Moniteurs hiérarchiques de performance, pour gérer l'utilisation des ressources partagées de la topologie

Nicolas Denoyelle

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

Moniteurs hiérarchiques de performance, pour gérer l'utilisation des ressources partagées de la topologie

Nicolas Denoyelle

Inria Bordeaux - Sud-Ouest – Université de Bordeaux

Résumé
L'avènement des machines multicœurs et manycœurs a permis en partie de soutenir la croissance de la performance des machines de calcul. Cependant, l'accroissement du nombre d'unités d'exécution n'est pas nécessairement synonyme de réduction de la durée d'exécution. Les différentes tâches d'une application peuvent entre autre s'interrompre pour envoyer ou attendre des messages (synchronisations) ou concourir pour utiliser une ressource matérielle partagée dans la machine. Lorsque la modification du code n'est pas permise (e.g dans un support d'exécution), le placement des tâches et des données permet de surmonter en partie ces problèmes, respectivement en minimisant le chemin des communications, ou en équilibrant la charge sur la machine. Dans ces deux cas, il est nécessaire d'avoir un modèle de l'architecture et des métriques sur l'utilisation de celle-ci, pour espérer caractériser le couple (machine, application) et calculer un placement efficace.
Dans ce contexte, nous proposons un outil de mesure de performances, permettant d'agréger des évènements collectés au cours du temps sur des noeuds de la topologie d'une machine afin d'en tirer une analyse couplée du programme et de la machine. Cet outil est basé sur un modèle d'architecture fourni par hwloc et des greffons de collecte d'évènements (implémentés avec papi et maqao) et permet l'analyse d'applications parallèles sur un système. L'outil proposé ne réalise pas directement cette analyse mais propose des mécanismes pour y parvenir. Un utilitaire permet d'afficher la topologie et l'évolution des évènements ainsi que de générer une trace des évènements collectés, accompagnés de leur localisation, au cours du temps. Nous montrons avec une application simple écrite pour l'occasion qu'il est possible d'utiliser les résultats fournis par notre outil pour déduire un placement des fils d'exécution plus efficace.

1. Introduction

Alors que la hiérarchie des processeurs et des systèmes devient de plus en plus large et profonde, la localité des données devient une condition critique pour le passage à l'échelle des applications. On peut améliorer cette localité, notamment à travers le placement des tâches et des données sur la machine. Cependant, déterminer un placement optimal reste un problème difficile, en particulier parce que le matériel est d'une complexité croissante et car les affinités entre les tâches sont nombreuses et difficile à capturer. Plusieurs travaux [8, 6, 11] démontrent que la façon de gérer les affinités a un impacte non négligeable sur la performance des applications.
L'identification des affinités pour placer les tâches nécessite le support matériel et logiciel permettant de caractériser une exécution à travers des métriques adaptées. Par exemple pour un

programme qui exécute un nombre d'instructions fixe, on peut mesurer le nombre d'instructions traitées par cycle de processeur pour attester dynamiquement de l'efficacité d'une optimisation. Ce nombre peut être en partie le résultat d'accès à une hiérarchie de ressources matérielles partagées (accès à la mémoire, au cache *etc.*), de même que l'utilisation d'une de ces ressources (par exemple le dernier niveau de cache) peut elle-même être liée à l'utilisation de ressources filles (caches intermédiaires). Pour comprendre de telles interactions, nous pensons qu'il est nécessaire de pouvoir faire le lien entre des mesures que nous obtenons des applications et le matériel sur lequel elles s'exécutent.

Nous proposons donc un logiciel permettant d'abstraire la collecte d'évènements à l'exécution (compteurs de performance, données de l'architecture, *etc.*), et de les exposer de manière synthétique afin d'aider à détecter et à comprendre les problèmes de localité pouvant survenir à l'exécution. Dans notre preuve de concept [1] nous reprenons le modèle d'architecture issue du logiciel Hardware Locality (hwloc [2]) et amplifions sa représentation graphique avec les informations de performance obtenues à l'exécution.

Celle-ci permet la détection visuelle de problèmes de localité, et la bibliothèque qui l'accompagne, fourni des outils logiciels utiles aux algorithmes de placement.

La suite de ce papier est organisée de la manière suivante : La section 2 décrit un bref état de l'art, La section 3 détaille les buts, et les fonctionnalités de l'outil proposé. Un cas d'utilisation est finalement étudié dans la section 4.

