Amélioration des stratégies d'ordonnancement sur architectures NUMA à l'aide des dépendances de données

Philippe Virouleau

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

Amélioration des stratégies d'ordonnancement sur architectures NUMA à l'aide des dépendances de données

Philippe Virouleau

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

Résumé
Le récent ajout des dépendances de données à la norme OpenMP 4.0 offre au programmeur une manière flexible de synchroniser les tâches. Grâce à cela, le compilateur et le support exécutif peuvent tous les deux savoir exactement quelles données sont lues ou écrites par quelles tâches. Les performances sur architectures NUMA peuvent être fortement impactées par le placement des données et l'ordonnancement des tâches. Les informations présentes dans les dépendances de données peuvent être utilisées pour contrôler le placement physique des données, ainsi que pour contrôler les stratégies de placement des tâches en fonction de la topologie. Cet article présente plusieurs heuristiques pour ces stratégies, et leurs implémentations dans notre support exécutif OpenMP : XKAAPI. Nous présentons également nos évaluations sur des applications d'algèbre linéaire, exécutées sur une machine NUMA à 192 cœurs, et comparées aux stratégies proposées par l'état de l'art.

Mots-clés : OpenMP, tâche avec dépendances, support exécutif, NUMA, ordonnancement

---

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. Compte tenu de la différence de temps d'accès en fonction de la distribution physique de la mémoire (une donnée proche sera accédée beaucoup plus rapidement qu'une donnée distante), contrôler le placement des données tout au long de l'exécution d'une application est donc l'un des points clés pour améliorer sa scalabilité et ses 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. Ils offrent un moyen d'exprimer beaucoup de parallélisme à grain fin, tout en ayant un coût relativement faible. 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 de gérer la localité des données sur des systèmes NUMA.
L'ajout récent des dépendances de données au modèle de programmation par tâches d'OpenMP fournit au support exécutif des informations précises, notamment sur les accès mémoires des tâches de l'application. L'ordonnanceur de tâches peut implémenter des stratégies dédiées aux architectures NUMA, comme montré dans cet article.
Cet article décrit plusieurs stratégies que nous avons implémenté dans notre support exécutif XKAAPI exploité à travers libKOMP [3], une interface pour la norme OpenMP. Nous avons identifié trois points majeurs dans l'ordonnanceur qui peuvent impacter les applications parallèles pour systèmes NUMA. Cet article les décrit et les évalue, en montrant leur impact sur

les performances d'applications s'exécutant sur une machine NUMA à 192 cœurs. Nous les comparons également aux stratégies de l'état de l'art implémentées dans XKAAPI.
Le plan de cet article est le suivant : dans la partie 2, nous donnons quelques informations essentielles sur les architectures NUMA et le modèle de tâches avec dépendances. Ensuite dans la partie 3 nous décrivons les idées, stratégies, et détails d'implémentation que nous avons utilisé pour améliorer les performances du système exécutif. La partie 4 présente les évaluations de performances. Enfin nous présentons des travaux de l'état de l'art dans la partie 5, avant de conclure.

2. Description des systèmes NUMA et de leur exploitation

Nous avons effectué nos expériences sur une machine 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. Nous utilisons le nom Intel192 pour faire référence à cette machine dans l'article.
Afin d'exploiter les grosses architectures à mémoire partagée, 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 ; d'autre part de contrôler l'exécution de l'application, en particulier la manière dont sont distribuées les données.
Les environnements de programmation parallèle à base de tâches fournissent un moyen d'exprimer le parallélisme à grain fin. OpenMP [10], 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.

2.1. Un aperçu du modèle de tâches d'OpenMP

Une *tâche* OpenMP peut être vue comme une *unité indépendante de travail* qu'un thread OpenMP peut exécuter. Les tâches peuvent être exécutées par un thread quelconque de la région parallèle.
Dans la norme OpenMP 3.0, la synchronisation des tâches est effectuée à l'aide du mot clé `taskwait`, imposant l'attente de la complétion de toutes les tâches de la région parallèle courante.
La norme 4.0 enrichit ce concept en introduisant le mot clé `depend`, qui permet de spécifier le mode d'accès des variables utilisées par une tâche pendant son exécution. Le mode d'accès peut être soit `in`, `out`, ou `inout`, en fonction de si la variable correspondante est respectivement lue par la tâche, écrite, ou bien les deux. Cette information peut être analysée par le support exécutif, qui décidera si la tâche est prête à être exécutée ou si elle doit attendre la complétion d'une autre tâche.

