# Étude de l'impact d'une clause d'affinité sur les performances et l'énergie dans un support exécutif OpenMP

Philippe Virouleau

**Compas'2017** : Parallélisme / Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

# Étude de l'impact d'une clause d'affinité sur les performances et l'énergie dans un support exécutif OpenMP

Philippe Virouleau

Inria,
Univ. Grenoble Alpes, CNRS, Grenoble Institute of Technology, LIG, Grenoble, France
philippe.virouleau@inria.fr

**Résumé**
La norme OpenMP 4.0 a introduit les tâches avec dépendances, qui permettent au programmeur d'exprimer du parallélisme à grain fin.
Contrôler la localité des données lors de l'exécution de ces tâches est l'un des facteurs clés pour obtenir un bon niveau de performances sur des architectures NUMA. Dans ce domaine, OpenMP ne propose pas encore beaucoup de flexibilité au programmeur, ce qui laisse au support exécutif la responsabilité de décider où les tâches doivent être exécutées.
Cet article présente la description, l'implémentation, et l'évaluation d'une nouvelle clause *affinity* pour les tâches, qui est actuellement en discussion au sein du comité du langage OpenMP. Cette clause permet au programmeur de donner des informations au support exécutif à propos du placement de la tâche au cours de l'exécution, ce qui peut également être utilisé par le programmeur pour contrôler le placement de données sur l'architecture. Il présente également des classes d'applications qui pourraient bénéficier d'une telle flexibilité de placement de tâches et de données, ainsi qu'un aperçu rapide de l'impact sur la consommation énergétique.

**Mots-clés :** OpenMP, affinité, support exécutif, NUMA, énergie

## 1. Introduction

Les architectures à temps d'accès mémoire non uniforme (NUMA) sont aujourd'hui le choix le plus populaire pour créer des grosses machines à mémoire partagée. Le temps d'accès à la mémoire dépend de la distribution physique des données, leur placement et leur localité durant la vie du programme sont donc des points clés pour améliorer la scalabilité et les performances.
Les environnements de programmation parallèle tels que OpenMP, OpenCL, ou TBB sont devenus très populaires pour exploiter les machines à mémoire partagée avec plusieurs centaines de cœurs. La plupart d'entre eux fournissent également un moyen d'équilibrer dynamiquement la charge de travail sur tous les processeurs, mais aucun d'entre eux ne fournit de moyen assez flexible de gérer la localité des données sur des systèmes NUMA.
Le concept de *places* ajouté avec OpenMP 4.0 permet d'associer les threads d'une région parallèle à un ensemble de cœurs physiques, ce qui permet d'aider à conserver l'affinité d'un thread avec la mémoire. Néanmoins cela n'est pas suffisant pour améliorer les performances des applications à base de tâches, au cours desquelles les tâches sont ordonnancées dynamiquement sur les différent threads.

Cet article présente un contrôle du placement des tâches et des données dans un support exécutif OpenMP, en implémentant une clause *affinity*. Il montre également comment cette information peut être utilisée à l'exécution pour améliorer la localité des données, ce qui a un impact direct sur les performances et la consommation énergétique. L'évaluation de ces travaux a été effectuée sur deux machines NUMA : une disposant de 192 cœurs, et l'autre disposant de 48 cœurs.
Le plan de cet article est le suivant : la partie 2, contient quelques informations essentielles sur les architectures NUMA et les outils utilisés. La partie 3 regroupe la description de la syntaxe et de l'implémentation d'une clause *affinity*. La partie 4 présente les évaluations de performances sur des applications de types Stencil et d'algèbre linéaire. Enfin les travaux de l'état de l'art sont présentés dans la partie 5, avant les travaux en cours et la conclusion.

