RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Algorithmes adaptatifs de tri parallèle

Daouda Traoré[1], Jean-Louis Roch[1], Christophe Cérin[2] *

[1]MOAIS, Laboratoire d'Informatique de Grenoble (INRIA, CNRS, INPG-UJF-UPMF),
51, Av.Jean Kuntzman,
38330 Montbonnot - France
[2]Laboratoire de Recherche en Informatique de Paris XIII
UMR CNRS 7030, 99 avenue J.B. Clément
93430 Villetaneuse – France
daouda.traore@imag.fr, Jean-Louis.Roch@imag.fr, christophe.cerin@lipn.univ-paris13.fr

Résumé
Dans cet article, nous présentons deux algorithmes parallèles de tri pour des architectures multicœurs à mémoire partagée dont le nombre où la vitesse des processeurs physiques alloués à une application donnée peuvent varier en cours d'exécution. Ces algorithmes exploitent un ordonnancement dynamique de type vol de travail tel qu'on le trouve dans les librairies Kaapi, Cilk ou Intel TBB. Les performances théoriques sont prouvées asymptotiquement optimales par rapport au temps séquentiel. Les performances expérimentales sont analysées sur deux machines différentes à 8 et 16 cœurs.

Mots-clés : parallélisme, tri, ordonnancement dynamique, vol de travail, algorithmes adaptatifs.

1. Introduction

De part son importance pratique considérable, le tri d'un ensemble de données est intégré dans de nombreuses bibliothèques séquentielles et parallèles. Ainsi, la bibliothèque standard STL du langage séquentiel C++ fournit deux algorithmes génériques différents de tri d'un tableau (ensemble disposant d'un itérateur avec accès aléatoire à un élément) : `sort` (tri introspectif [1] basé sur une partition quicksort) et `stable_sort` (basé sur un tri par fusion), ce dernier garantissant l'ordre relatif de deux éléments de même valeur par rapport à la relation d'ordre choisie. Dans cet article nous considérons la parallélisation de ces deux algorithmes sur les architectures multicœurs.
Une telle architecture 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. Pour réaliser cette adaptation, nous utilisons la technique adaptative proposée par Daoudi&al [12] basée sur le couplage d'un algorithme parallèle à grain fin minimisant la profondeur qui est ordonnancé par vol de travail et d'un algorithme séquentiel minimisant le nombre d'opérations (travail). Cette parallélisation permet d'obtenir une garantie de performances en temps écoulé par rapport à une exécution de l'algorithme séquentiel dans les mêmes conditions. Cette technique et les notations utilisées sont décrites dans la section 2.
Nous appliquons ce couplage pour la construction d'algorithmes parallèles adaptatifs pour `stable_sort` et `sort`. Le premier repose sur une fusion adaptative, le deuxième sur une partition parallèle adaptative. Pour `stable_sort`, la fusion parallèle [11] conservant le nombre d'opérations, sa parallélisation adaptative, présentée dans le paragraphe 3 est directe. Par contre, pour le tri rapide introspectif en place `sort`, le nombre de permutations lors de la partition n'est pas le même en parallèle et en séquentiel.

* Ce travail est réalisé dans le cadre du projet ANR SAFESCALE-BGPR n. ANR-05- SSIA-005.

Tsigas et Zhang [2] ont étudié expérimentalement une version parallèle du tri rapide ; leurs expérimentations donnent de bons résultats. Le temps d'exécution de leur algorithme parallèle sur p processeurs identiques est en $O\left(\frac{n \log n}{p}\right)$ en moyenne et en $O\left(\frac{n^2}{p}\right)$ dans le pire des cas. Leur algorithme est basé sur une parallélisation de la partition [4] avec une découpe qui est dépendant du nombre p de processeurs. Une découpe similaire est utilisée dans la librairie parallèle MCSTL [9]. Outre un pire cas en $O(n^2)$ en travail, l'algorithme parallèle proposé par Tsigas et Zhang n'est pas performant si l'on dispose d'une machine avec des processeurs différents, ou d'une machine utilisée par plusieurs utilisateurs car la charge des processeurs varie au cours d'exécution.
Pour traiter ce problème, nous présentons dans les sections 4 et 5, une parallélisation adaptative du tri introspectif qui permet de garantir une performance $O\left(\frac{n \log n}{p \pi_{ave}}\right)$ où π_{ave} est la vitesse moyenne d'un processeur quelconque asymptotiquement optimale par rapport au temps de l'exécution de l'algorithme séquentiel `sort`. Cette contribution nouvelle est le centre de l'article. Elle repose sur une parallélisation adaptative, totalement indépendante du nombre de processeurs et de leurs vitesses relatives, de la partition utilisée dans le tri introspectif. Les expérimentations sur deux machines multi-cœurs, l'une à base d'Itanium (8 coeurs) l'autre à base d'Opteron (16 coeurs) sont présentées dans la section 6.