2. État de l'art

L'analyse de performance est un champ extrêmement actif en HPC. De nombreux outils de complexité variable existent, et sont parfois complémentaires. On peut généralement les diviser en deux catégories :

— les outils basés sur l'échantillonnage qui collectent des évènements à intervalle de temps régulier,
— et les outils basés sur l'annotation ou l'instrumentation de code qui collectent des évènements pour des segments d'instructions tels que des fonctions.

Comme le font MAQAO [1], PIN [10] ou bien Intel VTune Amplifier, il est possible d'effectuer des analyses de la granularité du processus entier à la granularité de l'instruction. Certains sont efficaces pour détecter des goulets d'étranglement logiciels (synchronisations, implémentation sous optimal d'un noyau de calcul *etc.*), mais font rarement cas de la compétition pour une ressource matérielle. Par exemple, il est difficile de savoir si une donnée est trop longue à charger parce que le bloc de données ne tient pas dans le cache ou parce qu'il a été évincé par le matériel au profit d'une autre tâche.

Nous préférons nous concentrer sur cette seconde approche, car nous cherchons à caractériser une affinité entre tâches, composées des données qu'elles partagent(affinité positives) et des ressources limitantes qu'elles exploitent en commun (affinité négatives) dans le but de faire du placement de tâches. S'il est nécessaire de migrer des tâches à la volée, cela est également très coûteux et doit être effectué avec parcimonie et à basse fréquence. Pour cela nous choisissons une approche à gros grain. Nous proposons également, d'analyser les échantillons à la volée afin de générer une quantité moindre de données .

En général les observations à cette granularité, nécessitent quand même une analyse temporelle dynamique. Cette analyse peut impliquer des outils en temps réel tels que numatop ou tiptop [13]. Les traces composées d'échantillons rapprochés sur une longue durée et un grand

1. `https://github.com/NicolasDenoyelle/dynamic_lstopo/`

nombre de tâches sont extrêmement volumineuses en espace et longues à analyser avec des outils d'analyse post-exécution tels que VampirTrace [9]. Notre travail peut également se baser sur ces approches mais nous mettons l'accent sur la présentation des données afin de renvoyer une vue localisée de celles-ci, dynamiquement, ou après l'exécution.
Une solution courante pour mesurer le comportement d'une application à l'exécution est de lire les compteurs de performance du processeur. De nombreux outils tels que PAPI [3] ou l'utilitaire Linux perf proposent d'effectuer ce travail. Notre outil est pour le moment centré sur une analyse intra-nœud, mais son principe peut être étendu à l'échelle du système. D'autres outils tels que SCALASCA [4] ou Paraver [12] permettent une analyse à l'échelle d'un cluster pour mesurer par exemple des congestions dans les switchs.
L'association de données de performance avec les données du matériel, n'est pas encore un sujet populaire, alors que la croissance de la taille des machines et de leur complexité en fera sans doute un sujet à surveiller. En effet, la détection de la topologie et de certaines caractéristiques matérielles n'a que récemment été maîtrisée par des outils tels que hwloc [2]. Les approches antérieures étaient souvent moins portables et n'exposaient pas autant de détails sur le partage des caches *etc.*
MemAxes [5] permet une analyse à grain fin des performances avec une vue graphique radiale de la hiérarchie mémoire. Mais cette approche reste statique, après l'exécution, et se limite aux accès à la mémoire. Notre approche se veut dynamique et propose d'abstraire la collecte d'évènements de performance pour permettre d'intégrer n'importe quelle mesure et n'importe quelle analyse. De plus notre modèle de topologie repose sur une bibliothèque de détection éprouvée (hwloc est intégrée dans Slurm et OpenMPI) qui détecte plus de ressources partagées.

3. Moniteurs hiérarchiques de performance

Nous définissons des moniteurs (au sens défini par Raj Jain [7] :Partie 2, chapitre 7), pour représenter une métrique de performance localisée. Il collecte des évènements, les analyse et affiche un résultat. Observer le comportement du système respectivement à son utilisation par une application, à l'aide de moniteurs permet de mesurer l'utilisation des ressources dans le but de trouver des goulets d'étranglements, caractériser une exécution, optimiser l'utilisation du système *etc.* Nous avons donc construit une bibliothèque implémentant des moniteurs localisés sur les noeuds de la topologie de la machine, dans le but de caractériser le comportement local d'une application, afin de l'optimiser ou d'aider à sa modélisation.

