Algorithmes parallèles adaptatifs et applications

Daouda Traoré
Laboratoire d'Informatique de Grenoble,
51, Av.Jean Kuntzman,
38330 Montbonnot - France
daouda.traore@imag.fr

Mots-Clés : algorithme parallèle, ordonnancement dynamique, vol de travail.

1 Introduction

La proximité croissante des transistors dans un processeur a des effets secondaires qui ne peuvent plus être négligés comme des problèmes de dissipation thermique ou d'interférences. Les fréquences des processeurs atteignent donc une limite. Ainsi pour augmenter la puissance des calculs dans les ordinateurs d'aujourd'hui, la réponse est de ne plus augmenter les fréquences mais d'utiliser plusieurs microprocesseurs qui sont capables de traiter simultanément des données (architectures multicœurs).

Une architecture multicœur est généralement exploitée en concurrence par plusieurs applications en contexte multi-utilisateurs ; aussi le nombre de processeurs physiques et leurs vitesses relatives par rapport à une application peut varier en cours d'exécution de manière non prédictible. Aussi, tout algorithme parallèle introduisant un surcoût, en cas de surcharge, un algorithme séquentiel peut s'avérer plus performant en temps écoulé qu'un algorithme parallèle. L'implémentation efficace nécessite alors un algorithme parallèle qui s'adapte automatiquement et dynamiquement aux processeurs effectivement disponibles. Dans cet article nous présentons les résultats que nous avons obtenus sur quelquues algorithmes complexes (calcul des préfixes et find_if) en les rendant adaptatifs. Pour réaliser l'adaptation de ces algorithmes, nous avons utilisé la technique adaptative proposée par Daoudi et al [6] basée sur le couplage de deux algorithmes, l'un séquentiel et l'autre parallèle à grain fin. Pour réduire les surcoûts dus aux synchronisations et aux parallélismes nous avons aussi utilisé la technique basée sur les trois niveaux de boucle dans [5] et la technique de préemption dans [10, 4]. La section 2 présente l'ordonnancement adaptatif par vol de travail permettant d'ordonnancer l'exécution de l'algorithme parallèle sur des processeurs disponibles. Dans la section 3 nous comparons notre algorithme parallèle adaptatif du calcul des préfixes à l'algorithme parallèle du calcul des préfixes implémenté dans la bibliothèque parallèle C++ TBB [1] qui est basé sur l'agorithme récursif de Ladner et Fischer [8] ; aussi nous comparons notre algorithme de find_if à l'algorithme de find_if implémenté dans la bibliothèque parallèle C++ MPTL [3] qui est basé sur le nombre de processeur sur lequel l'algorithme doit s'exécuter.

2 Ordonnancement adaptatif par vol de travail

L'ordonnancement dynamique adaptatif par vol de travail détaillé dans [6] est basé sur le couplage dynamique de deux algorithmes spécifiés pour chaque fonction du programme, l'un séquentiel, l'autre parallèle récursif à grain fin ordonnancé par vol de travail [7, 6].

Au niveau de l'application, chaque processeur physique exécute un processus unique, appelé exécuteur, qui exécute localement un programme séquentiel. Lorsqu'un exécuteur devient inactif, il devient voleur et cherche à participer au travail restant à faire sur un autre exécuteur actif. Pour cela, il choisit aléatoirement un autre exécuteur (victime) qui est actif et effectue une opération d'extraction d'une fraction du travail restant à faire sur celui-ci grâce à l'algorithme parallèle. Ceci permet de paralléliser la dernière fraction du travail total restant à faire sur la victime, typiquement la dernière moitié, sans interrompre l'exécuteur victime. Le voleur démarre alors l'exécution du travail volé en exécutant l'algorithme séquentiel associé. Lorsque l'exécuteur victime atteint une première opération qui lui a été volée, deux cas apparaissent. Soit le travail volé est terminé et le processeur victime finalise ses calculs grâce au travail effectué par le voleur. Soit le travail volé n'est pas terminé : la victime préempte alors le voleur, lui démande de finaliser le travail déjà réalisé, récupère les résultats calculés par le voleur dont il a besoin et reprend le calcul séquentiel pour terminer le travail restant en sautant le travail déjà réalisé. A l'initialisation, un seul exécuteur démarre l'exécution du programme (version séquentielle), les autres étant inactifs donc voleurs. Lorsque cet exécuteur termine l'exécution, le programme est terminé. Pour amortir le surcoût de la préemption distante et de la synchronisation au niveau de l'accès au travail local, chaque exécuteur exécute le programme séquentiel par bloc d'instructions élémentaires, un bloc étant de taille proportionnelle à la profondeur parallèle du travail restant localement à faire (nanoloop [5]). Nous désignons par W_1 le nombre total d'opérations effectuées par un algorithme parallèle sur p processeurs ; W_∞ le chemin critique en nombre d'opérations de cet algorithme. Pour modéliser la vitesse (nombre d'opérations élémentaires effectuées par unité de temps écoulée), nous utilisons le modèle proposé par Bender&Rabin [2]. A un instant τ, la vitesse $\pi_i(\tau)$ de l'exécuteur i, $1 \leq i \leq p$, dépend du nombre de processus (correspondant à d'autres applications) ordonnancés sur le processeur physique auquel il est affecté. Pour une exécution parallèle de durée T (en temps écoulé) sur p exécuteurs, on définit la vitesse moyenne $\Pi_{ave} = \frac{\sum_{\tau=1}^{T} \sum_{i=1}^{p} \Pi_i(\tau)}{p.T}$.