2. Couplage adaptatif par vol de travail et notations

Dans toute la suite, nous utilisons l'ordonnancement dynamique adaptatif par vol de travail basé sur le principe travail d'abord [6] pour ordonnancer l'exécution de l'algorithme parallèle sur p processeurs physiques. Cet ordonnancement détaillé dans [12] 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 [6,12]. Ce couplage a été appliqué avec de bonnes performances au calcul parallèle des préfixes [7] et aux algorithmes `for_each` et `find_if` de la STL [8].
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, 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 [8]). Ainsi l'exécution de l'algorithme adaptatif peut être modélisée par un programme Cilk [13] correspondant à un graphe fork-join [14].
Dans toute la suite, on appelle travail, noté W, le nombre total d'opérations élémentaires réalisées lors d'une exécution sur p processeurs. Si il n'y a qu'un exécuteur, l'algorithme séquentiel est exécuté qui effectue W_s opérations. Mais, pour une exécution donnée sur p exécuteurs, le travail dépend aussi des opérations de vol effectuées.
Si l'algorithme est exécuté sur un nombre infini d'exécuteurs, l'algorithme parallèle récursif est exécuté jusqu'au grain fin ; on note alors D la profondeur de ce graphe en nombre d'opérations élémentaires (de coût unité correspondant à un top). Pour une entrée fixée, D et W_s sont supposés invariants ; mais W dépend de l'exécution. Cependant, grâce à l'ordonnancement aléatoire par vol de travail, le temps T_p de l'exécution peut être majoré à partir de W et D sur une architecture avec processeurs de vitesse variable. Pour modéliser la vitesse (nombre d'opérations élémentaires effectuées par unité de temps écoulée), nous utilisons le modèle proposé par Bender&Rabin [14]. 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 $\Pi_{ave} = \frac{\sum_{\tau=1}^{T} \sum_{i=1}^{p} \Pi_i(\tau)}{p.T}$. Si l'on connait le nombre d'opérations W' effectuées par les autres applications concurrentes, on a $\Pi_{ave} = \pi_{max} - \frac{W'}{p.T}$ où π_{max} est la vitesse maximale d'un processeur. Le théorème suivant [13,7] permet d'obtenir une borne sur le temps d'exécution du programme adaptatif à partir de W et D.

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

Ainsi, si $D \ll W$ et $W \simeq W_s$, l'espérance du temps est proche de l'optimal $\frac{W_s}{p\Pi_{ave}}$. La parallélisation classique du tri stable `stable_sort` présentée dans la section suivante illustre ce cadre optimal.