Dans ce paragraphe nous décrivons les caractéristiques des moniteurs.
De manière synthétique, un moniteur est composé d'une localisation sur la machine et d'un ensemble d'évènements à collecter.
Les évènements de chaque moniteur sont sauvegardés dans une mémoire tampon et écrits sur la sortie lorsque le tampon est plein.
Ajouté à cela, il est possible d'analyser les données à la volée (avec un système de greffons) pour réduire encore le coût des entrées/sorties ou bien pour optimiser dynamiquement une exécution en fonction des valeurs dans les moniteurs.
Le domaine d'activités observable par les moniteurs n'est pas limité en théorie. Il existe une ABI pour greffer n'importe quelle bibliothèque capable d'enregistrer des évènements, et nous en avons implémentées deux (avec papi et maqao) pour mesurer des compteurs matériels.
La résolution [2] des moniteurs n'est pas mesurée car elle dépend du domaine d'activité de ceux-

2. fréquence d'échantillonnage maximum

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
8410	22110	22330	23520	22740	818800

FIGURE 1 – "Résolution d'un moniteur vide (en nanosecondes) sur 41827 échantillons

ci. En pratique on peut mesurer celle du moniteur avec un domaine d'activité vide. Nous avons donc observé un tel moniteur [3] et mesuré la fréquence d'échantillonnage maximale donnée par la trace de sortie. Le tableau 1 montre une résolution à peu près stable à 23 microsecondes. Il n'est donc pas raisonnable d'espérer observer à une faible granularité, d'autant que sur une architecture large et profonde, le coût de synchronisation et d'écriture des moniteurs est plus grand.

De même le surcoût d'exécution des moniteurs dépend du domaine d'activité, il est donc par exemple très probable qu'un compteur d'accès au premier niveau de cache soit largement pollué par l'exécution des moniteurs.

Pour maîtriser ce surcoût nous avons implémenté un utilitaire dont la fréquence d'échantillonnage est paramétrable ainsi qu'une bibliothèque pour pouvoir restreindre la mesure à une portion de code sans faire d'échantillonnage.

En pratique, la description des moniteurs pour une exécution, se fait par un fichier de description au format suivant :

```
L3_miss_per_cycle{
OBJ:=PU;
PERF_LIB:=papi;
EVSET:=PAPI_L3_TCM, PAPI_TOT_CYC;
EVSET_REDUCE:=$0/$1;
SAMPLES_REDUCE:=MONITOR_SAMPLES_LAST;
N_SAMPLES:=128;
}
```

Le champ `OBJ` donne la profondeur à laquelle le moniteur sera. Un moniteur par objet à cette profondeur est alors créé. `PU` désigne la plus petite unité d'exécution, aux feuilles de la topologie dans hwloc.

Le champ `PERF_LIB` donne l'implémentation du domaine du moniteur à charger. Outre les implémentations avec les bibliothèques papi et maqao, nous avons implémenté l'ABI du domaine avec des moniteurs hiérarchiques, afin d'accumuler les évènements obtenus aux feuilles dans les niveaux supérieurs. On peut par exemple voir le détail des défauts de cache par cache (et non par PU) de cette manière.

Le champ `EVSET` donne les évènements du domaine à acquérir.

Le champ `EVSET_REDUCE` sélectionne l'opérateur de réduction sur les évènements.

Le résultat de cette réduction constitue un échantillon. (Ce n'est pas la valeur du moniteur).

Le champ `N_SAMPLES` donne le nombre d'échantillons mis en mémoire tampon. Si la méthode de sortie est celle temporisée, alors la sortie est écrite uniquement lorsque le tampon est plein. Cela permet également d'appliquer un traitement sur les évènements tout en ayant une connaissance du passé proche.

Le champ `SAMPLES_REDUCE` sélectionne l'opérateur de réduction sur les échantillons. La valeur retournée par cette opérateur est également celle du moniteur. Cet opérateur est particuliè-

