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

Détection automatique d'interférences entre threads

Mohamed Saïd Mosli Bouksiaa, François Trahay, Gaël Thomas

CNRS Samovar
Télécom SudParis
Université Paris-Saclay
Email : prenom.nom@telecom-sudparis.eu

Résumé
Comprendre les performances d'une application multi-threadée peut s'avèrer difficile à cause des interférences entre les threads. Alors que certaines interférences sont prévisibles (par exemple, l'acquisition d'un verrou), d'autres sont plus subtiles (par exemple, le faux partage) et complexe à détecter. Dans cet article, nous proposons une méthodologie et une métrique permettant de détecter les interférences entre des threads et d'en quantifier l'impact sur les performance globales de l'application. Cette méthodologie consiste à étudier la variation de la durée d'exécution d'une portion de code. Nous avons appliqué cette méthodologie à un ensemble de micro-benchmarks et d'applications. Les résultats montrent que cette méthodologie permet effectivement de détecter les interférences entre les threads d'une application.

Mots-clés : multithreading ; analyse de performance ; contention ; faux partage

---

1. Introduction

Les applications multi-threadées sont largement utilisées dans de nombreux domaines d'applications (HPC, big data, cloud, etc). Optimiser les performances de ces applications est une tâche difficile. En effet, quand des threads différents partagent une ressource, cela peut entrainer des interférences notamment lorsque plusieurs threads essaient d'accèder à la ressource en même temps. Une interférence entre threads peut être due à une synchronisation volontaire (prise d'un verrou pour accèder à une structure de données partagée, opération atomique sur une variable partagée, etc) comme elle peut être involontaire (partage du cache, partage d'un bus ou d'un controlleur, faux-partage, etc).
Explicitement ou implicitement, les synchronisations qui régissent l'accès aux ressources partagées entrainent des attentes au niveau des threads, ce qui conduit à la sérialisation de l'exécution et donc à une dégradation importante des performances. Il est donc indispensable de pouvoir réduire les interférences entre les threads [4, 8, 9] afin d'optimiser les applications multi-threadées. Mais pour ce faire, il faut commencer par détecter ces interférences et comprendre leur impact sur la performance. Actuellement, à notre connaissance, il n'existe pas de méthode unique qui permet d'identifier l'ensemble de ces interférences.
Pour faciliter la détection des interférences entre threads et d'en estimer le surcoût, nous proposons une métrique unifiée que nous appelons *facteur de dispersion* et qui vise à détecter différents types d'interférences. L'idée derrière cette métrique est qu'une séquence d'opérations peut soit être exécutée normalement, soit être ralentie à cause d'une interférence avec un autre thread.

Une séquence d'opérations qui s'exécute de manière répétitive a de fortes chances de s'exécuter normalement au moins une fois, étant donné le non-déterminisme des interférences. Cela entraine une variation dans les temps d'exécution d'une même séquence. Le rôle du facteur de dispersion est de mesurer ces variations ainsi que leur impact sur le temps d'exécution de l'application.
Nous avons utilisé 3 micro-benchmarks qui représentent différents types d'interférence (faux-partage, spinlock, contention sur un verrou) afin de valider notre métrique sur des cas simples. En même temps, ces micro-benchmarks ont servi à calibrer le facteur de dispersion.
Pour chaque micro-benchmark, nous avons fait varier la fréquence d'apparition des interférences entre threads et mesuré le facteur de dispersion des séquences d'opérations intéressantes. Ceci nous a permis d'observer l'évolution du facteur de dispersion en fonction de la fréquence des interférences. Nous avons ensuite testé les résultats de cette phase de calibrage sur la suite d'application Splash-2. Les résultats montrent que :

— Empiriquement, le facteur de dispersion admet la valeur 0.3 comme seuil pouvant indiquer la présence d'interférences problématiques entre threads.
— Le facteur de dispersion nous a permis d'identifier les applications de Splash-2 qui souffrent de contention au niveaux des verrous. La suppression de cette contention (remplacement de la stratégie de lock POSIX par RCL) baisse significativement la valeur du facteur de dispersion, souvent en dessous du seuil empirique de 0.3.