3. Algorithme classique de tri stable

Dans toute la suite, la taille d'un tableau t est notée $n = |t|$; les indices sont numérotés à partir de 0 ; ainsi le tableau contient les éléments $t[0], \ldots, t[|t|-1]$ et est aussi noté $t[0, \ldots, |t|[$.
Le tri stable par fusion consiste à découper le tableau initial en deux parties de taille $n/2$ qui peuvent être triées en parallèle puis à fusionner ces deux parties.
La parallélisation de la fusion de deux tableaux triés T_1 et T_2 suit le schéma proposé dans [12]. Elle consiste à prendre l'élément $e = T_1[|T_1|/2]$ au milieu de T_1 et à chercher, par recherche dichotomique en $\log_2 |T_2|$ comparaisons, l'indice n_e du premier élément supérieur à e dans T_2. Ensuite on applique récursivement le même algorithme de fusion parallèle sur $T_1[0 \ldots |T_1|/2[$ et $T_2[0 \ldots n_e[$ d'une part et en parallèle d'autre part sur et $T_1[|T_1|/2, \ldots, |T_1|[$ et $T_2[n_e, \ldots |T_2|[$.
Pour obtenir un travail W^{fusion} de la fusion parallèle proche de celui de la fusion séquentielle, on arrête la découpe récursive de la fusion de deux tableaux de taille n à un grain $\alpha.\log n$; la constante α permet d'amortir le coût de synchronisation supposé constant [8]. Ainsi le surcoût en nombre d'opérations de création de parallélisme est $O\left(\frac{n}{\log n}\right)$, et le travail $W^{fusion}(n) = W_s^{fusion}(n) + O\left(\frac{n}{\log n}\right)$ est asymptotiquement égal au travail $W_s^{fusion}(n) = \Theta(n)$ de la fusion séquentielle. La profondeur D^{fusion} de cette fusion est $D^{fusion}(n) = D^{fusion}(\frac{n}{2}) + \Theta(\log n) = \Theta\left(\log^2 n\right)$. Grâce au théorème 2.1, on a donc $T_p \leq \frac{W_s^{fusion}}{p.\Pi_{ave}} + O\left(\frac{\log^2 n}{\Pi_{ave}}\right) \simeq \frac{W_s^{fusion}}{p.\Pi_{ave}}$ qui est asymptotiquement optimal.
Pour l'algorithme de tri parallèle par fusion, on en déduit : $W^{tri\_fusion}(n) = 2.W^{tri\_fusion}(\frac{n}{2}) + W_s^{fusion}(n)(1+o(1)) \simeq W_s^{tri\_fusion}(n)$ et la profondeur $D^{tri\_fusion}(n) = D^{tri\_fusion}(\frac{n}{2}) + O\left(\log^2 n\right) = O\left(\log^3 n\right)$. Le temps d'exécution de l'algorithme de tri fusion stable est donc équivalent à $\frac{W_s^{tri\_fusion}}{p.\Pi_{ave}} + O\left(\frac{\log^3 n}{\Pi_{ave}}\right)$ donc asymptotiquement optimal par rapport à l'algorithme séquentiel. On remarquera que l'ordre de grandeur des deux termes composant le temps d'exécution n'est pas le même si bien que dans la pratique on peut raisonnablement espérer obtenir un temps d'exécution en $\frac{n \log n}{p.\Pi_{ave}}$
Grâce au choix du seuil d'arrêt de parallélisation de la fusion de deux tableaux de taille m à un grain $\Theta(\log m)$ [8], l'algorithme parallèle effectue un nombre d'opérations équivalent à la fusion séquentielle. Ainsi, l'exécution séquentielle profondeur d'abord, réalisée par défaut par l'ordonnancement par vol de travail, garantit l'optimalité sans recourir au couplage adaptatif.
Mais la parallélisation du tri introspectif de Musser présentée dans les sections suivantes est elle basée sur le couplage adaptatif avec un algorithme séquentiel.

4. Algorithme adaptatif de partition rapide

Le tri introspectif est basé sur une partition en place de type << quicksort >> : un élément pivot e (pseudo-médiane) est choisi qui est utilisé pour réarranger en temps $\Theta(n)$ le tableau en deux parties, le premier contenant les éléments inférieurs à e, le deuxième ceux supérieurs.

Nous considérons le tableau T comme des juxtapositions de plusieurs blocs de taille éventuellement différentes. Nous disons qu'un bloc a été traité lorsque tous ses éléments ont été visités par l'algorithme de partition. Nous désignons par B_g (resp. B_d) le bloc non traité situé dans la partie extrême gauche (resp. droite) du tableau T privé des blocs traités. Soit **local_partition**(B_g, B_d) la fonction séquentielle qui parcourt les deux blocs B_g et B_d jusqu'à ce qu'au moins tous les éléments d'un de ces deux blocs aient été visités. Elle range les éléments qui sont supérieurs au pivot dans B_d et ceux qui sont inférieurs dans B_g. Notre algorithme parallèle adaptatif pour la partition est alors le suivant.