2.2. Notre manière d'exécuter les programmes à base de tâches

Le support exécutif que nous développons, XKAAPI, implémente un modèle d'exécution par vol de travail pour les programmes à base de tâches OpenMP. Ce modèle a été présenté initialement dans Cilk [7], et est couramment utilisé dans les environnements de programmation à base de tâches.
XKAAPI crée un thread, appelé un *kproc*, pour chaque unité d'exécution (dans le cas d'une machine NUMA, il s'agit d'un cœur). Chaque kproc possède sa propre queue de travail dans laquelle il poussera les tâches qu'il crée, et qui est implémentée comme une pile.

2.3. Contrôler la distribution des données sur les architectures NUMA

Contrôler le placement des données nécessite de bien connaître l'architecture NUMA. Les programmeurs peuvent contrôler l'allocation des données à l'aide d'outils dédiés, tels que `numactl` [14].

Par exemple l'option `--interleave=all` permet de répartir toutes les pages sur l'architecture suivant une politique de round-robin sur tous les nœuds NUMA. Cette politique est largement utilisée avec du parallélisme dynamique, car elle répartit le trafic mémoire sur tous les contrôleurs mémoire, rendant tous les processeurs "*équitablement mauvais*" par rapport aux accès mémoire.
Nous proposons deux approches différentes pour distribuer la mémoire sur les systèmes NUMA, basées sur le principe de l'allocation *first-touch* de la mémoire : la première en donnant la possibilité d'allouer explicitement la donnée sur un nœud NUMA spécifique de la machine, via une API que nous fournissons [6, 2] ; la seconde en donnant la possibilité de marquer certaines portions de code comme étant du code d'initialisation, et le support exécutif se charge de la distribution des tâches créées dans cette portion. Olivier et al. [9] ont utilisé cette approche avec des restrictions sur l'exécution des tâches créées par l'application.
Ces deux approches permettent au support exécutif de savoir où les données de l'application sont stockées, et cela peut être utilisé pour guider l'équilibrage de charge.

3. Amélioration du placement des tâches et des données grâce aux dépendances

Dans cette partie, nous décrivons comment le support exécutif peut avoir un impact positif sur l'exécution de l'application, en utilisant les informations présentes dans les dépendances.

3.1. La mécanique interne de XKAAPI

XKAAPI implémente un modèle d'exécution par vol de travail pour les programmes à base de tâches OpenMP, d'après le modèle présenté dans Cilk [7]. Les paragraphes suivant décrivent quelques mécanismes clés.
XKAAPI modélise la topologie de l'architecture comme une hiérarchie de `places`. Une `place` est en fait une liste de tâches associées à un ensemble de cœurs. La plupart du temps XKAAPI considère deux niveaux de places : les places de niveau NUMA (une place par nœud NUMA, regroupant l'ensemble des cœurs sur ce nœud), et les places au niveau du processeur (une place par cœur).
Le graphe de dépendances est construit, et lors d'un vol le voleur va choisir l'une des tâches dans la liste des tâches prêtes de la victime.
XKAAPI [1] base son exécution par vol de travail sur trois actions importantes :

- La *distribution initiale* des tâches prêtes
- La *sélection d'une place* dans laquelle pousser une tâche prête.
- La *sélection d'une victime* à voler lorsqu'un thread devient inactif.

Ces trois points mis ensemble définissent un algorithme d'ordonnancement dans XKAAPI.

3.2. Distribution initiale des tâches

Nous avons implémenté deux stratégies de distribution : `cyclicnuma`, qui distribue les tâches en round-robin sur les nœuds NUMA, et `randnuma`, aléatoirement. Il faut noter que contrairement à `numactl`, nos stratégies fonctionnent par blocs de données dans les clauses **depend**, et non pas par pages.