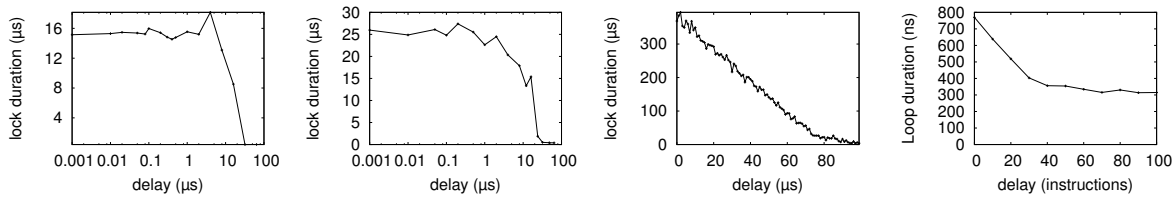
La suite de l'article est organisée comme suit : la section 2 explique l'utilisation du facteur de dispersion. La section 3 détaille la phase de calibrage en utilisant des micro-benchmarks. La section 4 présente les résultats obtenus sur les applications de la suite Splash-2. La Section 5 présente un bref état de l'art et la dernière section conclut l'article.

2. Utilisation du facteur de dispersion

La première étape consiste à identifier les parties de l'application consommant le plus de temps CPU. Cela peut se faire à l'aide d'un *profiler* tel que Gprof [6] ou Oprofile [7]. Les principales fonctions en terme de temps d'exécution sont ensuite instrumentées avec un outil de collecte de traces tel qu'EZTrace [14] ou VampirTrace [10] et une trace d'exécution de l'application est générée. La trace générée est ensuite analysée afin d'en extraire les motifs d'événements répétitifs [13] ou les appels de fonction.
Chaque motif est une séquence d'opérations qui s'exécute plusieurs fois et admet donc un facteur de dispersion qu'on calcule à l'aide de la formule 1

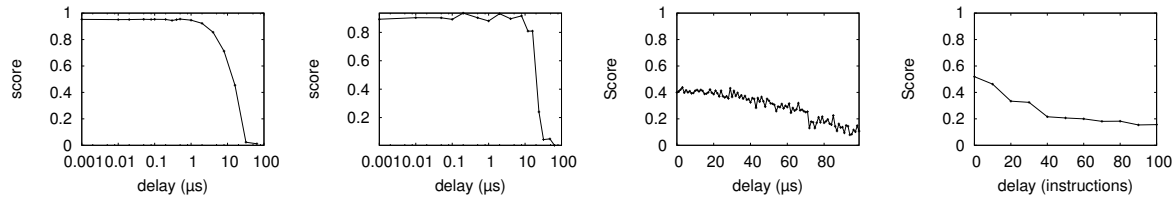
$$score = \frac{\sum_{i=1}^{N}(x_i - x_1)}{T_{thread}} \quad (1)$$

où N est le nombre d'occurences d'un motif, x_i est la durée de l'occurence i du motif, x_1 sa durée minimale, et T_{thread} la durée totale d'exécution du thread. Une fois que les facteurs de dispersion de tous les motifs ont été calculés, les motifs dont le score est supérieur à un seuil prédéfini sont sélectionnés afin que le développeur les examine.
L'analyse de quelques centaines de traces représentant plusieurs Gio de données nous a permis de définir empiriquement ce seuil à 0.3. Ainsi, seuls les motifs dont les variations sont responsables de 30 % de la durée totale d'exécution du thread sont présentés au développeur pour qu'il les examine.



(a) Mutex contention (b) Spinlock contention (c) Spinlock (d) Faux-partage

FIGURE 1 – Évolution des performances des micro-benchmarks



(a) Mutex contention (b) Spinlock contention (c) Spinlock (d) Faux-partage

FIGURE 2 – Évolution des scores des micro-benchmarks

3. Application à des micro-benchmarks

Afin d'évaluer la méthodologie proposée, nous l'appliquons, dans un premier temps, à un ensemble de micro-benchmarks. Les résultats de cette évaluation sont reportés dans les figures 1 et 2. Cette Section décrit les micro-benchmarks ainsi que les résultats obtenus.

3.1. Contention sur un verrou

Ce micro-benchmark consiste à créer N threads qui, une fois synchronisés, exécutent le code présenté Listing 1 : pour chaque itération, chaque thread calcule pendant `delay` µs, acquiert un mutex, incrémente une variable globale, puis relache le mutex.

