Recouvrement des Collectives MPI Non-bloquantes sur Processeur Manycore

Hugo Taboada

Compas'2018 : Parallélisme / Architecture / Système
Toulouse, France, du 3 au 6 juillet 2018

Recouvrement des Collectives MPI Non-bloquantes sur Processeur Manycore

Hugo Taboada

CEA, DAM, DIF, F-91297 Arpajon, France
Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP, France
hugo.taboada@inria.fr

Résumé
Les collectives MPI non-bloquantes ont été proposées pour recouvrir les communications par du calcul afin d'en amortir le coût. Cependant, ces opérations consomment plus de temps CPU que les opérations point-à-point. L'utilisation d'un seul CPU dédié aux threads de progression n'est donc pas efficace et rend les communications lentes. D'un autre côté, si les communications sont exécutées sur les cœurs applicatifs, aucun recouvrement n'est obtenu. Pour aborder ce problème, nous proposons un algorithme pour les opérations collectives en arbre qui scinde l'arbre des communications entre les cœurs applicatifs et les cœurs dédiés aux communications afin d'obtenir un compromis entre le taux de recouvrement et les performances globales. Nous proposons un modèle afin d'étudier et prédire le comportement de cet algorithme ainsi qu'une implémentation dans le framework MPC. Nous avons obtenu de bons résultats en testant notre approche sur des processeurs manycores tels que le KNL et le Skylake.

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

---

1. Introduction

Message Passing Interface (MPI) est l'interface standard utilisée pour effectuer des communications dans les applications HPC. Le coût des communications est un problème majeur pour avoir une bonne performance dans les applications MPI. Pour amortir leur coût, les développeurs d'applications essaient de recouvrir les communications par du calcul en utilisant les communications MPI non-bloquantes, en espérant que celles-ci s'effectuent en arrière-plan pendant que le calcul s'exécute.
Initiallement, les communications non-bloquantes n'étaient disponible que pour les communications point-à-point. L'extension des communications non-bloquantes aux opérations collectives (c.-à-d. des primitives qui impliquent plus de deux rangs MPI, tels que broadcast, reduce, scatter, gather, ...) est apparue dans la dernière version majeure de la norme MPI [9]. Cela a permis le recouvrement des communications collectives par du calcul. Cependant, les communications collectives sont plus gourmandes sur l'utilisation du CPU que les communications point-à-point, et sont donc plus difficiles à faire progresser en arrière-plan.
Dans cet article, nous nous concentrons sur le problème du recouvrement des communications par du calcul pour les collectives non-bloquantes sur les processeurs manycore. Nous étudions le compromis entre exécuter les communications collectives sur des cœurs dédiés et utiliser les

cœurs applicatifs. Nous nous restreignons au cas des opérations collectives basées sur un arbre de communication car c'est le cas où ce compromis a le plus d'impact sur les performances.
Pour résumer, les contributions sont :

- Proposer un *algorithme* qui scinde l'arbre des opérations collectives en exécutant une partie de cet arbre sur les cœurs dédiés aux communications et une autre sur les cœurs applicatifs.
- Proposer un *modèle* pour l'algorithme précédent, afin de démontrer l'amélioration des performances globales quand les communications sont recouvertes par du calcul et pour ajuster ses paramètres.
- Implémenter l'algorithme dans l'implémentation MPI de MPC (Multi-processor Computing) [10].

Le reste du papier est organisé de la façon suivante. La Section 2 présente les travaux connexes sur le recouvrement du calcul/communications en général et sur les opérations collectives en particulier. La Section 3 présente notre algorithme « split-tree » pour les communications collectives en arbre. Dans la Section 4, nous présentons le modèle de l'algorithme et comment nous le paramétrons pour avoir les performances optimales. La Section 5 présente les résultats expérimentaux et la Section 6 conclut.

2. Travaux Connexes

Plusieurs stratégies existent pour faire progresser les communications point-à-point en arrière-plan, telles que transférer la communication au matériel [13, 11] et le laisser faire la progression, l'utilisation d'un thread [4] ou d'un processus [7] dédié à la progression des communications, ou encore l'ordonnancement opportuniste de tâches de communication [3, 12].
Les communications MPI non-bloquantes sont plus difficiles à faire progresser en arrière-plan. Elles requièrent non seulement de faire progresser le transfert des données mais aussi l'algorithme en lui-même. Cela rend la progression plus complexe avec l'utilisation seule du matériel. Des travaux spécifiques en rapport avec l'utilisation de la progression assistée par le matériel sur Blue Gene [2] ainsi que le transfert des collectives en mémoire partagée à un module noyau [8] existent (bien que les auteurs ont seulement traité les performances obtenues sur les opérations collectives bloquantes et pas la progression des collectives non-bloquantes). La libNBC [6], implémentation de référence des collectives non-bloquantes, utilise un thread de progression, avec certaines améliorations [5] pour augmenter le taux de recouvrement sur InfiniBand.