## 2. Cible et outil

### 2.1. Description des systèmes NUMA et de leur exploitation

Les architectures NUMA sont des machines à mémoire partagée dont les cœurs physiques ainsi que la mémoire sont divisés en plusieurs ensembles ("nœuds"), interconnectés entre eux de manière transparente pour l'utilisateur.
Afin d'exploiter de telles architectures, le programmeur a besoin d'une part d'exprimer beaucoup de parallélisme à grain fin, pour profiter au maximum du nombre important de processeurs disponibles ; et d'autre part de contrôler l'exécution de l'application, en particulier la manière dont sont distribuées et accédées les données.
Les environnements de programmation parallèle à base de tâches fournissent un moyen d'exprimer le parallélisme à grain fin. OpenMP [12], le standard utilisé en pratique pour la programmation des architectures à mémoire partagée, supporte le parallélisme à base de tâches avec dépendances de données depuis la version 4.0.

### 2.2. Mesure et consommation énergétique

Les mesures énergétiques effectuées ont été basées sur la fonctionnalité Intel RAPL (Running Average Power Limit), disponible sur les processeurs Intel de nos machines. Cette fonctionnalité expose la consommation de certains composants sur le socket (comme l'ensemble du processeur - *package* - et la DRAM) à travers les MSR (Model Specific Registers).
La précision de ces compteurs dépend du modèle de processeur, les différences entre Sandy Bridge et Haswell on été étudié dans [6, 13].
Un petit outil a été conçu pour accéder périodiquement aux informations des registres MSR : basé sur l'API de LIKWID [14], il récupère les informations pour la totalité du processeur (`PWR_PKG_ENERGY`), pour les coeurs (`PWR_PP0_ENERGY`), et pour la DRAM (`PWR_DRAM_ENERGY`). Cet outil récupère les valeurs toute les 100ms, et les associe à un timestamp.

## 3. Extension d'OpenMP pour supporter l'affinité

Cette partie détaille l'introduction du mot clé **`affinity`**, ainsi que l'implémentation côté support exécutif pour exploiter cette fonctionnalité.

### 3.1. Description d'une clause affinité

Les deux principaux composants des architectures NUMA que l'on considère pour cette proposition sont les cœurs et les nœuds. Un point clé pour obtenir de bonnes performances sur des architectures NUMA est de garantir qu'une tâche s'exécute *proche* de ses données. On distingue

donc trois types d'affinité que le programmeur pourrait avoir besoin d'exprimer :

**affinité à un thread :** le support exécutif devrait essayer d'ordonnancer la tâche sur le thread donné.

**affinité à un nœud NUMA :** le support exécutif devrait essayer d'ordonnancer la tâche sur n'importe quel thread du nœud NUMA donné.

**affinité à une donnée :** quand une tâche devient prête pour l'exécution, le support exécutif devrait l'ordonnancer sur n'importe quel thread attaché au nœud NUMA sur lequel la donnée a été physiquement allouée.

De plus, le programmeur peut indiquer si cette affinité est *stricte*, indiquant que la tâche **doit** s'exécuter sur la ressource indiquée. Si le programmeur n'indique pas une affinité stricte, l'ordonnanceur peut décider d'exécuter la tâche sur une ressource différente, pour assurer l'équilibrage de charge par exemple.
Cette extension visant les constructions de type tâche, elle a été implémentée comme une nouvelle clause pour la directive `task`. La syntaxe proposée est la suivante :

```
affinity([node | thread | data]: expr[, strict])
```

Si *expr* désigne un id de thread, elle devrait désigner l'id de thread dans les `OMP_PLACES` définies pour la *team* courante. Exemple : si les places sont `"{2},{5},{8}"`, et que *expr* est évaluée comme valant 0, l'id du thread désigné est `"{2}"`.
Si *expr* désigne un id de nœud NUMA, elle devrait désigner un id de nœud dans l'ensemble des nœuds NUMA construit à partir de la liste des `OMP_PLACES`.
Si *expr* désigne une donnée, elle devrait être une adresse mémoire. Si le nœud NUMA associé à la donnée ne peut être déterminé, le nœud par défaut est le premier dans la *team* OpenMP.
Si *expr* désigne une ressource hors limites, la valeur est prise modulo le nombre de ressources.