Nous désignons par P_s l'exécuteur qui initie l'exécution de la partition et par P_v les autres exécuteurs qui font du vol de travail. P_s suit l'algorithme séquentiel de partition, en bénéficiant éventuellement des opérations anticipées par les autres exécuteurs voleurs ; ces opérations correspondent à des intervalles volés qui sont chaînés dans une liste L_v gérée par P_s.

Initialement, P_s démarre la partition séquentielle du tableau T sur l'intervalle $[0, n[$. Il extrait deux blocs non traités de taille $\alpha.\log n$: B_g à l'extrémité gauche et B_d à l'extrémité droite de l'intervalle qui lui reste à partitionner. La constante α est choisie pour rendre négligeable le coût de synchronisation [8].

• **Algorithme séquentiel pour un processus** P_s :

1. P_s travaille localement sur deux blocs de taille $\alpha.\log n$ en extrayant deux blocs non traités (B_g et B_d) à l'extrémité gauche et droite de l'intervalle qui lui reste à faire.
2. P_s exécute **local_partition**(B_g, B_d) jusqu'à ce que l'un des blocs au moins soit traités.
3. Soit alors m le nombre d'éléments de l'intervalle que P_s doit encore partitionner.
   - Si B_g a été traité et qu'il reste des blocs non traités, P_s extrait le bloc de taille $\alpha \log m$ juste à droite de B_g et repart à 2.
   - Si B_d a été traité et qu'il reste des blocs non traités, P_s extrait le bloc de taille $\alpha \log m$ juste à gauche de B_d et repart à 2.
   - Si B_g et B_d ont été traités et qu'il reste des blocs non traités, P_s revient à l'étape 1 en extrayant deux blocs B_g et B_d de taille $\alpha \log m$.
   - Sinon P_s va à l'étape 4.
4. Si L_v n'est pas vide, P_s dépile le premier intervalle volé dans la liste L_v. Si ce vol est terminé, il récupère l'information sur les blocs traités et repart à 4. Sinon, P_s se synchronise avec l'exécuteur P_v qui l'a volé le premier ; pour cela il attend éventuellement que P_v ait terminé son exécution en cours de **local_partition** (préemption faible).
   - Si P_v n'a pas de blocs non traités, P_s récupère l'information sur les blocs traités et repart à 4.
   - Si P_v a un seul bloc non traité, P_s récupère celui-ci et le met dans sa liste B_F des Blocs à finaliser et repart à 4.
   - Sinon si P_s a fini le travail de son intervalle gauche (resp. droit), il récupère la moitié du travail restant à faire par P_v à l'extrême gauche (resp. extrême droite) et repart à l'étape 1.
5. Si il ne reste plus que des intervalles volés non traités à gauche (resp. à droite), P_s ne peut plus extraire de blocs à droite (resp. à gauche), les exécuteurs sont alors inactifs. Les exécuteurs réarrangent les blocs contenus dans la liste B_F et les blocs non traités pour remettre éventuellement chaque bloc à leur bonne place par rapport au pivot. Si tous les blocs étaient traités , P_s s'arrête et l'algorithme se termine. Sinon, il reste au sein du tableau un unique intervalle restant à partitionner : P_s repart alors à l'étape 1 sur cet intervalle.

• **Algorithme parallèle pour les exécuteurs** P_v . Chaque exécuteur possède deux intervalles, l'un à gauche l'autre à droite.

– Lorsqu'il est inactif, P_v choisit au hasard un processeur jusqu'à trouver un exécuteur actif P_w sur lequel il reste des blocs à traiter. Il peut s'agir soit de P_s, soit d'un autre processus voleur. Soit q_g (resp. q_d) le nombre d'éléments restant à partitionner dans l'intervalle gauche (resp. droit) sur P_w.