3.3. Sélection d'une place pour les tâches prêtes

Nous présentons quatre stratégies pour pousser les tâches prêtes dans une place du système NUMA. Deux d'entre elles ne prennent pas en compte les données, les deux autres si, via la clause OpenMP **depend** sur les tâches.
La stratégie `pLoc` permet au processeur de pousser localement les tâches prêtes. La stratégie `pLocNuma` permet au processeur de les pousser dans la place de son nœud NUMA local.
La stratégie `pNumaW` pousse les tâches sur la place NUMA où se trouve le premier bloc de

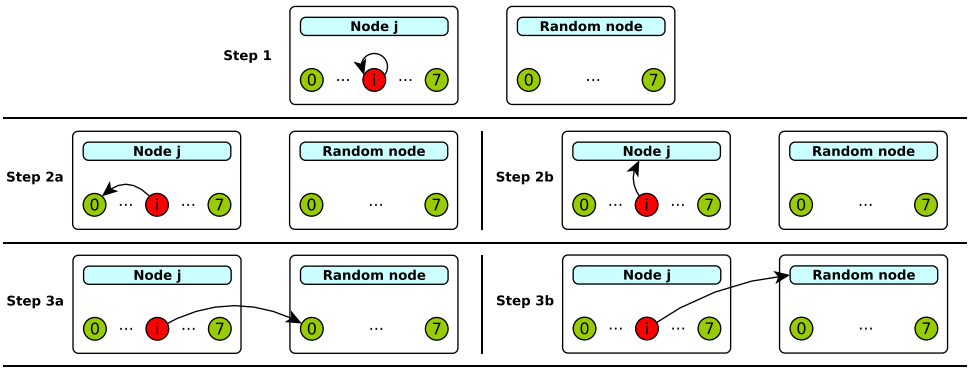

FIGURE 1 – Illustration de la stratégie de sélection *sProcNuma*

données écrit par la tâche (W est l'initiale de Write).
La dernière stratégie, `pNumaWLoc`, est similaire à `pNumaW`, mais si la donnée est allouée sur le nœud NUMA local, alors la tâche est poussé dans la place du processeur courant.

3.4. Sélection d'une victime lors du vol de travail

Nous avons implémenté des stratégies utilisant différents niveau de hiérarchie, afin de pouvoir étudier l'impact de celle ci sur la fonction de sélection.
Les deux premières, `sRand` et `sRandNuma`, sont similaires à celles étudiées dans [8]. Elle ne prennent en compte qu'un seul niveau de hiérarchie : `sRand` sélectionne une place processeur aléatoire, et `sRandNuma` sélectionne une place NUMA aléatoire.
Nous avons également implémenté plusieurs stratégies prenant en compte les deux niveaux de hiérarchie :

- `sProcNuma` : cette stratégie est illustrée dans la Figure 1. L'ordre de visite de la topologie est le suivant : d'abord on visite la place du processeur local (1), puis celles de ses voisins (2a), puis la place NUMA associée au processeur courant (2b). Si ces tentatives de vol échouent, un noeud est choisi au hasard sur la machine et les places de ses processeurs sont visitées (3a), avant que sa place NUMA soit visitée (3b).
- `sNumaProc` : cette stratégie est similaire, mais on visite les places NUMA avant de visiter les places processeur (inversion des étapes 2a et 2b, ainsi que 3a et 3b).
- `sProc` : le voleur ne visite que les places processeur et sa place NUMA locale (suppression de l'étape 3b).
- `sNuma` : le voleur ne visite que les places NUMA et sa place processeur (suppression des étapes 2a et 3a).

4. Évaluation de performances

Toutes nos expériences ont été effectuées sur la machine Intel192 décrite en partie 2. Nous avons évalué nos stratégies sur deux applications tirées des benchmarks KASTORS [13] [1] : une factorisation QR par bloc (`dgeqrf_taskdep`), et une factorisation de Cholesky par bloc (`dpotrf_taskdep`). Ces applications utilisent des routines BLAS et LAPACK fournies par OpenBLAS 2.15. Nous avons utilisé la bibliothèque OpenMP compatible avec GCC libKOMP [3] basée sur le support exécutif XKAAPI [2]. Nos traces et résultats sont également disponibles en ligne [3] afin que notre analyse soit reproductible par une tierce personne.

