

Utilisation de cœurs dédiés pour la progression des communications collectives non bloquantes

Florian Reynier

▶ **To cite this version:**

Florian Reynier. Utilisation de cœurs dédiés pour la progression des communications collectives non bloquantes. COMPAS 2020 - Conférence francophone d'informatique en Parallélisme, Architecture et Système, Jun 2020, Lyon, France. hal-03118807

HAL Id: hal-03118807
https://inria.hal.science/hal-03118807

Submitted on 22 Jan 2021

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

Utilisation de cœurs dédiés pour la progression des communications collectives non bloquantes

Florian Reynier *

CEA, DAM, DIF, F-91297 Arpajon, France / Inria Bordeaux - Sud-Ouest, France

Résumé
Dans MPI, il est particulièrement ardu de recouvrir correctement les collectives non bloquantes. Quand ils existent, les mécanismes proposant un recouvrement correct comme les threads de progression sont particulièrement pénalisants pour les applications. En effet, la cohabitation de threads applicatifs et MPI est un problème complexe à gérer. Dans cet article, nous proposons de mesurer et analyser l'attribution d'un cœur dédié à la progression des communications non bloquantes. Nous mesurerons les gains induits par l'utilisation d'un cœur dédié, recouvrant les pertes raisonnables du retrait d'un cœur à l'application.

Mots-clés : Collectives non bloquantes, cœur dédié, MPI, Mesure des collectives

1. Contexte

Dans le monde du calcul haute performance (HPC), les applications s'exécutent sur de nombreux nœuds de calculs ce qui nécessite des communications, le plus souvent réalisées avec "Message Passing Interface" (MPI). Ces communications peuvent prendre beaucoup de temps. Pour limiter leur impact sur les performances, il peut être intéressant de les recouvrir, c'est à dire continuer les calculs de l'application pendant le déroulement des communications. Depuis sa version 3.0 [6], MPI propose des collectives non bloquantes (NBC) permettant une telle fonctionnalité. Cependant, réussir à faire progresser les communications en arrière plan sans avoir d'impact sur le calcul est difficile à mettre en place.
Dans cet article nous proposons de prendre un cœur de calcul pour le dédier à la progression des communications MPI. Nous étudierons ce partage des ressources dans un contexte hybride en utilisant le modèle de programmation OpenMP pour la parallélisation intra-nœud, associé à MPI. Nous avons mis en œuvre cette solution au sein de la bibliothèque NewMadeleine [2].
Cet article présentera d'abord une analyse des performances actuelles de bibliothèques MPI, puis la solution à base de cœurs dédiés. Nous analyserons les résultats obtenus avant de conclure.

2. Travaux connexes

La progression des communications collectives non bloquantes a déjà été longuement étudiée.Il existe différentes manières de réaliser cette progression comme l'exécution opportuniste de tâches de communication [4][13], la progression matérielle [14][12], ou enfin les threads de

*. Remerciement à Alexandre Denis, Julien Jaeger et Emmanuel Jeannot pour leurs conseils et relectures

progression [8]. Ce mécanisme, en particulier, a introduit diverses problématiques liées au placement et aux interactions MPI/application [15].
La mesure des NBC est aussi une nécessité pour évaluer les mécanismes proposés. Mesurer des communications collectives impliquant plusieurs nœuds pose des problèmes de mesure de temps, chaque nœud possédant sa propre horloge [9].

3. Attribution de ressources à la progression

3.1. Besoin du processeur pour la progression

Depuis MPI 1.0, il existe des mécanismes de progression pour les communications point-à-point non bloquantes. La progression mise en place pour les communications collectives non bloquantes repose sur les mêmes mécanismes. Cependant les communications collectives engendrent plus de messages et nécessitent régulièrement du temps processeur pour progresser. Simultanément, l'application continue son exécution. Les deux ont donc besoin de temps processeur au même moment. Cela peut provoquer des interactions et dégrader les performances.