3. en implémentant l'ABI avec des fonctions minimalistes

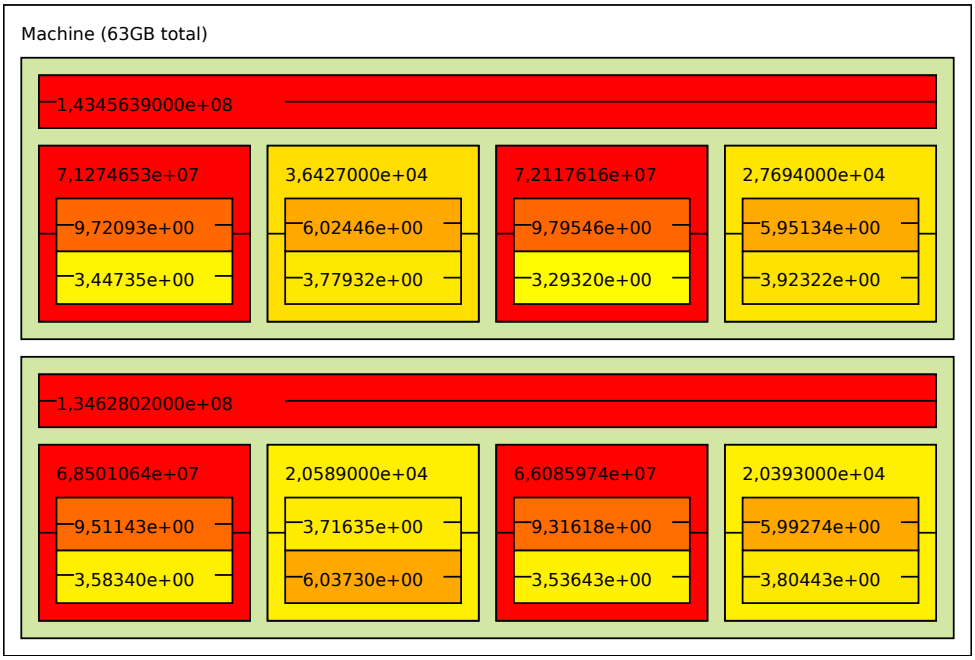

FIGURE 2 – Sortie lstopo(hwloc) augmentée de moniteurs.

rement important car c'est celui qui réalise l'analyse du moniteur. Plusieurs opérateurs simples sont implémentés(somme, maximum, minimum) mais il est possible d'en créer de plus complexes dans une fonction au prototype suivant : `double call(struct monitor *)`. Une telle fonction compilée en bibliothèque dynamique peut alors être chargée par le moniteur et appelée à chaque mesure. La structure du moniteur étant ouverte, il est possible d'utiliser ses attributs (historique d'évènements, échantillons *etc.*) et d'y stocker des données.
Les données collectées puis agrégées dans chaque moniteur sont ensuite transcrites dans un fichier de trace et/ou affichées sur la topologie. La figure 2 illustre une telle sortie.
Un utilitaire permet d'échantillonner et de synchroniser le déclenchement des moniteurs. Outre le fait de mesurer l'activité du système, il est également possible de restreindre les moniteurs à la fois au domaine d'exécution d'une application, ainsi qu'au domaine où elle est effectivement active. Enfin, il est possible d'utiliser une bibliothèque pour suivre des portions d'une application en l'encadrant de fonction pour déclencher des mesures et de placer des marqueurs dans la trace pour identifier le code correspondant.
La figure 3 décrit l'interaction de notre bibliothèque avec hwloc, l'application et les greffons de performance(domaine du moniteur) et d'analyse.

4. Déduction du placement des fils d'exécution par l'analyse de la concurrence sur les caches

Dans cette section, nous présentons un cas de mauvais placement de tâches observable. Pour cette exemple, l'outil suit le comportement de toute la machine.

À l'échelle d'un noeud, l'efficacité de l'ordonnancement de fils d'exécution sous un cache dépend à la fois de la pression qu'ils mettent sur le cache, et de la distance de réutilisation des données qu'ils partagent. Deux solutions souvent discutées sont soit d'éparpiller les fils d'exécution pour équilibrer la pression sur les caches partagés, soit de les rassembler pour accélérer