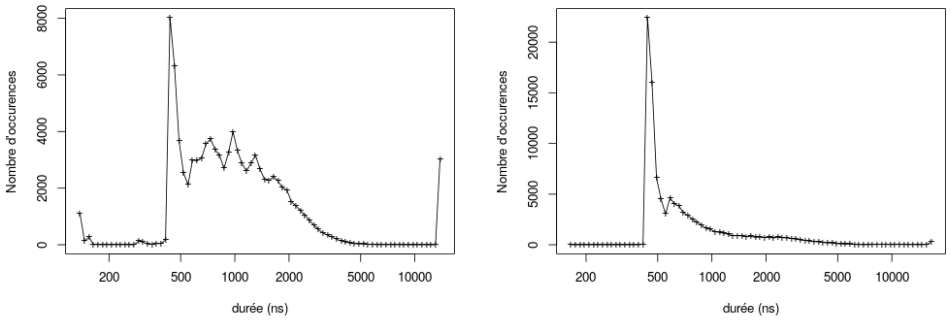
Listing 1 – Code du micro benchmark Mutex contention

```
for(i=0; i<NITER; i++) {
  compute(delay);
  pthread_mutex_lock(&l);
  value++;
  pthread_mutex_unlock(&l);
}
```

Listing 2 – Code du micro benchmark spinlock

```
for(i=0; i<NITER; i++) {
  compute(delay);
  pthread_spin_lock(&l);
  compute(100-delay);
  pthread_spin_unlock(&l);
}
```

Nous avons exécuté ce micro-benchmark sur la machine `amd48b`, équipée de 48 cœurs répartis sur 4 nœuds NUMA constitués chacun d'un processeur dodéca-cœurs AMD Opteron 6172. Le nombre de threads s'exécutant en parallèle est fixé à 47 et nous faisons varier `delay` de 0 à 100 µs afin de faire varier la contention sur le mutex. Pour chaque valeur de `delay`, nous calculons le facteur de dispersion du motif qui correspond à la section critique + prise de verrou. La durée moyenne de la prise de verrou en fonction de `delay` est présentée dans la Figure 1a. Le facteur de dispersion obtenu est présenté dans la Figure 2a. Les résultats montrent que



(a) $delay = 1\mu s$ (b) $delay = 64\mu s$

FIGURE 3 – Distributions de la durée des motifs

lorsque $delay$ est faible, la prise de verrou prend en moyenne 15 μs. Cela s'explique par la très forte contention pour l'accès au verrou due aux 47 threads tentant d'acquérir simultanément le mutex. Lorsque $delay$ devient suffisament grand (environ 30 μs), la contention est réduite et la prise de verrou ne prend plus que 450 ns. Le calcul du facteur de dispersion pour cette expérience montre que lorsque $delay$ est faible (et donc, que la contention est élevée), le facteur de dispersion est d'environ 0.95. Lorsque $delay$ devient suffisament grand pour réduire la contention, le facteur de dispersion baisse à moins de 0.03.
Afin d'analyser plus en détail ces résultats, les courbes de distributions des durées des motifs pour les cas $delay = 1\mu s$ et $delay = 64\mu s$ sont présentés dans les Figures 3a et 3b. Ces courbes de distribution montrent que, dans les deux cas, la majorité des occurences du motif a une durée d'environ 450 ns. Pour le cas $delay = 64\mu s$, le nombre d'occurences ayant une durée plus longue décroit rapidement, et la durée moyenne des occurences du motif (977 ns) est donc proche de la durée minimale. Pour le cas $delay = 1\mu s$, un grand nombre d'occurences ont une durée comprise entre 500 et 2000 ns. Le dernier point du graphe de distribution représentant toutes les occurences de durée supérieure à 13 μs, on observe également que plus de 3000 occurences ont une durée comprise entre 13 μs et 966 μs. La durée moyenne des occurences de ce motif est donc élevée (16 μs).
Nous avons ensuite refait la même expérience en utilisant un spinlock à la place du mutex. Les résultats qui sont présentés dans les Figures 1b et 2b montrent ques les deux cas sont très similaires.
Le facteur de dispersion proposé semble donc être un bon indicateur pour estimer le nombre d'interférences entre les threads sur ce type d'application.

3.2. Spinlock

Un autre type d'interférence entre threads peut être lié à l'utilisation de spinlock : si le système d'exploitation dé-ordonnance un thread ayant acquis un spinlock, les autres threads cherchant à acquérir le verrou devront attendre que le thread propriétaire du spinlock soit ordonnancé. Ce type d'événement est généralement rare, mais très couteux : le temps d'attente est ici de l'ordre de la milliseconde.
Pour évaluer l'efficacité de la méthodologie que nous proposons, nous avons conçu un micro-