3.2. Dédier un cœur en contexte hybride

Dans un contexte hybride MPI + OpenMP, il est courant d'avoir un rang MPI par nœud ou par banc NUMA, et d'occuper tout les cœurs avec des threads OpenMP pour bénéficier de la mémoire partagée. La progression MPI risque donc d'interagir avec un thread OpenMP.
Nous proposons donc de réduire le nombre de threads OpenMP pour libérer des ressources à affecter à la progression des communications. Pour un nœud à n cœurs, cela consiste à créer seulement $n-1$ threads OpenMP au lieu de n.
Il est alors possible d'estimer les pertes induites par cette solution en se basant sur la loi d'Amdahl [1]. Si l'on représente un temps b, durée séquentielle de la partie parallélisée de l'application, et a, durée de la partie non parallélisable, une parallélisation parfaite donnerait un temps $\frac{b}{n} + a$. Le temps de la partie parallèle pénalisée d'un cœur de calcul serait $\frac{b}{n-1} + a$. Le temps de calcul perdu entre l'utilisation de n cœurs et de $n-1$ cœurs pour le calcul est donc de : $\frac{-b}{n(n-1)}$.
Ce temps diminue quadratiquement en fonction du nombre total de cœurs. Plus le nombre de cœurs est grand, plus il devient "négligeable" de voler un cœur pour faire de la progression.

3.3. Etude sur le passage à l'échelle des applications

La validité de la méthode dépend de la capacité du recouvrement des communications à absorber cette perte de temps. Il est donc nécessaire que l'application ait une scalabilité linéaire pour limiter, et pouvoir évaluer, le temps perdu dû au retrait d'un cœur.
Nous souhaitons évaluer notre méthode sur différents benchmarks, tel que Lulesh des benchmarks Coral [10], miniMD des benchmarks Mantevo [7], et BT-MZ des benchmarks NAS [3].
Avant de tester l'apport de notre méthode sur la progression, nous commençons par évaluer la scalabilité de ces applications. Nous exécutons ces applications sur un calculateur composé de nœuds Sandy Bridge de 16 cœurs, en utilisant 8 nœuds pour Lulesh et MiniMD et 16 nœuds pour BT-MZ. Les tests s'exécutent avec un processus MPI par nœud, et nous faisons varier le nombre de threads OpenMP de 1 à 16 (nœud complet). Les résultats de scalabilité en fonction du nombre de thread OpenMP sont présentés en figure 1.
On observe des comportements différents pour les 3 applications. Pour Lulesh, à gauche, on peut observer un ralentissement en passant de 8 à 9 threads. cela est dû au fait que l'on commence à utiliser le deuxième nœuds NUMA, ce qui provoque des effets NUMA pour cette application. On observe un résultat plus linéaire sur la deuxième graphe avec l'application miniMD malgré une perte de temps à 16 threads. Enfin, l'application BT-MZ montre la meilleure

scalabilité, le temps séquentiel est quasiment divisé par le nombre de cœurs ce qui représente une parallélisation quasi parfaite. C'est typiquement dans ce cas de figure que retirer un cœur est le plus pénalisant car le gain de $n-1$ à n cœurs est optimal.
Sur cette application, on observe analytiquement que sa parallélisation suit le modèle présenté en Section 3.2, avec $a = 2s$ et $b = 58s$. Ainsi, pour $n = 15$, nous obtenons un temps de $2 + 58/15 = 5,87s$, et pour $n = 16$, un temps de $2 + 58/16 = 5,56s$. La perte est donc estimée à environ 5% entre $n = 16$ et $n = 15$. Ce résultat implique que pour pouvoir gagner du temps en retirant un cœur de calcul à l'application, notre mécanisme de progression doit pouvoir recouvrir plus de 5% du temps total de l'application.

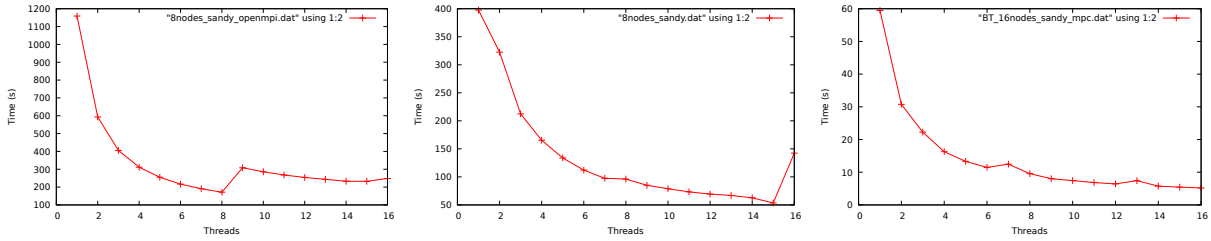


FIGURE 1 – Passage à l'échelle sur Nœud Sandy Bridge, avec 1 rang MPI par nœud (de gauche à droite) : lulesh/OpenMPI, miniMD/MadMPI et BT-MZ/MPC