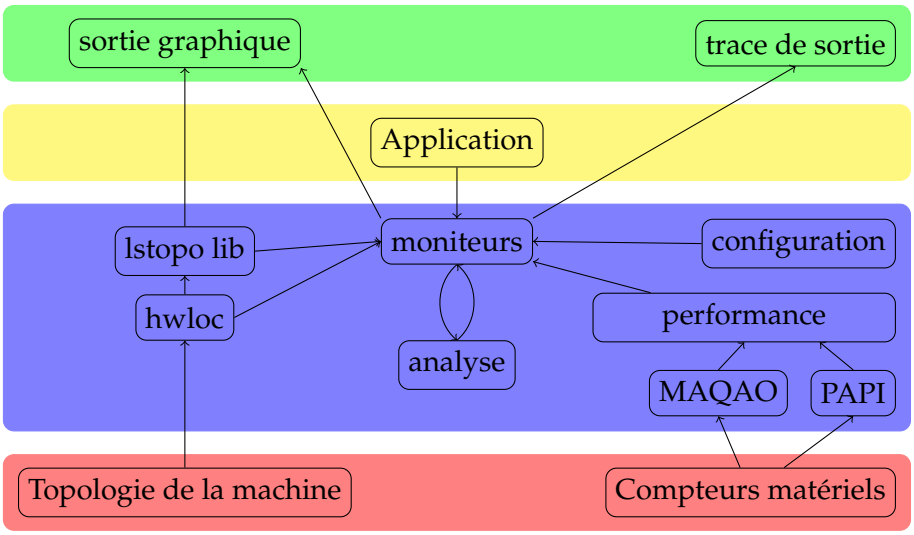

FIGURE 3 – architecture de la bibliothèque

l'accès aux données partagées. [11]. Ces deux politiques de placement sont souvent utilisées et implémentées dans la plupart des supports d'exécution OpenMP. Mais le choix de l'une ou de l'autre reste à la discrétion de l'utilisateur.
Certains travaux utilisent le ratio de conflits de cache comme métrique pour mesurer la pression sur un cache [14] et décider de la politique de placement.

En se basant sur cette observation, nous avons construit une application capable de changer arbitrairement la pression mise par chaque fil d'exécution sur le dernier niveau cache. Il s'agit d'un parcours de liste aléatoirement chaînée.
Soit n le nombre d'éléments de la liste et k le nombre d'éléments qui peuvent tenir dans le cache.
Si $n < k$ alors la liste tient dans le cache, et on ne devrait voir que peu de défauts de cache.
Si $n > k$, on peut écrire $n = n' + k$ et donc la probabilité d'accéder à un élément en dehors du cache est $\frac{n'}{n}$, c'est à dire le rapport du nombre d'éléments qui dépassent du cache sur le nombre total d'éléments.
On voit que pour avoir un taux élevé de défauts de cache il faut une liste bien plus grande que celui-ci.

L'application crée un fil d'exécution par cœur sur la machine. La première moitié instancie chacun une liste longue (16* la taille du dernier niveau de cache (LLC)) tandis que l'autre moitié instancie chacun une liste courte(de la taille du premier niveau de cache (L1)). La machine sur laquelle se déroule l'expérience est intel Xeon E5-2650 composée de 2 caches LLC et de 8 cœurs par LLC.
On observe deux scénarios de placement, dans le premier les listes longues partagent le même cache et les listes courtes partagent l'autre cache (placement regroupé), tandis que dans le second scénario les listes sont équitablement réparties entre les caches (placement équilibré).
Chaque fil d'exécution parcourt un nombre fixe de fois sa liste, ceux ayant une liste courte la parcourent $16 * \mathtt{taille}(\mathrm{LLC})/\mathtt{taille}(\mathrm{L1})$ fois de plus pour équilibrer la durée d'exécution de chaque fil.

Cache	cache miss	cycles sans instruction	microsecondes	placement
L3Cache :0 L3Cache :1	21767764300 22064689720	696207553 698293586	15976767	équilibré
L3Cache :0 L3Cache :1	52472419090 218502	1480721113 1297	17514009	regroupé

FIGURE 4 – moniteurs sur le cache en fonction du scénario

Le tableau 4 montre que le scénario qui équilibre les défauts de cache est plus rapide à exécuter. Les moniteurs permettent de constater quel est le cache le plus sollicité grâce aux compteurs de cache miss. Le détail par cœur n'apparaît pour des raisons de place, mais il permet de vérifier le placement. Ici, comme le code est uniquement composé d'accès mémoire et que les accès à une mémoire distante sont lents, on vérifie également que le processeur est moins sollicité lorsqu'il y a beaucoup de défauts de cache.

5. Conclusion et perspectives

Le chemin vers l'ère exascale nécessite une conception minutieuse des supports d'exécution parallèles en accord avec les affinités logicielles et la localité physique du système, de sorte que les tâches et leurs données soient associées au mieux. L'analyse de la performance et son association avec les informations de la topologie est cruciale pour comprendre et optimiser l'utilisation des ressources. Dans cet article, nous avons présenté une bibliothèque de moniteurs, capable de rassembler des informations de performance de diverses sources, des informations de la topologie depuis hwloc, et de les agréger pour fournir la donnée utile. Les résultats sont stockés dans une trace qui peut être relue ou réinterprétée avec la bibliothèque. Nous avons montré avec une application bas-niveau et une métrique pertinente que l'association de mesures de performance avec la topologie de la machine peut révéler des problèmes de localité.