1. git disponible ici : https ://scm.gforge.inria.fr/anonscm/git/kastors/kastors.git, tag *paper-hierarchy*
2. https ://scm.gforge.inria.fr/anonscm/git/kaapi/kaapi.git, branche *viroulea/public/paper*
3. https ://github.com/viroulep/results-paper-hierarchy

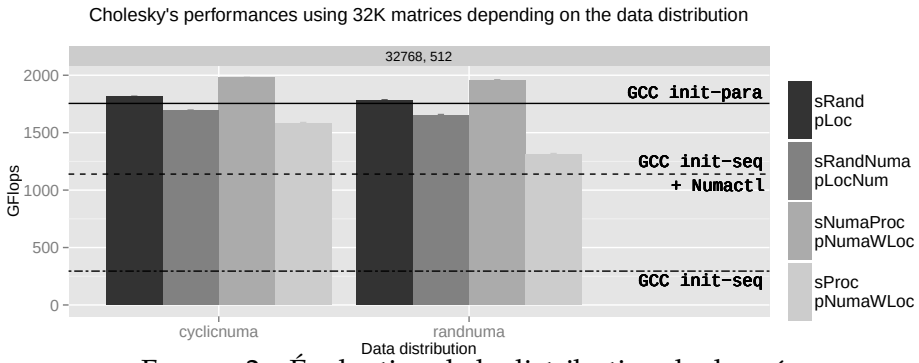

FIGURE 2 – Évaluation de la distribution de données

4.1. Impact de la distribution de données

Dans un premier temps nous avons évalué l'impact de la distribution de données sur les performance de la factorisation de Cholesky. Nous avons fait cette évaluation sur plusieurs tailles de matrices et tailles de bloc, ainsi que sur plusieurs combinaisons de stratégies de *sélection de victime* et de *sélection de place* pour les tâches prêtes. Nous avons comparé ces résultats aux performances de GCC [4] (via libGOMP), en utilisant, ou pas, une initialisation parallèle, ainsi que `numactl`.

La Figure 2 montre les résultats de cette évaluation. La conclusion de ces expériences est que l'initialisation parallèle des données est capitale.

Pour XKAAPI, l'utilisation d'une distribution contrôlée de données (`cyclicnuma`, `randnuma`) permet aux stratégies indépendantes de la hiérarchie de dépasser les meilleures performances de libGOMP, en revanche il n'y a pas de différence notable (à stratégie similaire), entre une distribution cyclique ou aléatoire, pour l'application considérée (cette conclusion est similaire pour les résultats de QR, non présents sur cette figure).

La distribution `cyclicnuma` sera utilisée comme stratégie par défaut pour les expériences suivantes, étant donné qu'elle offre de légèrement meilleures performances.

4.2. Aperçu des performances des différentes stratégies

À distribution de données fixée (`cyclicnuma`), nous avons comparé les différentes stratégies d'ordonnancement, caractérisées par un couple de stratégies de *sélection* de victimes, et de sélection de place pour *pousser* les tâches prêtes.

La Figure 3 montre les résultats des expériences pour la factorisation de Cholesky sur des matrices carrées de dimension 32768, divisées par bloc de 512×512 éléments (meilleure taille de bloc pour cette taille de matrice). La ligne en pointillés représente la performance du runtime de GCC, libGOMP, en utilisant une initialisation parallèle.

La première chose à noter est que même une combinaison de stratégies basique telle que `sRand+pLoc` (stratégie 1) obtient des performances acceptables grâce à la distribution de données. De plus, étant donnée une stratégie de sélection (e.g. `sRandNuma`), pousser les tâches sur le nœud NUMA où elle écrit des données (stratégie 7) offre de meilleures performances que simplement pousser les tâches sur son nœud NUMA (stratégie 2).

En considérant que les tâches sont poussées en utilisant la stratégie `pNumaWLoc`, ne prendre en compte qu'un niveau de hiérarchie (stratégies 3 ou 5) ne permet pas d'atteindre les performances obtenues avec des stratégies naïves. En revanche, prendre en compte les deux niveaux de hiérarchie (stratégies 4 et 6) permet d'obtenir de meilleures performances que les stratégies étudiées précédemment.

