illustrent ces résultats. Nous observons que notre méthode offre un meilleur taux de recouvrement que les bibliothèques IntelMPI et

Open MPI. Il existe certains cas où IntelMPI obtient de meilleurs temps d'exécution que MPC-numa. Néanmoins, nous pensons que notre approche pourrait améliorer le taux de recouvrement d'autres implémentations. Cela pourrait induire un gain de performance.

5. Travaux connexes

Plusieurs travaux portant sur le recouvrement des communications ont été réalisés. Ils concernent principalement le recouvrement des communications point-à-point. Dans [13] et [12], les auteurs proposent des algorithmes se basant sur l'utilisation d'une technologie matérielle appelée « Remote Direct Memory Access » (RDMA) qui permet à une carte réseau d'envoyer directement un message sur une carte distante sans l'intervention du processeur. Ces travaux reposent sur des technologies matérielles qui ne sont pas généralisables pour les opérations collectives non-bloquantes actuellement.
Des travaux portant sur les opérations collectives existent. Dans [4], les auteurs présentent des optimisations des collectives non-bloquantes mais qui sont liées à leur machine Blue Gene.
D'autres travaux reposant sur des techniques logicielles permettent une meilleure progression des communications. Dans [7], l'auteur présente pioman. C'est un logiciel voué à être utilisé par les bibliothèques MPI pour effectuer les tâches de communications. Aucune implémentation pour les collectives MPI non-bloquantes n'existe. Cependant, l'approche de pioman est très similaire à la nôtre.
Dans [10], l'auteur présente une implémentation des collectives non-bloquantes. C'est cette implementation qui est intégrée au support exécutif MPC [11]. Dans [9], il démontre l'utilité des threads de progression pour effectuer les communications non-bloquantes. Nous proposons une gestion efficace de ces threads dans cet article.

6. Conclusion

Nous avons présenté une solution permettant de manipuler un des leviers pour avoir de bon taux de recouvrement. Nous avons proposé un placement des threads de progression tenant compte des effets NUMA et qui améliore le taux de recouvrement des communications liées au collectives MPI non-bloquantes. Nous avons comparé les taux de recouvrement de plusieurs bibliothèques MPI entre elles afin de savoir lesquelles proposaient des taux de recouvrement élevés.
Dans cet article, nous nous sommes concentrés sur le placement des threads de progression pour améliorer le taux de recouvrement des collectives MPI non-bloquantes. Un deuxième levier est d'ordonnancer les threads plus efficacement. Nos travaux à venir sont d'implémenter un mécanisme permettant d'attribuer des types à des threads en fonction de leur nature (tâche MPI, thread OpenMP, thread de progression, etc.) puis de mettre en place un algorithme d'ordonnancement de thread à priorité dynamique en fonction de ces types.

7. Remerciements

Ce document est réalisé dans le cadre du projet ELCI, un projet collaboratif Français FSN (« Fond pour la Société Numérique »), qui associe des partenaires académiques et industriels pour concevoir et fournir un environnement logiciel pour le calcul intensif.

Bibliographie

1. IMB-NBC benchmarks. – `https://software.intel.com/fr-fr/node/561946`. Accessed : 2016-02-23.
2. Measuring Communication and Computation Overlap. – `https://software.intel.com/fr-fr/node/561947`. Accessed : 2016-02-23.
3. MPI : A Message-Passing Interface Standard Version 3.0. – 2012.
4. Almási (G.), Heidelberger (P.), Archer (C. J.), Martorell (X.), Erway (C. C.), Moreira (J. E.), Steinmacher-Burow (B.) et Zheng (Y.). – Optimization of mpi collective communication on bluegene/l systems. – In *Proceedings of the 19th Annual International Conference on Supercomputing*, *ICS '05*, ICS '05, pp. 253–262, New York, NY, USA, 2005. ACM.
5. 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.
6. Carribault (P.), Pérache (M.) et Jourdren (H.). – *Beyond Loop Level Parallelism in OpenMP : Accelerators, Tasking and More : 6th Internationan Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010 Proceedings*, chap. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC, pp. 1–14. – Berlin, Heidelberg, Springer Berlin Heidelberg, 2010.
7. Denis (A.). – pioman : a Generic Framework for Asynchronous Progression and Multithreaded Communications. – In *IEEE International Conference on Cluster Computing (IEEE Cluster)*, Madrid, Espagne, septembre 2014.
8. Derradji (S.), Palfer-Sollier (T.), Panziera (J.-P.), Poudes (A.) et Wellenreiter (F.). – The BXI Interconnect architecture. – In *High-Performance Inter-connects (HOTI)*. 2015 IEEE 23th Annual Symposium, 2015.
9. Hoefler (T.) et Lumsdaine (A.). – Message Progression in Parallel Computing - To Thread or not to Thread ? – In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.
10. Hoefler (T.), Lumsdaine (A.) et Rehm (W.). – Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. – In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.
11. Pérache (M.), Jourdren (H.) et Namyst (R.). – MPC : A Unified Parallel Runtime for Clusters of NUMA Machines. – In Springer (édité par), *the 14th International Euro-Par Conference*, *LNCS*, volume 5168, pp. 78–88, Las Palmas de Gran Canaria, Spain, août 2008.
12. Rashti (M. J.) et Afsahi (A.). – Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects. – In *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*, pp. 95–101. IEEE, 2008.
13. Sur (S.), Jin (H.), Chai (L.) et Panda (D.). – RDMA read based rendezvous protocol for MPI over InfiniBand : design alternatives and benefits. – In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 32–39. ACM New York, NY, USA, 2006.