3. L'algorithme « split-tree » pour les collectives MPI non-bloquantes

Dans cet article, nous nous concentrons sur les communications intra-noeud sur une machine manycore, avec une tâche MPI par cœur. Pour obtenir un recouvrement calcul/communications efficace, celles-ci doivent s'exécuter en parallèle. Sur une machine manycore, une façon simple de faire progresser les communications en arrière-plan est de dédier certains cœurs aux communications. Ainsi, quelques cœurs exécutent les tâches MPI (appelés par la suite *cœurs applicatifs* ou *CA*), tandis que les cœurs restant seront réservés pour les threads de progression (appelés par la suite *cœurs de communication* ou *CC*).
Cependant, les algorithmes de communications collectives nécessitent une quantité élevée de communications point-à-point. Quand les *CC* exécutent toutes les communications au nom de toutes les tâches MPI s'exécutant sur les *CA*, l'algorithme est *replié* et les communications provenant d'une étape donnée de l'algorithme peuvent être sérialisées. Par conséquent, quand

nous les replions sur peu de *CC*, les communications collectives s'exécutent plus lentement que lorsqu'elles sont exécutées comme une collective bloquante sur tous les *CA*.
Nous nous sommes restreints aux collectives utilisant un algorithme *en arbre*.Les étapes d'une collective basée sur des communications en arbre binomial sont représentées sur la Figure 1. Chaque niveau de l'arbre est une étape de l'algorithme, des feuilles vers la racine. Le rang de chaque tâche MPI est représenté par les sommets. Les sommets reliés par une arête en pointillés représentent les mêmes tâches MPI. Seuls les arcs pleins impliquent une communication.
Sur les algorithmes basés sur une topologie en arbre, nous observons que le nombre de communications est très mal équilibré. Sur l'exemple présenté en Figure 1, pour 16 tâches MPI, nous avons 15 communications réparties sur 4 étapes. Si nous replions ces communications sur un seul cœur de communication, nous aurons 15 étapes, ce qui est 4 fois plus lent. Une partie importante des communications s'effectue à la première étape, représenté par $S = 1$ avec les étapes numérotées depuis les feuilles. S'il n'y a que les communications de la première étape qui s'exécutent sur les *CA*, et que le reste des communications s'exécutent sur les *CC*, le nombre d'étapes nécessaires est 2 fois moins élevé que de tout exécuter sur les *CC* alors qu'une étape de l'algorithme ne peut pas être recouverte par du calcul.
L'algorithme que nous proposons est une généralisation de ce principe pour avoir un *compromis entre les performances des communications et le taux de recouvrement* : scinder l'arbre de communication avec une partie s'exécutant sur tous les *CA* pour bénéficier du parallélisme et une autre partie sur les *CC* pour bénéficier du recouvrement des communications.
Soit *S* le nombre d'étapes (étages de l'arbre) s'exécutant sur les *CA*. $S = 0$ est équivalent à exécuter toutes les communications sur les *CC*. L'algorithme exécute *S* étapes de l'arbre sur les *CA* comme décrit dans la Figure 1. Quand $S = 1$, l'algorithme exécute la plus petite partie en nombre d'étapes mais la plus lourde en communications sur les *CA* tandis que la partie la plus longue en nombre d'étapes mais la plus légère en communications est exécutée sur un ou plusieurs *CC*. Toutes les communications qui sont exécutées sur les *CA* ne bénéficient pas du recouvrement par du calcul car elles sont exécutées sur le même cœur. Cependant, cette partie de l'arbre est celle qui concentre le plus de communications et l'exécuter sur un petit nombre de *CC* compromettrait les performances des communications. La partie s'exécutant sur les *CC* bénéficie d'un recouvrement total de ses communications.
Si *S* augmente, l'algorithme perd de sa capacité à recouvrir les communications mais peut augmenter les performances globales de l'application, si le ratio calcul/communications le permet.