Plusieurs voies d'évolutions sont envisagées pour ce travail : D'abord, la visualisation par lstopo n'est actuellement pas adaptée aux systèmes composés de beaucoup de noeuds. On peut donc envisager de représenter la topologie, de manière radiale, à l'instar de MemAxes. Ensuite, le procédé qui consiste à suivre un processus, est peu précis et bénéficierait une extension avec un module communiquant avec l'ordonnanceur système pour mesurer à grain fin les tranches de temps allouées à l'application. Finalement, nous projetons d'utiliser cet outil d'analyse, à des fins d'ordonnancement dynamique hiérarchique et multi-critères de tâches et de caractérisation des applications, et de leur phases.

Bibliographie

1. Barthou (D.), Charif Rubial (A.), Jalby (W.), Koliai (S.) et Valensi (C.). – Performance Tuning of x86 OpenMP Codes with MAQAO. *In : Tools for High Performance Computing 2009*, éd. par Müller (M. S.), Resch (M. M.), Schulz (A.) et Nagel (W. E.), pp. 95–113. – Springer Berlin Heidelberg, 2010.
2. Broquedis (F.), Clet-Ortega (J.), Moreaud (S.), Furmento (N.), Goglin (B.), Mercier (G.), Thibault (S.) et Namyst (R.). – hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications. – In IEEE (édité par), *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italie, février 2010.

3. Browne (S.), Dongarra (J.), Garner (N.), London (K.) et Mucci (P.). – A scalable cross-platform infrastructure for application performance tuning using hardware counters. – In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, *SC '00*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.
4. Geimer (M.), Wolf (F.), Wylie (B. J. N.), Ábrahám (E.), Becker (D.) et Mohr (B.). – The SCALASCA performance toolset architecture. – In *Proc. of the International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece*, pp. 51–65, June 2008.
5. Gimenez (A.), Gamblin (T.), Rountree (B.), Bhatele (A.), Jusufi (I.), Bremer (P.-T.) et Hamann (B.). – Dissecting On-Node Memory Access Performance : A Semantic Approach. – In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pp. 166–176, New Orleans, LA, novembre 2014. IEEE Press.
6. Hursey (J.), Squyres (J. M.) et Dontje (T.). – Locality-aware parallel process mapping for multi-core hpc systems. – In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pp. 527–531. IEEE, 2011.
7. Jain (R.). – *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*,. – mai 1991, xxvii + 685p. Winner of "1991 Best Advanced How-To Book, Systems" award from the Computer Press Association.
8. Jeannot (E.), Mercier (G.) et Tessier (F.). – Process placement in multicore clusters : Algorithmic issues and practical techniques. *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, n4, 2014, pp. 993–1002.
9. Knüpfer (A.), Brunst (H.), Doleschal (J.), Jurenz (M.), Lieber (M.), Mickler (H.), Müller (M. S.) et Nagel (W. E.). – The vampir performance analysis tool-set. – In *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pp. 139–155, 2008.
10. Luk (C.-K.), Cohn (R.), Muth (R.), Patil (H.), Klauser (A.), Lowney (G.), Wallace (S.), Reddi (V. J.) et Hazelwood (K.). – Pin : Building customized program analysis tools with dynamic instrumentation. – In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, *PLDI '05*, PLDI '05, pp. 190–200, New York, NY, USA, 2005. ACM.
11. Majo (Z.) et Gross (T. R.). – Memory management in numa multicore systems : Trapped between cache contention and interconnect overhead. *SIGPLAN Not.*, vol. 46, n11, juin 2011, pp. 11–20.
12. Pillet (V.), Labarta (J.), Cortes (T.) et Girona (S.). – PARAVER : A Tool to Visualize and Analyze Parallel Code. – In Nixon (P.) (édité par), *Proceedings of WoTUG-18 : Transputer and occam Developments*, pp. 17–31, mar 1995.
13. Rohou (E.). – *Tiptop : Hardware Performance Counters for the Masses*. – Research Report n RR-7789, novembre 2011.
14. Zhuravlev (S.), Blagodurov (S.) et Fedorova (A.). – Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, vol. 45, n3, mars 2010, pp. 129–142.