4. Performance des collectives non bloquantes

4.1. Mesure des Collectives non bloquantes

Pour gagner en performance, il faut donc que l'on gagne suffisamment de temps sur les communications pour recouvrir le temps de calcul perdu. Il est donc nécessaire de mesurer le recouvrement des NBC, ainsi que leur vitesse pour évaluer l'impact de la progression sur les performances. L'évaluation des NBC se base sur plusieurs critères : le temps de la collective et l'efficacité du recouvrement de la collective. Lorsque l'on mesure une collective, chaque rang va mesurer son temps. On définit donc le temps de communication d'une collective comme le temps du dernier rang à terminer la collective, auquel on soustrait le temps du premier rang à la lancer : $tcoll = max(tfin_{1..n}) - min(tdebut_{1..n})$. Dans le cas de collectives inter-nœuds il est nécessaire d'utiliser des mécanismes de synchronisation d'horloge pour avoir des temps cohérents.
Le ratio d'overhead [5] est une métrique permettant d'évaluer l'efficacité du recouvrement des communications point-à-point non bloquantes. Elle peut être adaptée à la mesure des NBC en prenant en compte les temps des différents rangs MPI. Pour avoir une mesure fiable de la capacité de recouvrement, il est important que les temps de communications et de calculs soient similaires. Pour cela nous avons réalisé un petit programme qui permet d'estimer quels volumes de calcul et de données sont nécessaires pour obtenir un temps de calcul (multiplication de matrice) et de communication (pour chaque collective) équivalent à la valeur souhaitée. $tmesure$ est le temps que prends le calcul donnant le temps $tcalc$ et la communication prenant le temps $tcoll$ en situation de recouvrement. Il est mesuré selon le protocole suivant :

```
t1 = gettime ();
MPI_Icollective ();
Calcul ();
MPI_Wait ();
t2 = gettime ();
tmesure = t2-t1;
```

```
t1 = gettime ();
MPI_Icollective ();
MPI_Wait ();
t2=gettime ();
Calcul ();
t3=gettime ();
tcalc = t3 - t2;
tcoll = t2 - t1;
```

$$overhead = \frac{tmesure - max(tcoll, tcalc)}{min(tcoll, tcalc)}$$

4.2. Résultats

Nous avons mis en oeuvre ce protocole sur les deux principales implémentations MPI open-source OpenMPI 4.0.2 et Mpich 3.3, ainsi que sur MadMPI (implémentation basée sur New Madeleine développée à L'INRIA Bordeaux). OpenMPI n'a pas d'option pour lancer son thread de progression, il est donc lancé avec ses options par défaut. MadMPI est aussi lancé avec ses options par défaut : des threads de progression pouvant migrer sur des cœurs libres. Ces threads sont volontairement peu actifs pour ne pas perturber l'application. Mpich est lancé avec son thread de progression à l'aide de la variable d'environnement MPICH_ASYNC_PROGRESS=1. Nous exécutons ce benchmark sur 8 nœuds Skylake de 48 cœurs, avec la collective MPI_Ireduce.