benchmark consistant à exécuter, sur une machine à N cœurs, $N + 1$ threads qui, une fois synchronisés, exécutent le code présenté dans le Listing 2.
Nous avons exécuté ce micro-benchmark sur une machine équipée d'un processeur quad-cœur Intel Xeon E5-2603. Sur cette machine, 5 threads sont créés et $delay$ varie de 0 à 100 μs afin de faire varier la fréquence d'apparition des interférences.
La durée moyenne de la prise de verrou en fonction de $delay$ est présentée dans la Figure 1c. Les facteurs de dispersion calculés sont présentés dans la Figure 2c. Les résultats montrent que plus $delay$ augmente, plus la durée moyenne nécessaire à l'acquisition du spinlock diminue. Cela est dû à la probabilité que le thread ayant acquis le spinlock soit dé-ordonnancé qui diminue.
Les facteurs de dispersion calculés évoluent de la même manière que les performances de l'application : le facteur est élevé lorsque la durée moyenne de prise du verrou est affectée par les interférences. À mesure que l'impact des interférences diminue, le facteur diminue également.

3.3. Faux-partage

Les interférences entre threads peuvent également être dues à un problème de faux-partage. Le faux-partage apparaît lorsque plusieurs threads accèdent chacun à des données situées dans une même ligne de cache, entrainant inutilement des opérations de maintient de la cohérence des caches [11].
Afin d'évaluer la méthodologie que nous proposons pour ce type d'interférence, nous avons conçu un micro-benchmark dans lequel deux threads accèdent à des variables indépendantes situées sur une même ligne de cache, de manière à provoquer un faux-partage plus ou moins fréquent. Pour cela, on mesure le temps nécessaire au premier thread pour modifier N fois à sa variable. Le deuxième thread modifie sa variable (invalidant ainsi la ligne de cache de l'autre thread), puis exécute $delay$ instructions. En faisant varier $delay$, on modifie la fréquence d'apparition des interférences entre les deux threads.
La durée moyenne pour que le premier thread accède N fois à sa variable est reportée dans la Figure 1d. Les facteurs de dispersion correspondant sont présentés dans la Figure 2d. Ces résultats montrent que quand $delay$ augmente sur le deuxième thread, le temps d'accès du premier thread diminue jusqu'à atteindre 300 ns. Le faux-partage a donc un impact significatif sur les performances du premier thread jusqu'à $delay = 30$ (le temps d'accès moyen est alors de 403 ns). Les facteurs de dispersion évoluent de manière similaire : le facteur baisse jusqu'à atteindre le seuil de 0.3 lorsque $delay = 30$.

4. Applications : Splash2

Afin d'évaluer la méthodologie proposée, nous l'avons également appliqué à un ensemble d'applications : la suite Splash2 [15]. Nous avons exécuté chacune de ces applications sur la machine `amd48b` décrite en Section 3.1. Pour chaque application, deux types de verrous ont été utilisés : des mutex Posix, ainsi que des verrous RCL [9].
Les résultats de l'expérience sont reportés dans la Table 1. Pour chaque application, sont reportés les temps d'exécution pour Posix et RCL, ainsi que le facteur de dispertion le plus élevé détecté pour chacun des deux cas. Conformément aux observation de travaux précédents, certaines des applications de la suite Splash2 voient leurs performances améliorées en utilisant RCL [9] (Radiosity et Raytrace), alors que les performances des autres restent inchangées. Ceci est dû à la centralisation faite par RCL des accès à certaines variables partagées, qui réduit grandement la contention sur ces applications.
Les applications dont les performances sont améliorées en utilisant RCL à la place de Posix

	Posix		RCL	
Test case	run time (ms)	top score	run time	top score
Barnes	75.0	0.149	80.7	0.155
FMM	29.1	0.120	30.3	0.152
Ocean contiguous	80.1	0.010	79.6	0.005
Ocean non-contiguous	81.2	0.005	83.3	0.008
Radiosity	1041.1	0.894	193.8	0.329
Raytrace balls	250.2	0.673	112.6	0.447
Raytrace cars	272.86	0.868	66.89	0.572
Water Nsquare	63.6	0.214	100.1	0.206

TABLE 1 – Résultats des expériences sur les applications Splash2

ont un facteur de dispersion élevé. Ce facteur élevé semble donc être une métrique permettant d'indiquer quelles applications souffrent d'interférences. Il est à noter que les facteurs de dispersion des application Radiosity et Raytrace utilisant RCL, bien qu'étant plus faible que ceux obtenus avec Posix, restent élevés. Cela suggère que les versions RCL de ces applications souffrent également de contention, mais dans une moindre mesure.