4. GCC 5.2.0

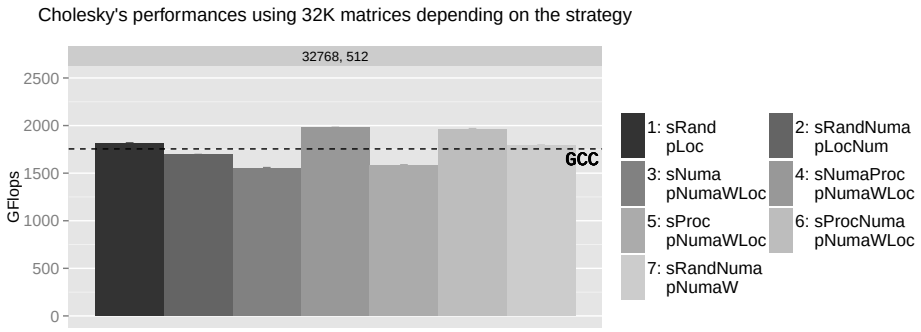

FIGURE 3 – Évaluation des différentes stratégies

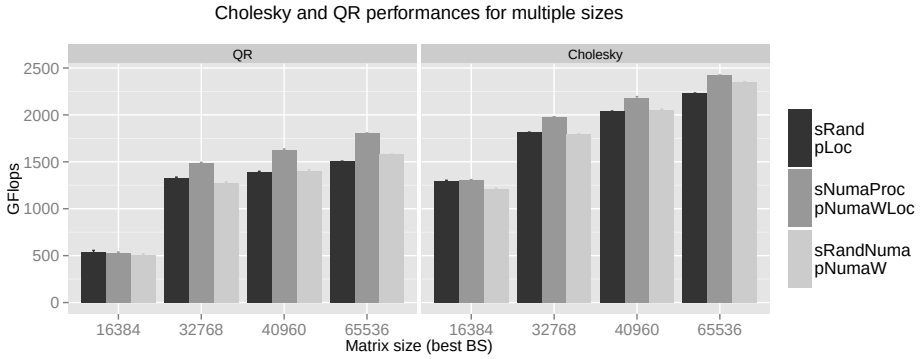

FIGURE 4 – Évaluation d'une sélection de stratégies sur plusieurs applications et tailles de matrice.

4.3. Scalabilité des performances des stratégies

Nous avons finalement sélectionné les trois combinaisons de stratégies offrant de meilleures performances que libGOMP pour évaluer leur scalabilité en fonction de la taille de la matrice en entrée, sur les applications de factorisation QR et de Cholesky. Il s'agit des stratégies 1, 4, et 7 présentent sur la Figure 3 (nous avons ignoré la 6, étant donnée qu'elle offrait des performances équivalentes à la 4, sans introduire une différence suffisante de stratégie). Nous avons choisi la meilleure taille de bloc pour une taille de matrice et une application donnée.

Ces résultats sont présentés dans la Figure 4. Sur des matrices de petites tailles l'avantage à utiliser des stratégies évoluées n'est pas flagrant. Cela peut être expliqué par le fait que le coût des stratégies hiérarchique est légèrement plus élevé que celui de la combinaison `sRand+pLoc`, et que le coût des communications induites par une petite taille de matrice n'est suffisant pour voir l'impact des stratégies hiérarchiques. En revanche dès une dimension de matrice 32768, l'utilisation d'une combinaison de stratégies qui prend en compte à la fois la localité des données et la hiérarchie de l'architecture (`sNumaProc+pNumaWLoc`), permet d'obtenir les meilleures performances par rapport aux deux autres stratégies, quelque soit l'application.

5. État de l'art

De nombreux travaux traitent de la localité et/ou de stratégie d'ordonnancement spécifiques aux architectures NUMA.