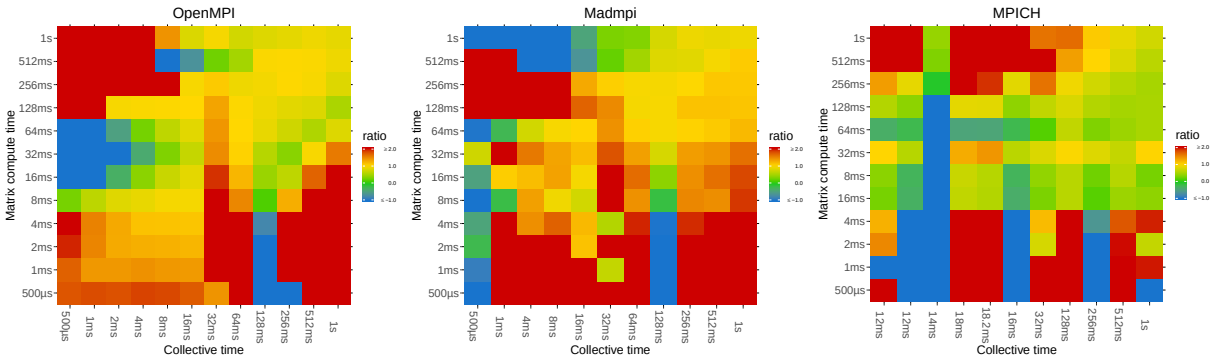


FIGURE 2 – Comparaison du recouvrement de collectives

La figure 2 montre les résultats obtenus. L'axe des abscisses représente le temps de communication et celui des ordonnées le temps de calcul. Les couleurs représentent le ratio d'overhead en fonction du temps de collective et du temps de calcul, le nuancier de couleur va de -1.0 à 2.0. Un ratio de 0.0 (vert) correspond à un recouvrement parfait : $tmesure = max(tcoll, tcalc)$. Un ratio de 1.0 (jaune) correspond à une exécution séquentielle : $tmesure = tcoll + tcalc$. Un ratio de 2.0 (rouge) correspond à une exécution plus longue : $tmesure > tcoll + tcalc$. Un ratio de -1.0 (bleu) implique que : $tmesure < max(tcoll, tcalc)$.
La diagonale où les temps de calculs et de communication sont identiques et sa zone proche sont les plus représentatifs. Au contraire pour les coins ou les temps de calcul et temps de communications sont très différents (en haut à gauche et en bas à droite), la moindre imprécision de mesure peut faire varier le ratio vers des valeurs extrêmes et le rendre inexploitable. Cela explique la présence des parties bleues sur les différents résultats.
Les résultats de la figure 2 montrent une progression quasi inexistante pour MadMPI et OpenMPI. Mpich en revanche arrive à légèrement recouvrir ses communications. Nous rappelons que les temps de calcul et de communications sont ici fixés. Les performances sont donc mesurées en quantité de données transmises et calculées pendant ce temps fixe. Par exemple, pour 1s de calcul, le benchmark va calculer une matrice de 1900x1900 avec OpenMPI, 1891x1891 avec MadMPI et 1573x1573 avec Mpich. Pour Mpich figure 2 à droite, le temps est recouvert en partie mais la quantité de calcul sur un temps fixe est moindre par rapport aux autres implémentations. Nous avons relancé le benchmark sans le thread de progression de Mpich. Dans

cette situation, une matrice de 1912x1912 est calculée en 1s. Cependant, il n'y a pas de recouvrement des communications.
On peut en déduire que le thread de progression de Mpich est efficace, mais ralentit fortement les calculs. La question se pose donc de savoir si un cœur dédié permet d'avoir un recouvrement acceptable, sans impacter l'application. Pour être utile, le recouvrement doit permettre de paralléliser calcul et communications et ainsi de compenser d'éventuels ralentissement du calcul. Si le calcul est trop ralenti, il est devient impossible de tout recouvrir et donc de gagner du temps.

5. Progression des communications avec des cœurs dédiés

5.1. Mise en œuvre de la progression à base de cœur dédié

Dans la partie précédente nous avons vu qu'un mécanisme de progression doit être efficace mais aussi avoir un impact limité sur l'application. Pour cela, nous proposons d'évaluer les performances d'un cœur dédié aux communications, et nous mettons en œuvre cette stratégie dans New Madeleine. le cœur 0 est premièrement réservé pour le thread de progression, et on attache chaque thread OpenMP sur les cœurs suivants, à raison d'un thread par cœur. New Madeleine organise chaque collective sous forme d'une succession d'évènements. Il est donc nécessaire d'utiliser une politique d'attente adéquate pour ces évènements. Comme un cœur est réservé pour faire la progression nous optons pour de l'attente active. Cette méthode est extrêmement réactive à la soumission d'un nouvel évènement, cependant il est très demandant en ressource de calcul. Dans notre cas, cela ne pose pas de problème en raison de notre ressource dédiée. De plus, l'attente active sur une variable impacte très peu la mémoire. Cela ne devrait pas avoir d'effet visibles sur les autres threads. Nous avons testé l'efficacité de cette politique de cœur dédié sur le même protocole utilisé en section 4.1. Les résultats sont présentés dans la figure 3.
On peut observer ici que le recouvrement est efficace sur une grande partie des couples (tcoll,tcalc) testés. Le calcul n'est pas perturbé sur les cœurs de l'application. En effet, on observe toujours une matrice de même taille calculée en 1s (1900x1900) sur chaque cœur de calculs (15 sur 16 sur Sandy et 47 sur 48 pour Skylake).