1. P_v découpe chacun des deux intervalles restants à traiter sur P_w en deux parties de tailles respectives $q_g/2$ et $q_d/2$; il vole la partie droite I_g de l'intervalle gauche et la partie à gauche I_d de l'intervalle droit. P_v insère alors l'information sur l'intervalle volé dans la liste L_v juste après celle de sa victime P_w.
2. Soit m le nombre d'éléments restant à partitionner sur P_v (initialement, $m = q_g/2 + q_d/2$) ; P_v extrait de I_g (resp. I_d) le bloc non traité de taille $\alpha \log m$ le plus à gauche B_g (resp. le plus à droite B_d).

3. P_v applique **local_partition**(B_g, B_d) jusqu'à ce que l'un des blocs au moins soit traités.
4. Si P_w a envoyé un signal de synchronisation à P_v (préemption) ; P_v attend que P_w ait terminé la découpe de l'intervalle qu'il lui reste à partitionner (cf étape 4 de P_s) et repart à 2.
5. Sinon, il continue sa partition en extrayant de nouveaux blocs B_g et/ou B_d (identiquement aux étapes 1, 2, 3 et 4 de P_s)

On remarque que si il y a un seul processeur, l'algorithme exécute une partition séquentielle identique à celle effectuée dans le tri introspectif `sort`. Dans la suite le surcoût de réarrangement, qui est borné n'est pas considéré. Cette partition séquentielle effectue un travail $W_s(n)$ (nombre de comparaisons et permutations) et le travail arithmétique sur p processeurs est $W_p(n) = W_s(n)$. Le théorème suivant montre alors l'optimalité asymptotique de cet algorithme de partition en prenant en compte le surcoût d'ordonnancement.

Théorème 4.1 *Soit $W_s(n)$ le travail séquentiel de la partition. Sur p processeurs de vitesse moyenne Π_{ave} et pour n suffisamment grand, le temps $T_p(n)$ vérifie*

$$T_p(n) = \frac{W_s}{p.\Pi_{ave}} + O\left(\frac{\log^2 n}{\Pi_{ave}}\right).$$

Il est donc asymptotiquement optimal.

Preuve. L'exécution est structurée par l'exécuteur P_s en étapes successives ; d'abord partitionnement partiel du tableau. Puis lorsque tous les vols sont terminés, les blocs de la liste B_F et les blocs non traités sont réarrangés (déplacement des éléments) pour former l'intervalle suivant à partitionner de taille n' égale au nombre d'éléments non classés dans B_F et des blocs non traités. L'étape suivante correspond alors au partitionnement (récursif séquentiel) de ces n' éléments restants à partitionner. A chaque vol correspond au plus un bloc ajouté dans B_F ; la taille de chaque bloc dans B_F est majorée par $\log n$.
Soit $D^{(1)}$ la profondeur de la première étape de partition sur un nombre non borné de processeurs identiques. De part la découpe récursive par moitié lors de chaque vol et la taille logarithmique de chaque bloc extrait pour **local_partition**, avec une infinité de processeurs, chaque processeur exécute une seule fois **local_partition** et traite au moins un bloc, au plus deux. On a donc $D^{(1)} = O(\log n)$ et par suite la profondeur de chacune des étapes est majorée par $O(\log n)$. De plus, le nombre de blocs dans B_F est au plus la moitié du nombre de blocs total : donc $n' \leq n/2$ et le nombre total d'étapes est donc majoré par $\log n$.
Considérons maintenant l'exécution sur p processeurs de la première étape, qui est de profondeur $O(\log n)$. Par le théorème 2.1, le nombre de vols durant cette étape est donc $O(p \log n)$. Comme la profondeur de chacune des étapes est majorée par $O(\log n)$, on en déduit qu'il y'a au plus $O(p \log^2 n)$ vols. Finalement, le théorème 2.1 permet de déduire le temps d'exécution $T_p(n)$ sur les p processeurs de vitesse Π_{ave} : $T_p(n) = \frac{W_s(n)}{p.\Pi_{ave}} + O\left(\frac{p \log^2 n}{\Pi_{ave}}\right)$. Pour $p = o\left(\frac{\sqrt{n}}{\log n}\right)$, le temps d'exécution est alors équivalent à $\frac{W_s(n)}{p.\Pi_{ave}}$ ce qui est optimal. *qed*
En pratique, p est fixé ; l'algorithme adaptatif de partition est alors asymptotiquement optimal. Dans la section suivante, il est utilisé pour paralléliser le tri introspectif `sort`.