Clet-Ortega et al. [4] ont étudié et évalué plusieurs façons de décorer la topologie de l'archi-

tecture, en privilégiant des listes de tâches privées par thread, parcourues de manière hiérarchique. Nos étendons ces travaux en considérant également des listes au niveau des nœuds, et nous avons montré que cela peut aider à améliorer les performances sur des architectures NUMA.
Olivier et al. [8] ont évalué des stratégies hiérarchiques d'ordonnancement de tâches, en utilisant des structures centralisées ou distribuées. Notamment en créant un ensemble de threads, appelé *shepherd*, par nœud NUMA, permettant à l'ordonnanceur hiérarchique d'avoir de meilleures performances qu'avec les autres approches.
Tahan et al. [11] ont également étudié le comportement des programmes à base de tâches OpenMP sur les systèmes NUMA, en étendant le support exécutif NANOS avec deux ordonnanceurs de tâches appellés DFWSPT et DFWSRPT. Ils prennent en compte la notion de priorité de tâches, et essayent également de diminuer la distance à la mémoire lors de l'équilibrage de charge. D'autres travaux ont été faits dans le même contexte [12, 14], mais aucun ne tire parti de la clause OpenMP `depend`, qui indique précisément les données lues et écrites par une tâche donnée. Comme montré par notre combinaison de stratégies `sNumaProc+pNumaWLoc`, il est rentable de prendre en compte cette information lors de l'ordonnancement. Drebes et al. [5] ont caractérisé les problèmes de performances des algorithmes de vol de travail ignorant les spécificités des architectures NUMA, et proposé des améliorations mises en œuvre dans OpenStream, leur support exécutif à base de tâches.

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 plusieurs stratégies d'exécution pour affecter efficacement les tâches prêtes aux listes de tâches représentant l'architecture NUMA. Ces stratégies contrôlent à la fois la manière dont les tâches prêtes sont initialement réparties, ainsi que la manière dont les tâches sont volées. Nous avons évalué plusieurs distributions de données, et évalué différentes combinaisons de stratégies "push" et "select" sur un système NUMA de 192 cœurs, en utilisant des applications d'algèbre linéaire. Les meilleures performances sont obtenues avec les stratégies prenant en compte à la fois le placement initial des données et la hiérarchie de la machine.
À court terme nous prévoyons d'étendre le compilateur pour indiquer de manière compatible avec OpenMP quelles sont les tâches d'initialisation.
Sur le long terme nous prévoyons de nous concentrer sur certaines techniques de compilation qui pourraient nous permettre d'évaluer des informations importantes sur les tâches, comme leur intensité opérationnelle, qui pourrait aider à la prise de décision au sein du support exécutif. Nous pensons que la coopération entre le compilateur et le support exécutif peut améliorer significativement les performances et la scalabité des applications à base de tâches.

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.). – Task-parallel programming on NUMA architectures. – In *20th International Conference, Euro-Par 2014, Porto, Portugal, August 25-29, 2014. Proceedings*, 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, Berlin, Heidelberg, 2012. Springer-Verlag.
4. Clet-Ortega (J.), Carribault (P.) et Pérache (M.). – Evaluation of openmp task scheduling algorithms for large NUMA architectures. – In *20th International Conference, Euro-Par 2014, Porto, Portugal, August 25-29, 2014. Proceedings*, *LNCS*, volume 8632, pp. 596–607. Springer, 2014.
5. 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 Transactions on Architecture and Code Optimization*, vol. 11, n3, octobre 2014, p. 30.
6. 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, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
7. Frigo (M.), Leiserson (C. E.) et Randall (K. H.). – The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, vol. 33, n5, mai 1998, pp. 212–223.
8. Olivier (S.), Porterfield (A.), Wheeler (K. B.), Spiegel (M.) et Prins (J. F.). – Openmp task scheduling strategies for multicore NUMA systems. *IJHPCA*, vol. 26, n2, 2012, pp. 110–124.
9. 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*, *SC '12*, SC '12, pp. 65 :1–65 :12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
10. OpenMP Architecture Review Board. – OpenMP application program interface version 4.0, juillet 2013.
11. Tahan (O.). – Towards efficient openmp strategies for non-uniform architectures. *CoRR*, vol. abs/1411.7131, 2014.
12. Terboven (C.), Schmidl (D.), Cramer (T.) et an Mey (D.). – Task-parallel programming on NUMA architectures. – In *18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012*, *LNCS*, volume 7484, pp. 638–649. Springer, 2012.
13. 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, Salvador, Brazil, septembre 2014. Springer.
14. Wittmann (M.) et Hager (G.). – Optimizing ccNUMA locality for task-parallel execution under openmp and TBB on multicore-based systems. *CoRR*, vol. abs/1101.0093, 2011.