5.2. Résultat du cœur dédié sur l'application P3DFFT

Nous voulons savoir si le recouvrement des communications permet d'avoir un gain de temps global, en compensant le retrait d'un cœur pour le calcul. Pour cela nous avons choisi de faire un test sur une application réelle : P3DFFT [11]. P3DFFT est une bibliothèque numérique réalisant des transformées de Fourier en trois dimensions et utilisant des collectives non bloquantes (MPI_Ialltoall).
Nous comparons ici les différents mécanismes de progression dans New Madeleine : sans progression, progression par défaut, et politique de cœur dédié. Nous exécutons P3DFFT sur Sandy avec le même jeu d'entrée représentatif. Les temps observés sont regroupés dans le tableau 4. On observe que la progression par défaut (thread peu actif et pouvant migrer) est plus lente que sans progression. Cela est dû à l'ordonnancement opportuniste des threads de progression sur des cœurs occupés par OpenMP. Dans cette situation le thread OpenMP est ralenti, ce qui va ralentir toute la section parallèle.

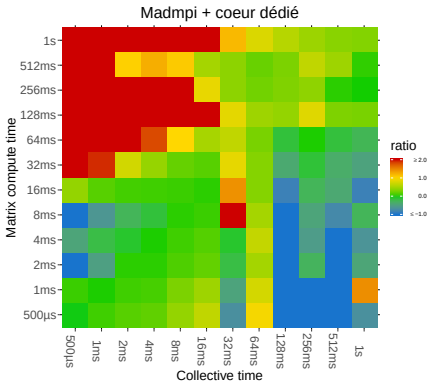


FIGURE 3 – Recouvrement avec un cœur dédié

Mécanisme	durée
Sans progression	137.42s
Progression par défaut	141.71s
Progression cœur dédié	133.40s

FIGURE 4 – Performances de P3DFFT en fonction des mécanismes de progression

L'application est donc fortement impactée, ce qui augmente le temps d'exécution. La version "progression avec cœur dédié" est plus performante malgré le retrait d'un cœur à l'application. Cela est dû aux bonnes performances du recouvrement permettant de compenser l'augmentation du temps de calcul.

5.3. Quel profil d'applications ?

Cette expérience montre qu'il est possible d'avoir un gain de performances sur certaines applications. Ces applications doivent être hybride (MPI+OpenMP ou autres threads de calcul). Nous avons utilisé l'application de profiling mpiP [16], pour determiner les caractéristiques de celles-ci. MpiP nous permet de mesurer quel est le pourcentage de communication MPI au sein de notre application. Nous avons remplacé les appels non bloquants par leur version bloquante afin de mesurer le volume de communication réel. On observe que le volume doit être conséquent : comme nous avons mesuré une perte théorique d'environ 5% il est impossible de gagner du temps si le volume de communication est inférieur à 5% du temps total.
L'application doit être conçue pour exécuter suffisamment de calcul pendant ces communications. Dans le cas contraire, si le calcul est trop court, l'application va perdre du temps dans le `MPI_Wait`.