### 3.2. Implémentation dans un compilateur et un support exécutif

La clause a été implémentée dans le frontend de Clang, et le support exécutif distribué avec LLVM (basé sur le support exécutif d'Intel) a été modifié pour supporter l'affinité. Ce support exécutif ayant une vision uniforme de la machine, il a fallu premièrement l'étendre pour mettre en place une vision hiérarchique de la machine. Cela a été fait en intégrant la gestion des queues hiérarchiques présente dans LIBKOMP [5, 3], ainsi que la gestion du vol de travail présent dans XKAAPI [1, 9].
Ensuite le support exécutif ne proposait qu'une stratégie de vol de travail aléatoire, il fallait donc ajouter des stratégies de vol pour prendre en compte les différentes queues hiérarchiques. Cela a été fait en intégrant des précédents travaux concernant des stratégies d'ordonnancement [15] sur architectures NUMA. L'implémentation de la clause `affinity` vient se greffer par dessus ces travaux, en ajoutant des stratégies de vol de travail.

## 4. Résultat expérimentaux

Les expériences ont été effectuées sur deux machines :
Une SGI UV2000, constituée de 24 nœuds NUMA, possédant chacun un processeur Intel Xeon E5-4640 à 8 cœurs, le tout formant un total de 192 cœurs. Cette machine dispose de 31Go de RAM par nœud, pour un total de 744Go. Le nom Intel192 sera utilisé pour faire référence à cette machine dans l'article.
Une SuperMicro 4048B-TR4FT, constituée de 4 nœuds NUMA, possédant chacun un processeur Intel Haswel-EX E7-4830 à 12 cœurs, le tout formant un total de 48 cœurs. Chacun des nœuds

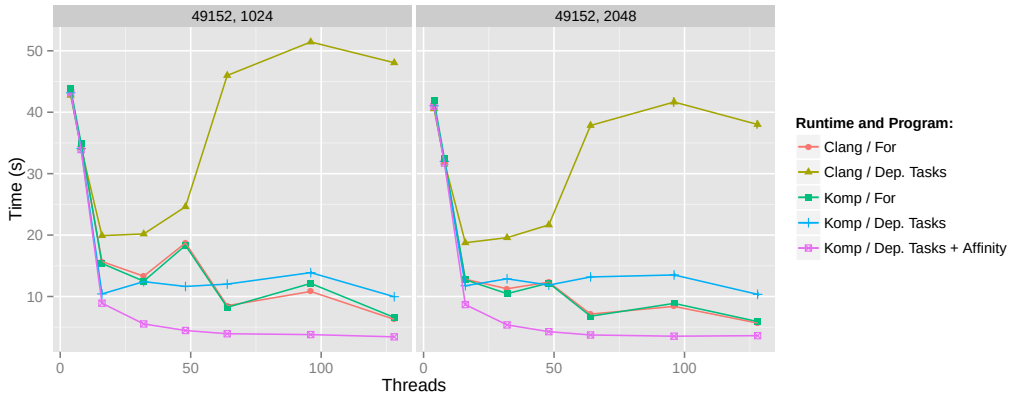


FIGURE 1 – Performance de Jacobi sur Intel192 avec une taille de matrice de 49152

dispose de 64Go de RAM, pour un total de 256Go. Sur cette machine l'hyperthreading peut être activé pour atteindre un nombre de 96 cœurs. Le nom Intel48 sera utilisé pour faire référence à cette machine dans l'article.
Deux types d'applications ont été considérés pour montrer l'intérêt d'une clause affinité :

— Les applications de type Stencil, illustrées ici par le programme Jacobi.
— Les applications d'algèbre linéaires, illustrées ici par les factorisations de Cholesky et QR.

Ces applications sont tirées de la suite de benchmarks KASTORS [16]. Elles ont été évaluées sur trois supports exécutifs différents : libGOMP, le support exécutif de Gcc (6.3) ; libOMP, le support exécutif de Clang (4.0), et libKOMP, décrit dans la section 3.2. Les résultats présentés sont des moyennes sur 6 runs, les barres d'erreurs sont affichées sur les courbes et permettent de montrer la stabilité des résultats.