Le théorème suivant [2, 10] permet d'obtenir une borne sur le temps d'exécution du programme adaptatif à partir de W_1 et W_∞.

Théorème 2.1 *Avec une grande probabilité, le nombre de vols (et donc de préemptions) est* $O(p.W_\infty)$ *et le temps* T_p *d'exécution vérifie* $T_p \leq \frac{W_1}{p\Pi_{ave}} + O\left(\frac{W_\infty}{\Pi_{ave}}\right)$.

Ainsi, si $W_\infty \ll W_1$ alors l'espérance du temps est proche de l'optimal $\frac{W_1}{p\Pi_{ave}}$.

3 Expérimentations

Nos expérimentations ont été faites sur une machine NUMA AMD Opteron à 2.2GHz composée de 8 noeuds bi-coeurs (soit 16 coeurs au total). Les algorithmes ont été implantés sur Kaapi/Athapascan [7]. On appele accélération le rapport entre le temps séquentiel optimal (sur un seul processeur) et le temps sur p processeurs, soit $\frac{T_{seq}}{T_p}$.

3.1 Calcul des préfixes

Etant donnés $x_0, x_1, \ldots, x_n$, les n préfixes π_k, pour $1 \leq k \leq n$, sont définis par $\pi_k = x_0 \star x_1 \star \ldots x_k$ où $\star$ est une loi associative. L'algorithme du préfixe est intéressant de par sa généralité : il n'est pas limité seulement qu'aux entiers, mais peut s'appliquer à toute structure dotée d'une opération associative, et peut donc être utilisé dans les domaines du calcul matriciel, du traitement de l'image, de la reconnaissance de langages réguliers, par exemple. Le calcul des préfixes est un problème ayant un grand degré de parallélisme ; toute parallélisation qui

divise par un facteur 2 le temps d'exécution requiert un nombre d'opérations d'un facteur 1.5 supérieur à un algorithme séquentiel [10]. Le premier algorithme parallèle adaptatif proposé pour ce problème a été donné dans [10]. Nous avons amélioré cet algorithme en utilisant les trois niveaux de boucles de [5]. La figure 1 donne une comparaison de l'accélération de notre algorithme et l'algorithme implémenté dans TBB [1]. Le temps séquentiel optimal mesuré sur un seul processeur est de 15s. Nous remarquons que l'accélération obtenue par notre algorithme est très proche de l'optimale (La borne théorique optimale du calcul des préfixes est de $\frac{2*T_{seq}}{p+1}$ voir [10]). Nous remarquons aussi que l'algorithme implémenté dans TBB est loin de l'optimal, en fait son temps d'exécution est influencé par le choix du grain qui est fixé statiquement et aussi aux surcoûts de créations des tâches dus à la récursivité.

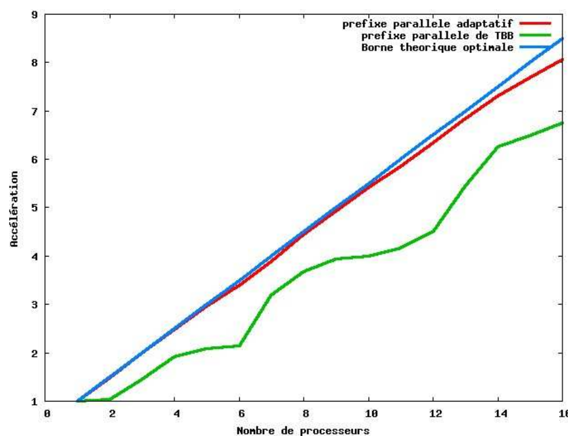

FIG. 1 – Comparaison de notre préfixe adaptatif et le préfixe implémenté dans TBB