6. Conclusion

Dans cet article, nous nous sommes intéressés à la progression des collectives non bloquantes en contexte hybride MPI + OpenMP. En particulier, nous avons cherché a obtenir un recouvrement efficace sans trop pénaliser le calcul afin de gagner du temps. Nous avons proposé dans cet article une méthode de mesure des collectives non bloquantes tenant compte à la fois du recouvrement des communications, mais aussi de la performance réelle des collectives. Nous avons pu, à partir de cette méthode, analyser les performances des collectives non bloquantes pour plusieurs bibliothèques MPI.
Nous avons mis en œuvre un mécanisme permettant d'avoir un taux d'overhead quasi nul, en particulier pour les collectives, tout en limitant l'impact sur les applications. Nous avons pu vérifier ces expériences dans un cas concret en accélérant l'application P3DFFT. Enfin, nous avons défini quel type d'application est particulièrement gagnant à l'utilisation de ce mécanisme de progression. Dédier un cœur à la progression des communications est donc une solution efficace sur des codes conçus pour le recouvrement. Cependant, les applications conçues pour utiliser des collectives non bloquantes sont très peu nombreuses. Cette solution est donc un bon moyen de favoriser le développement de ce type d'applications.

Bibliographie

1. Amdahl (G. M.). – Validity of the single processor approach to achieving large scale computing capabilities. – In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, AFIPS '67 (Spring), p. 483–485, New York, NY, USA, 1967. Association for Computing Machinery.
2. Aumage (O.), Brunet (E.), Furmento (N.) et Namyst (R.). – New madeleine : a fast communication scheduling engine for high performance networks. – In *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, March 2007.
3. Bailey (D. H.), Barszcz (E.), Barton (J. T.), Browning (D. S.), Carter (R. L.), Dagum (L.), Fatoohi (R. A.), Frederickson (P. O.), Lasinski (T. A.), Schreiber (R. S.), Simon (H. D.), Venkatakrishnan (V.) et Weeratunga (S. K.). – The NAS parallel benchmarks summary and preliminary results. – In *Supercomputing '91 :Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pp. 158–165, 1991.
4. Denis (A.). – pioman : a pthread-based Multithreaded Communication Engine. – In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, mars 2015.
5. Denis (A.) et Trahay (F.). – MPI Overlap : Benchmark and Analysis. – In *International Conference on Parallel Processing*, *45th International Conference on Parallel Processing*, 45th International Conference on Parallel Processing, Philadelphia, United States, août 2016.
6. Forum (M.). – *MPI - a Message Passing Interface Standard V3.0. Ean 1114444410030*. – English Book Service, 2012.
7. Heroux (M. A.), Doerfler (D. W.), Crozier (P. S.), Willenbring (J. M.), Edwards (H. C.), Williams (A.), Rajan (M.), Keiter (E. R.), Thornquist (H. K.) et Numrich (R. W.). – Improving performance via mini-applications.
8. Hoefler (T.) et Lumsdaine (A.). – Message progression in parallel computing - to thread or not to thread ? – In *2008 IEEE International Conference on Cluster Computing*, pp. 213–222, Sep. 2008.
9. Hunold (S.) et Carpen-Amarie (A.). – On the impact of synchronizing clocks and processes on benchmarking MPI collectives. – In *EuroMPI*, pp. 8 :1–8 :10. ACM, 2015.
10. Karlin (I.), Keasler (J.) et Neely (R.). – *LULESH 2.0 Updates and Changes*. – Rapport technique nLLNL-TR-641973, August 2013.
11. Pekurovsky (D.). – P3DFFT : a framework for parallel computations of fourier transforms in three dimensions. *CoRR*, vol. abs/1905.02803, 2019.
12. Rashti (M. J.) et Afsahi (A.). – Improving communication progress and overlap in MPI rendezvous protocol over rdma-enabled interconnects. – In *2008 22nd International Symposium on High Performance Computing Systems and Applications*, pp. 95–101, 2008.
13. Si (M.), Peña (A.), Balaji (P.), Takagi (M.) et Ishikawa (Y.). – MT-MPI : multithreaded MPI for many-core environments. – 06 2014.
14. Sur (S.), Jin (H.-W.), Chai (L.) et Panda (D.). – RDMA read based rendezvous protocol for MPI over infiniband. – pp. 32–39, 01 2006.
15. Taboada (H.). – Impact du placement des threads de progression pour les collectives nonbloquantes. – In *Compas 2016 : conférence d'informatique en Parallélisme, Architecture et Système*, Lorient, France, juillet 2016.
16. Vetter (J.) et Chambreau (C.). – mpiP : Lightweight, scalable mpi profiling. *URL : http ://www. llnl. gov/CASC/mpiP*, 01 2005.