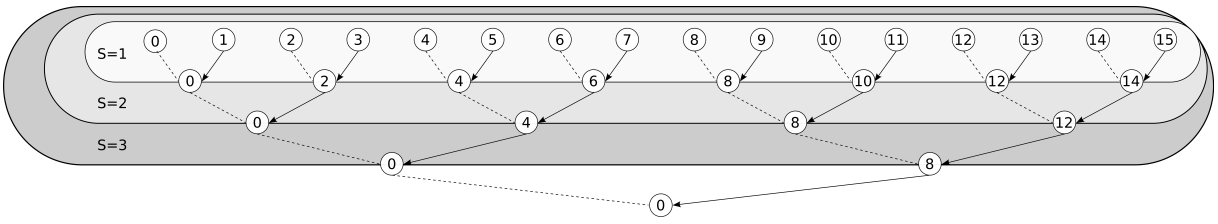

FIGURE 1 – L'arbre de communication pour la collective reduce avec 16 rangs MPI. *S* est le nombre d'étapes (niveau de profondeur de l'arbre) qui s'exécute sur les cœurs applicatifs. Les arcs pleins représentent les communications. Les sommets représentent les rangs MPI.

4. Modélisation

Dans cette section, nous proposons un modèle de performance de l'algorithme décrit dans la Section 3, afin de montrer sa pertinence et d'ajuster ses paramètres.

Modèle pour les opérations collectives

Soit N_{proc} le nombre total de cœurs et N le nombre de *CA* (c.-à-d. nombre de rangs MPI), alors le nombre de cœurs dédiés aux communications est $P(N) = N_{proc} - N$.
Nous considérons seulement les opérations collectives en arbre. Nous modélisons le coût des communications comme étant linéaire, car avec des tailles de messages suffisantes pour que le recouvrement ait un sens en intra-noeud, la latence et les effets de cache sont négligeables. Nous prenons comme unité le temps de transfert d'un tampon de la taille considérée dans l'opération collective pour une opération point-à-point. Nous avons d'abord étudié les opérations avec une taille de tampon constante à travers tout l'arbre de communication (reduce, broadcast).
La profondeur de l'arbre est $H(N) = \lceil \log_2(N) \rceil$. Dans le cas d'une opération bloquante où les communications sont exécutées simultanément par tous les *CA*, nous obtenons le temps d'exécution suivant : $T_{blocking}(N) = H(N) = \lceil \log_2(N) \rceil$.
Soit $C(N)$ le temps de calcul sur N cœurs. Pour modéliser le calcul et le recouvrement des communications, nous considérons que le programmeur de l'application a essayé d'obtenir un recouvrement parfait des communications, et que la charge de calcul est suffisante pour que le temps du calcul soit équivalent au temps d'une opération collective bloquante sur tous les cœurs, c.-à-d. $C(N_{proc}) = T_{blocking}(N_{proc})$. Sous l'hypothèse d'une charge de calcul ayant une accélération linéaire, le temps de calcul sur N cœurs est : $C(N) = \frac{N}{N_{proc}} \times C(N_{proc})$.

Modèle pour l'algorithme proposé

Modélisons l'algorithme « split-tree » en lui-même. Comme défini dans la Section 3, S est le nombre d'étapes s'exécutant sur les *CA*. Le temps nécessaire à exécuter ces étapes correspond à S. Ensuite, l'algorithme ordonnance les opérations de communication des $H(N) - S$ dernières étapes repliées sur les $P(N)$ *CC*. Soit $R(N) = N - 2^{\lfloor \log_2(N) \rfloor}$ le nombre de feuilles qui ne sont pas dans le plus grand sous-arbre binaire complet de l'arbre. Soit $F(N, i)$ le nombre de communications pour N tâches MPI pour l'étape i avec i compris entre 1 et $H(N)$:

$$F(N, i) = 2^{\lfloor \log_2(N) \rfloor - i} + \left\lfloor \frac{R(N) + 2^{(i-1)}}{2^i} \right\rfloor \tag{1}$$

Les étapes sont déroulées les unes après les autres pour respecter les dépendances entre les communications. Puisque chaque étape i contient $F(N, i)$ communications, le temps de communication prend $\lceil F(N, i)/P(N) \rceil$ quand l'étape est repliée sur $P(N)$ *CC*. Le temps d'exécution d'une collective non-bloquante avec notre algorithme qui scinde l'arbre des communications est décrit par l'Équation 2 suivante :

$$T_{non-blocking}(S, N) = \underbrace{min(S, H(N))}_{\text{premières } S \text{ étapes de l'arbre}} + \underbrace{\sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N, i)}{P(N)} \right\rceil}_{\text{dernières } H(N)-S \text{ étapes restantes de l'arbre}} \tag{2}$$