### 4.1. Stencil

Le noyau Jacobi est une application de type stencil 2D. Les performances de ce type d'application sont très dépendantes de la réutilisation du cache entre les itérations ; les implémentations à base de boucles OpenMP sont donc généralement beaucoup plus efficaces que celles à base de tâches, vu que pour ces dernières l'ordonnanceur peut déplacer deux tâches qui devraient rester sur le même cœur.
Plusieurs implémentations de ce noyau ont été comparées. Une version itérative utilisant des constructions de type *for* pendant l'initialisation et le calcul (nommée "For"). Une version à base de tâches avec dépendances pour l'initialisation et le calcul (nommée "Dep. Tasks"). Une version à base de tâches avec dépendances incluant des affinités (nommée "Dep. Tasks + Affinity"). Cette dernière version utilise des affinités strictes pour l'initialisation et le calcul, indiquant précisément la ressource sur laquelle la tâche devrait s'exécuter en fonction du nombre de threads. Parmi les expériences effectuées, les résultats présentés en figure 1 donnent un bon aperçu des comportements observés sur les différents support exécutifs.
Le premier commentaire que l'on peut faire est que l'application ne passe globalement pas très bien à l'échelle, peu importe le support exécutif utilisé. Le programme est *memory bound* et on ne peut pas faire beaucoup plus que s'assurer que le calcul s'effectue proche des données.
La version tâche avec dépendances de base n'est pas très performante (c'est même catastrophique avec le support exécutif de Clang !). Ici la seule explication derrière ces performances

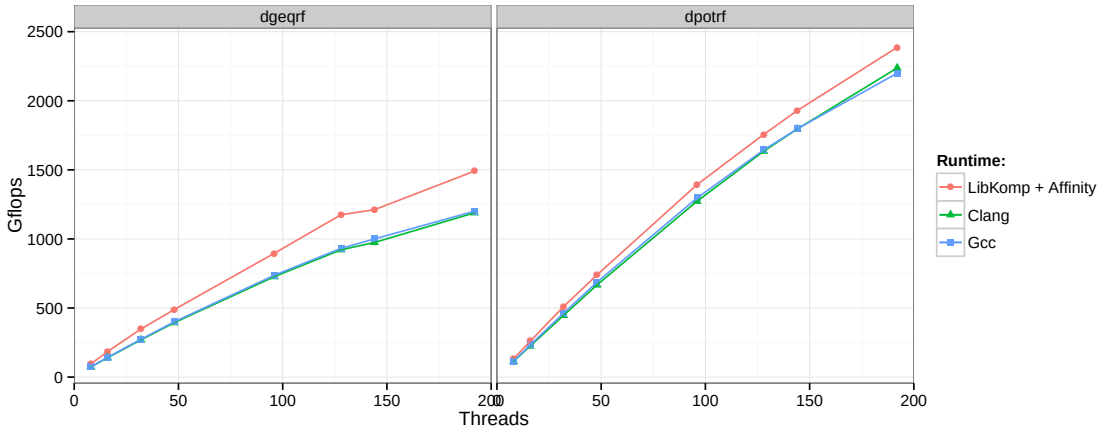


FIGURE 2 – Cholesky et QR sur Intel192 (Gflops). Taille de matrice 32768, taille de bloc 512

est le fait que la stratégie de sélection de victimes dans le vol de travail est complètement aléatoire dans libOMP, alors qu'il y a une sélection hiérarchique dans libKOMP. Cela n'empêche pas que pour les deux supports exécutif, certains vol de tâches dégradent les performances !
On remarque que la version avec affinité offre les meilleures performances grâce à un contrôle strict de l'association entre une tâche et ses données.
Étonnamment la version itérative "For" ne présente pas les même résultats que la version avec affinité : un meilleur découpage des itérations devrait être à même de palier à ce problème. Note : les pics à 48 et 96 threads sont probablement dû au fait que ces nombres ne sont pas des puissances de 2, ce qui ne permet pas automatiquement un découpage des itérations optimal correspondant à la topologie.