5. Parallélisation adaptative du tri introspectif

La parallélisation adaptative de la partition est directement utilisée pour effectuer chacune des partitions du tri introspectif. Après chaque partition adaptative, le sous-tableau contenant les éléments inférieurs d'une part et ceux supérieurs d'autre part peuvent être triés en parallèle.
L'algorithme 1 décrit l'algorithme parallèle adaptatif de tri en Athapascan/Kaapi[6]. L'écriture est similaire à celle de l'implantation de la STL, les deux seules différences étant l'appel à la partition parallèle adaptative (ligne 11) et la parallélisation potentielle des éléments supérieurs au pivot (ligne 12).
Deux seuils sont utilisés dans la boucle principale. Le paramètre `depth_limit`, clef de l'algorithme séquentiel introspectif est initialisé à $\log n$ dans l'appel principal et permet de limiter le travail du tri introspectif à $O(n \log n)$. Si `depth_limit` $== 0$, on fait appel à l'algorithme du tri par tas (ligne 4) qui a toujours une complexité en $O(n \log n)$. Le seuil `grain` permet de limiter la parallélisation récursive parallèle à des tableaux de taille supérieure à $\alpha \log n$.
Sur la ligne 9 de l'algorithme, le pivot choisi est la médiane des trois valeurs (le premier élément, l'élément du milieu et le dernier élément du tableau en cours de tri). Sur la ligne 11 tous les processeurs

inactifs exécutent l'algorithme adaptatif parallèle de la partition à partir du pivot fourni. Puis sur la ligne 12, une tâche est créée pour le tri en parallèle des éléments supérieurs au pivot.

Algorithme 1

```
1  atha_intro_sort_adapt(a1 :: remote < InputeIterator > first, a1 :: remote < InputeIterator > last,
                         size_t depth_limit){
2          int grain = α × depth_limit;
3          while( last − first > grain){
4                  if(depth_limit == 0) return heap_sort(first, last, last);
5                  depth_limit = depth_limit − 1;
6                  typedef typename std : :iterator_traits<InputIterator> : :difference_type diff_t;
7                  typedef typename std : :iterator_traits<InputIterator> : :value_type value_t;
8                  const diff_t sz = last-first;
9                  value_t median = value_t( std : :__median(first, first + sz/2, first + sz − 1));
10                 a1 : :remote<InputeIterator> split;
11                 split = adaptive_parallele_partition(first, last, less_than_median<value_t>(median));
12                 a1 : :Fork< atha_intro_sort_adapt <InputIterator> >()(split, last, depth_limit);
13                 last=split;
14         };
15         std : :sort(first, last); // tri séquentiel si (last − first < grain)
16 };
```

Algorithme 1: Algorithme parallèle adaptatif

6. Expérimentations

Nos expérimentations ont été faites sur deux machines à mémoire partagée NUMA : une Intel Itanium-2 à 1.5GHz avec 31GB de mémoire composée de deux noeuds quadri-coeurs ; une AMD Opteron composée de 8 noeuds bi-coeurs. Les algorithmes ont été implantés sur Kaapi/Athapascan ; la constante α a été fixée à $\alpha = 100$ sur les deux machines après calibrage expérimental.

Les premières expérimentations consistent à trier un tableau de 10^8 doubles (les données sont tirées aléatoirement) en faisant varier le nombre de processeurs utilisés (de 1 à 8 pour itanium et 1 à 16 pour AMD). Les tableaux 1 et 2 donnent les temps d'exécution obtenus par les deux algorithmes (tri par fusion et quicksort) sur les machines Itanium et AMD. Nous avons réalisé dix exécutions et pour chaque test nous avons pris le minimum, le maximum et la moyenne ; les résultats sont très stables.