3.2 find_if

L'algorithme find_if de la bibliothèque standard générique STL (Standard Template Library) du langage C++ [9] retourne la position du premier élément d'une séquence de données vérifiant un prédicat donné. La parallélisation de cet algorithme est difficile, puisque son temps d'exécution est non prédictible. Nous avons apporté une légère amélioration à l'algorithme proposé dans [5] en ajoutant la technique de préemption [10, 4]. Nous avons comparé cet algorithme à l'algorithme de la MPTL. La figure 2 compare les deux algorithmes sur un tableau de doubles (réels) de taille $n = 10^6$, l'élément à chercher se trouve à la 10^5 position dans le tableau. Le temps séquentiel optimal (sur un seul processeur) de cherche est $3,60s$. Nous remarquons sur la figure 2 que l'accélération obtenue par MPTL ne dépasse pas 2, ceci est dû du fait que l'algorithme est basé sur une découpe en fonction du nombre de processeurs, en fait le temps final du calcul est le temps du processeur terminant le dernier. Notre accélération est meilleure puisque notre algorithme découpe le tableau d'entrée en plusieurs étapes de tailles variables, et si l'élément n'est pas trouvé dans l'étape k il avance dans l'étape $k + 1$ sinon il s'arrête. La technique de préemption aussi accélère notre algorithme, puisque dès qu'un processeur trouve l'élément il arrête tous les processeurs travaillant sur sa partie droite (puisqu'on cherche le premier élément et il a été trouvé à gauche, il est donc inutile de continuer à droite).

4 Conclusion

Dans cet article, nous avons montré les performances expérimentales obtenues sur des machines à mémoire partagée sur deux algorithmes (préfixe et find_if) utilsant le schéma adaptatif

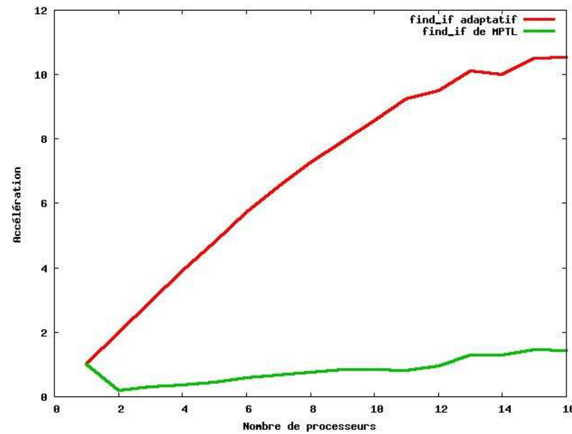

FIG. 2 – Comparaison de notre find_if adaptatif et le find_if de MPTL

basé sur le couplage d'un algorithme séquentiel avec un algorithme parallèle à grain fin ordonnancé par vol de travail. Nos résultats obtenus sont très meilleurs par rapport aux algorithmes basés sur un grain statique ou le nombre de processeurs.

Références

[1] http ://www.intel.com/software/products/tbb/.

[2] M. A. Bender and M. O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory of Computing Systems*, 35(3) :289–304, 2002.

[3] D. BAERTSCHIGER. Multi-processing template library. travail de master, université de genève, 2006.

[4] J. Bernard, J.-L. Roch, and D. Traore. Processor-oblivious parallel stream computations. In IEEE Computer Society, editor, *PDP'2008*, Toulouse, France, February 2008.

[5] V. Danjean, R. Gillard, S. Guelton, J.-L. Roch, and T. Roche. Adaptive loops with kaapi on multicore and grid : Applications in symmetric cryptography. In ACM, editor, *PASCO'07*, London, Ontario, Canada, July 2007.

[6] E.M. Daoudi, T. Gautier, A. Kerfali, R. Revire, and J.-L. Roch. Algorithmes parallèles à grain adaptatif et applications. *Technique et Science Informatiques*, 24 :1—20, 2005.

[7] T. Gautier, J.-L Roch, and F. Wagner. Fine grain distributed implementation of a dataflow language with provable performances. In *ICCS 2007 / PAPP 2007 4th Int. Workshop on Practical Aspects of High-Level Parallel Programming*, Beijing, China, May 2007.

[8] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4) :831–838, 1980.

[9] P.J Plauger, A. A. Stepanov, M. Lee, and David R. M. *The C++ Standard Template Library.* Prentice Hall, 2000.

[10] J.-L Roch, D. Traore, and J. Bernard. On-line adaptive parallel prefix computation. In LNCS 4128 Springer-Verlag, editor, *EUROPAR'2006*, pages 843–850, Dresden, Germany, August 2006.