Le recouvrement des communications par du calcul provenant d'une collective non-bloquante est effectué seulement sur la partie s'exécutant sur les *CC*. La partie des communications s'exé-

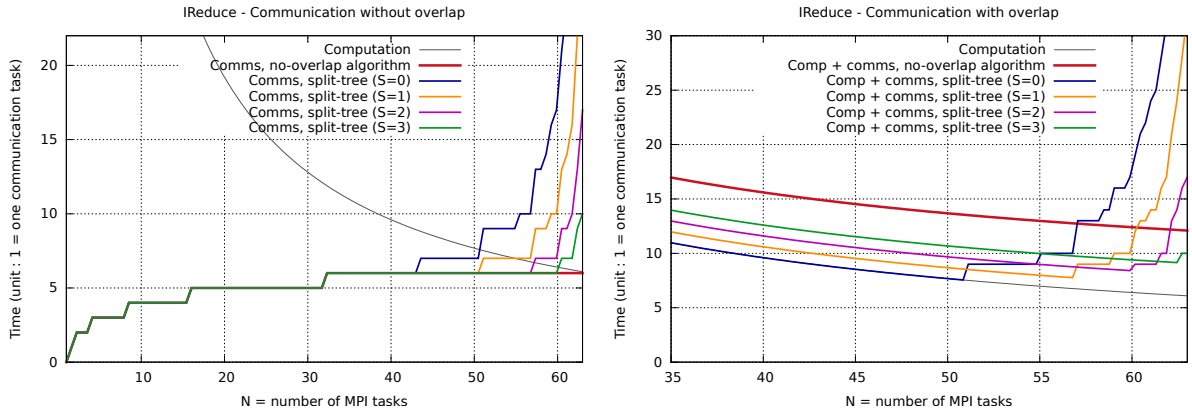

FIGURE 2 – Modèle de coût des communications (gauche) et du recouvrement calcul/communications (droite) avec un tampon de taille constante (reduce, bcast) sur 64 cœurs.

cutant sur les *CA* n'est pas recouverte. Le temps d'exécution est donné par l'Équation 3 suivante :

$$T_{overlapped}(S,N) = \underbrace{\min(S,H(N))}_{\text{communications non-recouvertes}} + \underbrace{\max\left(C(N), \sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil\right)}_{\text{communications recouvertes}} \quad (3)$$

Les courbes $C(N)$, $T_{blocking}$, et $T_{non-blocking}(N,S)$ pour différentes valeurs de S avec $N_{proc} = 64$ sont représentées sur la Figure 2 (gauche). Nous observons que pour des grandes valeurs de N, le temps estimé est très élevé pour $S = 0$ car toutes les communications sont réalisées sur peu de *CC*. Le coût de ces communications se réduit quand S augmente.
À droite de la Figure 2, le temps total du recouvrement calcul/communications est représenté dans le cas où le calcul et les communications s'effectuent sans aucun recouvrement, puis dans le cas où nous utilisons notre algorithme avec différentes valeurs de S. Nous observons que, pour les petites valeurs de N, augmenter la valeur de S augmente le coût car le taux de recouvrement est réduit. Néanmoins, ce coût est amorti pour les grandes valeurs de N où le temps total est dominé par le coût des communications repliées sur peu de *CC*.
Nous pouvons étendre le modèle proposé aux opérations collectives avec un poids non constant sur les arêtes, telles que gather et scatter. Le poids des arêtes double à chaque étage de l'arbre en partant des feuilles vers la racine.