Nous remarquons dans le tableau 1 que les deux algorithmes se comportent très bien sur 1 à 8 processeurs, et qu'ils se comportent moins bien sur 8 à 16 processeurs. Ceci est dû à la contention d'accès à la mémoire sur cette machine NUMA où chaque bi-processeurs partage la même mémoire. D'ailleurs, lorsque l'on augmente artificiellement le grain de la comparaison à un temps arithmétique de 1µs (en choisissant alors $\alpha = 1$), l'accélération obtenue est linéaire : jusqu'à 15,2 pour 16 processeurs avec $n = 10000$.

On remarque aussi que l'algorithme de tri introspectif (non stable) est meilleur que le tri par fusion. Dans le tableau 2, nous remarquons que les deux algorithmes se comportent très bien, avec de très bonnes accélérations.

Dans le tableau 3, des processus de charge additionnelles sont injectés pour perturber l'occupation de la machine et simuler le comportement d'une machine réelle, perturbée par d'autres utilisateurs. Par souci de reproductibilité, chaque expérience sur $p \leq 8$ processeurs est perturbée par $8 - p + 1$ processus artificiels sur la machine itanium-2 et par $16 - p + 1$ sur la machine AMD. Ainsi, l'un des exécuteurs a sa vitesse divisée par 2, ce qui conduit à $p.\Pi_{ave} \leq (p - 0,5).\Pi_{max}$ où π_{max} est La vitesse d'un coeur dédié. Si T_s est le temps séquentiel, le temps parallèle optimal théorique serait donc $\tilde{T_p} = \frac{T_s}{p-0,5}$; la dernière

ligne du tableau reporte ce temps pour $T_s = 22,45$s.
Conformément à la théorie, on constate les performances stables du tri adaptatif sur cette machine perturbée ; de plus les temps obtenus sont relativement proches à moins de 30% de l'estimation théorique $\tilde{T_p}$.
La figure 1 compare trois algorithmes de tri, le tri rapide avec la partition adaptative, le tri rapide avec la partition de la STL et le tri par fusion parallèle sur les deux machines. Elle montre que notre algorithme de tri parallèle adaptatif avec la partition adaptative est le meilleur sur les deux architectures.

	Tri rapide parallèle adaptatif				Tri par fusion parallèle			
	p=1	p=4	p=8	p=16	p=1	p=4	p=10	p=13
Minimum	22,45	5,51	3,05	2,60	26,1343	7,65498	4,37953	4,23151
Maximum	22,56	5,58	3,30	2,82	26,4367	7,79045	4,5218	4,29168
Moyenne	22,51	5,54	3,14	2,65	26,3054	7,72004	4,46626	4,27504

TAB. 1 – Temps d'exécution tri rapide versus tri par fusion parallèle sur AMD opteron 16 coeurs

	Tri rapide parallèle adaptatif			Tri par fusion parallèle		
	p=1	p=4	p= 8			
Minimum	60,2562	13,3525	7,06782	75,6164	16,5904	8,817766
Maximum	62,087	13,4384	7,77093	75,7446	17,0557	8,92339
Moyenne	60,8665	13,4098	7,1193	75,7019	16,9006	8,8881

TAB. 2 – Temps d'exécution tri rapide versus tri par fusion sur IA64 8 coeurs

	Tri rapide parallèle adaptatif perturbé								
	p=1	p=2	p=4	p=6	p=7	p=8	p=10	p= 12	p=16
Minimum	22,4562	14,2576	6,63771	4,58461	3,83735	3,44868	2,74097	2,74097	2,73606
Maximum	43,9689	17,6414	7,6414	4,92626	4,06878	3,68123	3,25123	3,02834	2,95637
Moyenne	43,6168	15,075	7,31	4,71937	3,91663	3,52036	3,12081	2,84938	2,62112
$\tilde{T_p}$	[22,45 ;44,9]	14,96	6,41	4,08	3,45	2,99	2,36	1,95	1,45

TAB. 3 – Temps d'exécution du tri rapide adaptatif parallèle perturbé sur la machine AMD opteron