### 4.2. Algèbre linéaire

L'objectif de cette section est double : observer l'impact de l'affinité sur les performances, et faire une première observation de l'impact potentiel sur l'énergie consommée.
La version "affinité" des factorisations de Cholesky et QR implémente une affinité assez similaire à l'*owner compute rule* présente dans HPF [8] : les différents noyaux de chaque application sont annotés d'une affinité sur les données en écriture pour ce noyau. L'initialisation des blocs de données a été fixée via une affinité stricte, selon une distribution bloc cyclique sur les nœuds NUMA.
Les gains de performances les plus importants ont été observés sur Intel192, qui dispose de 24 nœuds NUMA (contre 4 pour Intel48), et est donc plus sensible aux problématiques de localité de données. La figure 2 présente l'évolution des performances des factorisations de Choleky (dpotrf) et QR (dgeqrf) en fonction du nombre de threads, sur Intel192.
Afin de palier au manque de distribution de données dans les supports exécutifs de Gcc et de Clang, les données ont été distribuées via *numactl* pour ces expériences.
L'impact de l'affinité sur les performances est très clair : distribuer les données par "bloc" et assurer un ordonnancement hiérarchique au plus proche des données écrites offre de bien meilleures performances que de distribuer les données par "page" (ce que fait numactl), sans faire attention à la localité des données.
Cette affirmation est à moduler en fonction de la taille de matrice et de bloc : si la taille de bloc est trop petite, le temps économisé sur le transfert de données n'est pas significatif et les

TABLE 1 – Performances et énergie (Joules) consommée sur 48 threads d'Intel48

| Taille (bloc) | Runtime | dpotrf | | | dgeqrf | | |
|---|---|---|---|---|---|---|---|
| | | GFlop/s | E(PKG) | E(DRAM) | GFlop/s | E(PKG) | E(DRAM) |
| 16384 (256) | Clang | 665.94 | 796.52 | 80.00 | 455.75 | 4942.94 | 539.80 |
| | Gcc | 682.10 | 787.09 | 83.51 | 470.03 | 4633.48 | 498.07 |
| | Komp + Affinity | 677.91 | 784.54 | 73.80 | 471.10 | 4573.89 | 481.64 |
| 32768 (512) | Clang | 737.93 | 6284.87 | 682.36 | 503.02 | 36901.91 | 4321.06 |
| | Gcc | 746.41 | 6005.53 | 635.87 | 502.82 | 35797.91 | 4166.94 |
| | Komp + Affinity | 750.09 | 5866.55 | 599.66 | 508.12 | 34964.64 | 4058.75 |

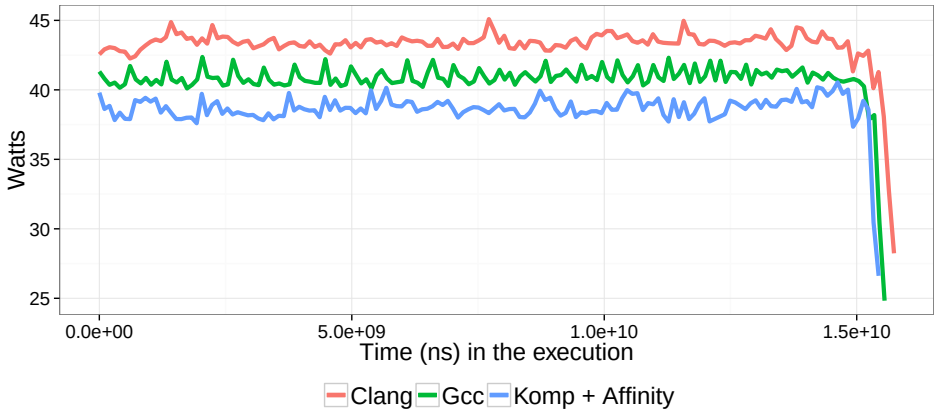


FIGURE 3 – Consommation (Watts) de la DRAM au cours d'un Cholesky sur Intel48

performances sont équivalentes à celles d'un support exécutif utilisant une stratégie aléatoire de vol de travail (ce qui est le cas de ceux de Clang et Gcc).