5. Résultats expérimentaux

Dans cette section, nous présentons les résultats expérimentaux de notre algorithme implémenté au sein du framework MPC [10], fournissant les threads de progression en intégrant la libNBC [6]. Nous avons implémenté notre propre suite de tests afin d'évaluer la performance de notre algorithme. Celle-ci est très similaire aux Intel MPI Benchmarks [1] mais avec une taille de problème fixe. Cela nous permet d'avoir la même charge de travail globale pour un nombre différent de tâches MPI. Les cœurs inutilisés sont alors dédiés aux threads de progression, afin de maximiser le taux de recouvrement. Un Intel Xeon Phi Knights Landing à 1.4GHz avec 64 cœurs (KNL) et un bi-socket Intel Xeon Platinum Skylake à 2.7GHz avec 48 cœurs ont

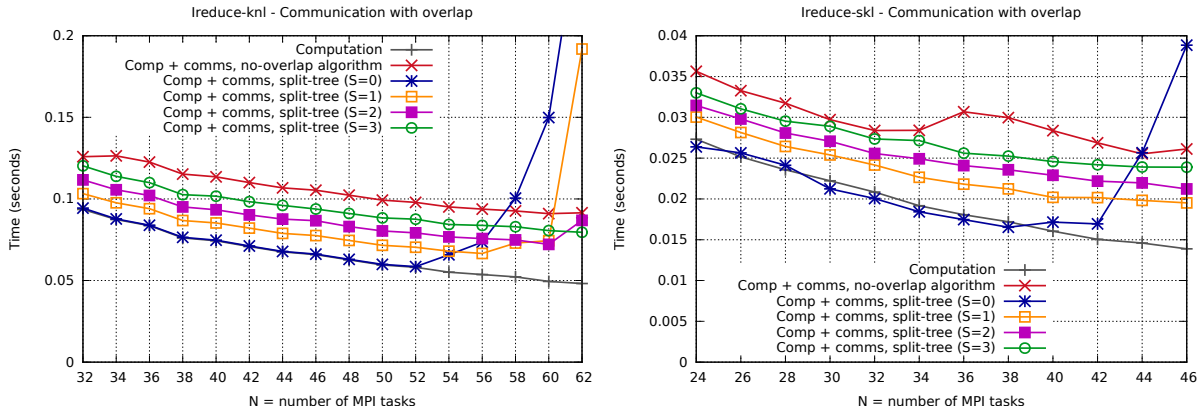

FIGURE 3 – Résultat pour l'algorithme « split-tree » avec différentes valeurs de S, pour MPI_Ireduce avec une taille de tampon constante de 2MB sur KNL (gauche) et Skylake (droite).

été utilisés pour nos tests.

Comparaison de l'algorithme « split-tree » au paramètre par défaut.

Replier les threads de progression sur des cœurs dédiés apporte de bonnes performances quand le nombre de cœurs dédiés est important. Cependant, les performances se dégradent quand trop de threads de progression sont repliés sur le même cœur. Ce comportement est illustré sur la courbe bleue de la Figure 3, où toutes les communications sont exécutées sur les *CC* pour la collective Ireduce.
Grâce à l'algorithme « split-tree », nous sommes capables d'équilibrer les communications entre les *CA* et les *CC*. Les courbes orange, violette et verte montrent les performances d'une même collective quand 1, 2 ou 3 étapes de l'arbre de communications sont effectuées sur les *CA*.
Le modèle nous permet de sélectionner automatiquement la meilleure valeur de S en tenant compte de la collective et du nombre de cœurs dédiés aux threads de progression.

Comparaison des implémentations MPI.

Nous avons comparé notre algorithme « split-tree », avec les valeurs de S données par notre modèle, à d'autres implémentations MPI telles qu'Intel-MPI et OpenMPI. Sur KNL, nous changeons de $S = 0$ à $S = 1$ pour 52 tâches MPI, de $S = 1$ à $S = 2$ pour 58 tâches MPI et de $S = 2$ à $S = 3$ pour 62 tâches MPI (respectivement 12, 6 et 2 cœurs de libres).
Les résultats des autres implémentations MPI testées sont représentées sur la Figure 4.
Par soucis d'équité, nous avons activé les flags permettant d'avoir la progression asynchrone pour Intel-MPI, mais ces flags réduisent les performances au lieu de les améliorer. Nous observons que notre algorithme (MPC model-based – courbe verte) est toujours meilleur que celui d'OpenMPI et d'Intel-MPI.