7. Conclusions

Dans cet article, nous avons proposé une parallélisation adaptative des algorithmes de la STL de tri par fusion `stable_sort` et de tri introspectif `sort`, les performances de ce dernier étant bien meilleures en séquentiel. La parallélisation du tri par fusion est classique. Par contre l'algorithme proposé pour le tri introspectif et son analyse théorique sont originales à notre connaissance.
Grâce au couplage d'un algorithme séquentiel avec un algorithme parallèle à grain fin ordonnancé par vol de travail, des garanties théoriques de performances sont obtenues par rapport au temps séquentiel sur des machines à mémoire partagée même lorsqu'elles sont utilisées en concurrence par d'autres applications.
Les expérimentations menées montrent le bon comportement des algorithmes même sur des machines dont la charge des processeurs est perturbée ce qui est particulièrement intéressant en contexte multi-utilisateurs. Les perspectives sont d'une part de compléter les études expérimentales sur la tolérance

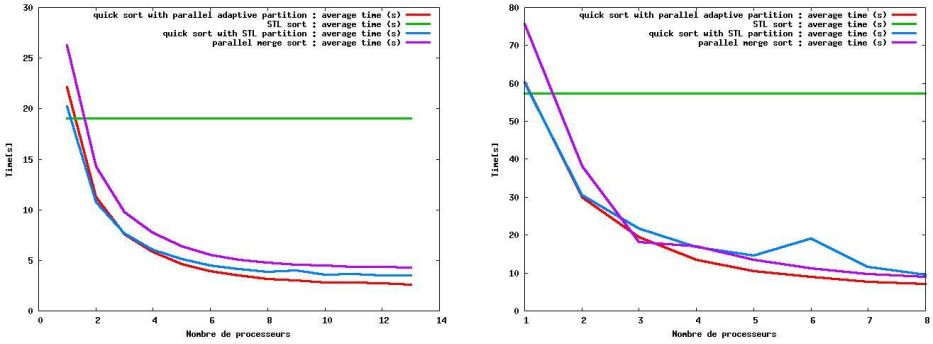

FIG. 1 – Le temps en fonction du nombre de processeurs

aux variations de charge en contexte perturbé : un point difficile concerne la reproductibilité des expérimentations. D'autre part, une autre perspective est de compléter ce travail par une analyse des défauts de cache, avec la comparaison à la parallélisation adaptative d'algorithmes de tri inconscients à la hiérarchie mémoire (cache-oblivious).

Bibliographie

1. D.R. Musser. Introspective Sorting and Selection Algorithms. Software - Practice and Experience, 27(8), pages 983-993, 1997.
2. P.Tsigas and Y.Zhang. A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. pdp, p. 372, PDP'03
3. C.A.R. Hoare. Quicksort. The Computer Journal, 5(1) :10-16, Apr. 1962
4. P.Heidelberger, A.Norton, and J.T.Robinson. Parallel quicksort using Fetch-and-Add. IEEE Transactions on Computers, 39(1) :133-137, Jan. 1990.
5. P. Sanders and S. Winkel. Super scalar sample sort. In 12th Annual European Symposium on Algorithms, ESA 2004.
6. 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.
7. J-L.Roch, D.Traore, J.Bernard. On-line adaptive parallel prefix computation. In proceedings of the EuroPar Computation, Dresden, Germany, 29^{th} August 2006, Springer-Verlag, LNCS 4128, pages 843–850.
8. V.Danjean, R.Gillard, S.Guelton, J-L.Roch, T.Roche. Adaptive Loops with Kaapi on Multicore and Grid : Applications in Symmetric Cryptography. PaSCo, London, Ontario, Canada, July 2007, ACM publishing.
9. F. Putze, P. Sanders, and J. Singler. MCSTL : The multi-core standard template library (extended poster). In ACM 2007 SIGPLAN Conference on Principles and Practice of Parallel Computing. Mar 2007.
10. Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, 2004, pages 235–244.
11. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. Theory of Computing Systems, 31(2) :135–167, 1998.
12. E-M. Daoudi, T. Gautier, A. K, R. Revire, J-L. Roch. Algorithmes parallèles à grain adaptatif et applications. TSI, 24 :1-20, 2005.
13. 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.
14. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems, 34(2) :115–144, 2001.