Sur Intel48 la différence de performance est beaucoup moins flagrante, néanmoins on peut observer un effet sur l'énergie consommée. La table 1 récapitule les performances, l'énergie consommée par l'ensemble du processeur (PKG), et celle consommée par la DRAM, en fonction du support exécutif et de la taille de matrice.

On peut observer que le support exécutif avec affinité n'est pas significativement meilleur que ceux de Gcc et Clang, par contre l'énergie consommée par le package du processeur est plus faible.

Évidemment quand les performances sont légèrement meilleures on peut s'attendre à ce que l'énergie consommée soit plus faible vu que le temps d'exécution est plus court. Or même dans certains cas où les performances de la version avec affinité sont en retrait, la consommation énergétique est plus faible (ce qui est d'autant plus vrai quand les performances sont équivalentes). Une partie de cela est probablement du au fait que l'affinité permet dans tous les cas de diminuer la consommation énergétique de la DRAM.

Ce comportement est mis en valeur dans la figure 3, illustrant la puissance consommée par la DRAM au cours d'une exécution de Cholesky sur une matrice de taille 32768 (taille de bloc 512).

## 5. État de l'art

De nombreux projets de recherche se sont intéressés à l'exécution d'applications OpenMP sur des machines NUMA.
À l'Université de Houston des chercheurs ont travaillés sur l'amélioration de l'affinité mémoire des régions parallèle et boucles sur des architectures hiérarchiques [10] ; ainsi que sur l'alignement des tâches et des données sur des partitions logiques de l'architecture appelées *locations* [7].
Drebes et al. [4] ont proposés des techniques d'ordonnancement pour contrôler le placement des tâches et des données, pour tirer parti de la localité des données. Ils ont implémenté ces techniques dans OpenStream, un modèle de programmation par flot de données. Leur approche se place du point de vue de l'ordonnanceur et ne permet pas à l'utilisateur d'avoir de la flexibilité dans le placement des données. Olivier et al. [11] ont introduit des queues de tâches OpenMP au niveau des noeuds, appelés *locality domains*, pour améliorer la localité des tâches vis à vis de leur données sur des architectures NUMA. Le placement et la localité des données sont obtenus implicitement en considérant la politique d'allocation *first-touch* du système.
Enfin l'équipe Runtime de l'Université de Bordeaux a proposée le support exécutif Forest-GOMP [2], qui utilise une API pour exprimer des affinités entre les régions parallèle OpenMP et des données allouées dynamiquement. Les affinités mémoire sont déterminées dynamiquement et peuvent être prise en compte dans l'équilibrage de charge.

## 6. Conclusion et travaux à venir

Les environnements de programmation à base de tâches tels qu'OpenMP sont devenus un moyen standard de programmer des systèmes NUMA à large échelle. Ils offrent au programmeur un moyen d'exprimer du parallélisme à grain fin, qui peut être associé dynamiquement à la topologie de l'architecture. OpenMP a récemment évolué pour permettre d'exprimer les dépendances de données entre tâches.
Cet article présente une nouvelle clause `affinity` offrant plus de flexibilité au programmeur pour exprimer le lien qui existe entre une tâche et ses données. Les résultats expérimentaux montrent d'une part qu'une telle clause permet d'offrir des performances qui n'étaient avant atteignables qu'avec des programmes itératif (Stencil) ; et d'autre part que les performances des applications d'algèbre linéaire peuvent directement en bénéficier lorsque l'architecture cible est fortement impactée par les problématiques NUMA. À court terme des collaborations avec d'autres personnes et équipes sont prévues pour offrir des observations plus exhaustives du point de vue de l'énergie.