5. État de l'art

L'analyse des traces d'exécution en vue de localiser des problèmes de performances dans les applications parallèles et distribuées a été l'objet de plusieurs travaux. Il y a d'un côté les approches manuelles et de l'autre les approches automatiques.
L'analyse manuelle des traces repose entièrement sur l'exploration de la trace à l'aide d'un outil graphique [3, 10]. Cette démarche a deux principaux défauts. D'une part, il est impossible de représenter les millions d'événements que contiennent certaines traces. D'autre part, une simple représentation graphique ne suffit pas à dévoiler les interférences implicites entre les composants de l'application.
L'analyse automatique des traces s'est donc révélée indispensable. Certains travaux cherchent à localiser des problèmes connus [1, 2, 5]. Ces derniers sont identifiés en localisant des motifs d'événements correspondants. Étant donné que les applications parallèles et distribuées ont tendance à se complexifier de plus en plus, cette approche risque de perdre en efficacité au fil du temps, surtout en dehors du cadre du HPC.
Par conséquent, des techniques qui visent à dévoiler les problèmes qui sont causés par des intéractions implicites et inattendues [12] commencent à émerger.

6. Conclusion

Les interférences entre les threads d'une application multithreadée peuvent causer des problèmes de performance conséquents. De part leur nature aléatoire, ces interférences sont difficilement détectable manuellement.
Dans cet article, nous proposons une méthodologie et une métrique permettant de détecter les interférences entre threads et d'en quantifier l'impact sur les performances globales de l'application. Nous avons appliqué cette méthodologie à un ensemble de micro-benchmark et d'applications. Les résultats de cette évaluation montrent que la méthodologie proposée permet de déterminer si une application souffre des interférences entre ses threads.

Bibliographie

1. Benedict (S.), Petkov (V.) et Gerndt (M.). – Periscope : An online-based distributed performance analysis tool. *In : Tools for High Performance Computing 2009*, pp. 1–16. – Springer, 2010.
2. Chung (I.-H.), Cong (G.), Klepacki (D.), Sbaraglia (S.), Seelam (S.) et Wen (H.-F.). – A framework for automated performance bottleneck detection. – In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–7. IEEE, 2008.
3. Coulomb (K.), Degomme (A.), Faverge (M.) et Trahay (F.). – An open-source tool-chain for performance analysis. *In : Tools for High Performance Computing 2011*, pp. 37–48. – Springer, 2012.
4. David (F.), Thomas (G.), Lawall (J.) et Muller (G.). – Continuously measuring critical section pressure with the free-lunch profiler. – In *ACM SIGPLAN Notices*volume 49, pp. 291–307. ACM, 2014.
5. Geimer (M.), Wolf (F.), Wylie (B. J.), Ábrahám (E.), Becker (D.) et Mohr (B.). – The scalasca performance toolset architecture. *Concurrency and Computation : Practice and Experience*, vol. 22, n6, 2010, pp. 702–719.
6. Graham (S. L.), Kessler (P. B.) et Mckusick (M. K.). – Gprof : A call graph execution profiler. vol. 17, n6, 1982, pp. 120–126.
7. Levon (J.), Elie (P.) et al. – Oprofile, a system-wide profiler for linux systems. *http ://oprofile.sourceforge.net*, 2006.
8. Liu (T.) et Berger (E. D.). – Sheriff : precise detection and automatic mitigation of false sharing. *ACM SIGPLAN Notices*, vol. 46, n10, 2011, pp. 3–18.
9. Lozi (J.-P.), David (F.), Thomas (G.), Lawall (J.) et Muller (G.). – Remote core locking : migrating critical-section execution to improve the performance of multithreaded applications. 2012, pp. 65–76.
10. Müller (M. S.), Knüpfer (A.), Jurenz (M.), Lieber (M.), Brunst (H.), Mix (H.) et Nagel (W. E.). – Developing Scalable Applications with Vampir, VampirServer and VampirTrace. vol. 15, 2007, pp. 637–644.
11. Scott (M.) et Bolosky (W.). – False sharing and its effect on shared memory performance. – In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, p. 57, 1993.
12. Song (L.) et Lu (S.). – Statistical debugging for real-world performance problems. – In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 561–578. ACM, 2014.
13. Trahay (F.), Brunet (E.), Mosli Bouksiaa (M.) et Jianwei (L.). – Selecting points of interest in traces using patterns of events. – In *23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2015.
14. Trahay (F.), Ishikawa (Y.), Rue (F.), Namyst (R.), Faverge (M.) et Dongarra (J.). – Eztrace : a generic framework for performance analysis. – In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pp. 618–619. IEEE, 2011.
15. Woo (S. C.), Ohara (M.), Torrie (E.), Singh (J. P.) et Gupta (A.). – The splash-2 programs : Characterization and methodological considerations. vol. 23, n2, 1995, pp. 24–36.