6. Conclusion

Pour recouvrir le coût des communications collectives par du calcul, les approches consistant à utiliser un thread de progression par tâche souffrent de la compétition entre les communications et le calcul. Celles qui consistent à allouer des cœurs dédiés aux communications souffrent d'une baisse des performances du temps de communication quand une collective est repliée sur

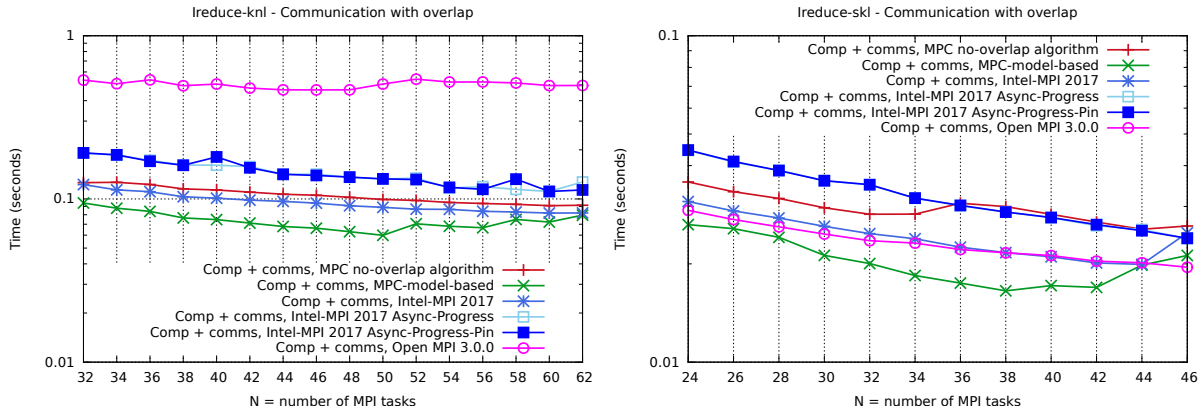

FIGURE 4 – Résultat pour plusieurs implémentations MPI pour MPI_Ireduce avec une taille de tampon constante de 2MB sur KNL (gauche) et Skylake (droite).

peu de cœurs dédiés.
Dans cet article, nous avons proposé un nouvel algorithme qui combine le meilleur des deux mondes. Il scinde l'arbre des communications et exécute la partie la plus lourde en communications sur les *CA* pour bénéficier de plus de parallélisme tandis que la partie la plus légère en communications mais occupant le plus grand nombre d'étapes est exécutée sur les *CC* afin de les recouvrir par du calcul. Nous avons modélisé l'algorithme pour démontrer sa pertinence et affiner ses paramètres. Nous l'avons implémenté dans MPC et évalué ses performances sur des processeurs manycores (Intel KNL et Skylake). Grâce à la précision de notre modèle, nous sommes capables de presque toujours trouver le meilleur compromis entre utiliser les cœurs dédiés ou les *CA* et par conséquent d'avoir une meilleure performance que d'autre implémentation MPI de l'état de l'art. Notre solution peut être implémentée sur n'importe quelle implémentation MPI avec des threads de progression.
Les travaux à venir visent à étendre l'approche de notre algorithme à des communications inter-noeuds, ayant un comportement différent des communications intra-noeuds considérées dans cet article. Nous pensons aussi améliorer l'auto-tuning pour choisir le nombre de tâches MPI (paramètre N) pour optimiser les performances globales d'une application et pas seulement des sections avec une collective non-bloquante.

Bibliographie

1. IMB-NBC benchmarks. – `https://software.intel.com/fr-fr/node/561946`. Accessed : 2018-03-01.
2. 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.
3. Denis (A.). – pioman : a pthread-based Multithreaded Communication Engine. – In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, mars 2015.
4. 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 Com-*

puting. IEEE Computer Society, Oct. 2008.

5. Hoefler (T.) et Lumsdaine (A.). – Optimizing non-blocking Collective Operations for InfiniBand. – In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC'08 Workshop*, Apr. 2008.
6. 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.
7. Lai (P.), Balaji (P.), Thakur (R.) et Panda (D.). – Proone : A general purpose protocol onload engine for multi- and many-core architectures. – June 2009.
8. Ma (T.), Bosilca (G.), Bouteiller (A.), Goglin (B.), Squyres (J. M.) et Dongarra (J. J.). – Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs. – In IEEE (édité par), *40th International Conference on Parallel Processing (ICPP-2011)*, Taipei, Taiwan, septembre 2011.
9. MPI Forum. – MPI : A Message-Passing Interface Standard Version 3.0, septembre 2012.
10. 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.
11. 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.
12. Si (M.), Peña (A.), Balaji (P.), Takagi (M.) et Ishikawa (Y.). – Mt-mpi : multithreaded mpi for many-core environments. – In *Proceedings of the International Conference on Supercomputing*, 06 2014.
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.