## Remerciements

Ce travail est réalisé dans le cadre du projet ELCI, un projet collaboratif Français financé par le FSN ("Fond pour la Société Numériqe"), qui associe des partenaires académiques et industriels pour concevoir et produire un environnement logiciel pour le calcul intensif.

## Bibliographie

1. Bleuse (R.), Gautier (T.), Lima (J. V. F.), Mounié (G.) et Trystram (D.). – *Euro-Par 2014 Parallel Processing : 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, chap. Scheduling Data Flow Program in XKaapi : A New Affinity Based Algorithm for Heterogeneous Architectures, pp. 560–571. – 2014.

2. Broquedis (F.), Furmento (N.), Goglin (B.), Wacrenier (P.-A.) et Namyst (R.). – ForestGOMP : an efficient OpenMP environment for NUMA architectures. *International Journal on Parallel Programming, Special Issue on OpenMP* ;, vol. 38, n5, 2010, pp. 418–439.
3. Broquedis (F.), Gautier (T.) et Danjean (V.). – Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. – In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, *IWOMP'12*, IWOMP'12, pp. 102–115. Springer-Verlag, 2012.
4. Drebes (A.), Heydemann (K.), Drach (N.), Pop (A.) et Cohen (A.). – Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, vol. 11, n3, août 2014, pp. 30 :1–30 :25.
5. Durand (M.), Broquedis (F.), Gautier (T.) et Raffin (B.). – An efficient openmp loop scheduler for irregular applications on large-scale numa machines. – In *Proceedings of the 9th International Conference on OpenMP in the Era of Low Power Devices and Accelerators*, pp. 141–155, 2013.
6. Hackenberg (D.), Schöne (R.), Ilsche (T.), Molka (D.), Schuchart (J.) et Geyer (R.). – An energy efficiency feature survey of the intel haswell processor. – In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 896–904, May 2015.
7. Huang (L.), Jin (H.), Yi (L.) et Chapman (B.). – Enabling locality-aware computations in openmp. *Sci. Program.*, vol. 18, n3-4, août 2010, pp. 169–181.
8. Kennedy (K.), Koelbel (C.) et Zima (H.). – The rise and fall of high performance fortran : An historical object lesson. – In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, *HOPL III*, HOPL III, pp. 7–1–7–22. ACM, 2007.
9. Lima (J. V. F.), Gautier (T.), Danjean (V.), Raffin (B.) et Maillard (N.). – Design and analysis of scheduling strategies for multi-cpu and multi-gpu architectures. *Parallel Computing*, vol. 44, 2015, pp. 37–52.
10. Marowka (A.), Liu (Z.) et Chapman (B.). – Openmp-oriented applications for distributed shared memory architectures : Research articles. *Concurr. Comput. : Pract. Exper.*, 2004.
11. Olivier (S. L.), de Supinski (B. R.), Schulz (M.) et Prins (J. F.). – Characterizing and mitigating work time inflation in task parallel programs. – In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 65 :1–65 :12. IEEE Computer Society Press, 2012.
12. OpenMP Architecture Review Board. – OpenMP application program interface version 4.0, juillet 2013.
13. Rotem (E.), Naveh (A.), Ananthakrishnan (A.), Weissmann (E.) et Rajwan (D.). – Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, vol. 32, n2, March 2012, pp. 20–27.
14. Treibig (J.), Hager (G.) et Wellein (G.). – LIKWID : lightweight performance tools. *CoRR*, vol. abs/1104.4874, 2011.
15. Virouleau (P.), Broquedis (F.), Gautier (T.) et Rastello (F.). – Using data dependencies to improve task-based scheduling strategies on numa architectures. – In *Proceedings of the 22Nd International Conference on Euro-Par 2016 : Parallel Processing*, pp. 531–544, 2016.
16. Virouleau (P.), Brunet (P.), Broquedis (F.), Furmento (N.), Thibault (S.), Aumage (O.) et Gautier (T.). – Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. – In *10th International Workshop on OpenMP, IWOMP2014*, pp. 16 – 29. Springer, septembre 2014.