RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

X-Kaapi : Une nouvelle implémentation eXtrême du vol de travail pour des algorithmes à grain fin

Xavier Besseron, Christophe Laferrière, Daouda Traore *, Thierry Gautier

EPI MOAIS, INRIA
52 av Jean Kuntzmann
F38330 Montbonnot Saint Martin - France

Résumé
Les processeurs multicœurs sont dorénavant des composants standards grand public que l'on retrouve dans un large spectre de machines, du lecteur de DVD incluant un MpSOC, au nœud de calcul des supercalculateurs, en passant par le processeur de la plupart des portables. Dans ce papier nous considérons l'implémentation d'un protocole de vol de travail original pour ordonnancer des applications parallèles sur architecture multicœur. Nous analysons les facteurs introduisant un surcoût à l'exécution et nous en dérivons une implémentation présentée à travers une interface de programmation. Le protocole utilise une coopération entre les voleurs et une victime, ce qui améliore à la fois ses surcoûts d'implantation et les surcoûts dûs à la restructuration du code des algorithmes parallèles. Nos expériences préliminaires montrent qu'il est possible d'obtenir une implémentation ultra-légère possédant un surcoût à l'exécution extrêmement faible pour des algorithmes à grain fin. Cette proposition d'interface constitue le cœur du nouveau moteur exécutif de Kaapi appelé X-Kaapi.

Mots-clés : multicœur, vol de travail, adaptation dynamique, algorithme

1. Introduction

De nos jours, qui ne possède pas chez lui un processeur multicœur ? En 2007 ST Microelectronics annonce la disponibilité de la puce STi7200, un MPSoC[2] pour des applications en vidéo haute définition, typiquement pour des lecteurs DVD. La plupart des portables récents sont équipés de processeurs multicœurs. Les serveurs et les nœuds de calcul de toutes les machines du TOP500 [14] en sont équipés. L'exploitation des performances d'une machine multicœur, c'est-à-dire possédant plusieurs cœurs de calcul, nécessite de comprendre finement les algorithmes et le fonctionnement de l'architecture afin de contrôler à la fois l'ordonnancement des calculs sur les différents cœurs ainsi que le placement des données sur les différents bancs mémoire[3].
Du point de vue de l'architecture et du fait d'une organisation hiérarchique de la mémoire, l'ordonnancement des calculs joue un rôle très important : les politiques de remplacement des données dans les différents niveaux de cache doivent être considérées avec attention sous peine de n'avoir que des performances moyennes à l'exécution.
Du point de vue de l'algorithme que l'on souhaite implémenter sur une telle architecture, sa capacité à utiliser de manière aveugle les caches, de part ses accès à la mémoire bien organisés, est une propriété importante et l'on parle d'algorithmes *cache oblivious* [8]. Une autre propriété fondamentale est que l'algorithme puisse utiliser de manière aveugle un nombre de cœurs qui peut varier dynamiquement, en vitesse si ce n'est en nombre, à cause de variations de charge de la machine ou bien parce que la fréquence de fonctionnement varie pour raison d'économie d'énergie par exemple.
Dans le cadre de cet article, nous considérons le problème de l'exploitation efficace des architectures parallèles multicœurs pour des algorithmes à grain fin. Ce problème important nécessite d'une part la

* L'auteur a réalisé ce travail dans le cadre du projet SCEPTRE.
[2] MultiProcessor System-on-Chip.
[3] Ou un seul, et le problème de placement est simplifié.

conception d'algorithmes adaptés et d'autre part l'étude de techniques d'ordonnancement spécialisées. Notons qu'il est l'un des thèmes cités comme axe prioritaire du plan stratégique de l'INRIA.
Nous restreignons notre étude à trois algorithmes écrits dans le langage de haut niveau Cilk [3], ou utilisant les bibliothèques Intel TBB [11], KaSTL [16] (bibliothèque STL parallèle au dessus de Kaapi [10]) et PaSTeL [13]. Le choix de ces bibliothèques s'explique parce qu'elles offrent toutes un ordonnancement par vol de travail. Cette étude devrait être aussi valable pour les langages de recherche X10 [4] ou Fortress [1] qui offrent le même type d'ordonnancement.
Dans la section suivante nous présentons les bibliothèques que nous avons étudiées. Nous détaillons les coûts induits par l'utilisation d'un algorithme de vol de travail pour ordonnancer les calculs. La section 4 présente l'interface utilisée pour l'insertion de points d'adaptation. Enfin dans la section 5 nous présentons les expérimentations préliminaires réalisées sur une architecture multicœur pour trois algorithmes et leurs résultats. Nous concluons cet article par des perspectives.

2. Travaux existants

Le principe du vol de travail est simple : chaque cœur possède une liste dans laquelle il stocke le travail qu'il doit effectuer ; si un cœur devient inactif alors il cherche à voler du travail dans la liste d'un autre cœur, et cela tant qu'il n'arrive pas à en trouver. En théorie comme en pratique, l'équipe de Cilk [3, 9] a montré que l'ordonnancement par vol de travail permet une exécution efficace des programmes *strict multithreaded computations*, c'est-à-dire pour lesquels les seules dépendances possibles sont entre une tâche mère et ses tâches filles. L'implémentation de Cilk se base sur le principe du travail d'abord (*work first principle*) [9] qui énonce que pour réduire le surcoût à l'exécution il vaut mieux augmenter le chemin critique que le *travail*[4].
On peut décomposer le surcoût d'une exécution d'un programme ordonnancé par vol de travail en deux parties. La première est due à la création de tâches, c'est-à-dire sauvegarder les variables locales en vue d'une continuation dans le cas d'une requête de vol [9]. La seconde partie de ce surcoût est due à la gestion de la file de priorité associée à chaque cœur pour stocker les tâches créées. Cette file de priorité est aussi appelée *workstealing queue*. Dans le cas de Cilk, le protocole de vol T.H.E. est en charge de gérer cette file de priorité. Il permet une concurrence entre un seul voleur et sa victime. Intel TBB [11] se base sur le même type de protocole de vol. Kaapi [10] implémente un protocole de vol de travail qui considère aussi des dépendances entre tâches. PaSTeL [13] utilise des primitives de synchronisation de type POSIX *lock* ou *spinlock* pour garantir une exclusion au moment de modification d'un état global.
Pour réduire ces coûts, deux approches ont été suivies : la première vise à rendre plus légère la synchronisation nécessaire entre les voleurs et la victime. Les travaux [2, 5] ont étendu le protocole de Cilk afin de construire un protocole non bloquant en utilisant une instruction atomique *compare and swap* ou équivalente. Plus récemment [12] a proposé de relâcher la sémantique de gestion de la file de priorité par cœur afin d'éviter l'utilisation d'une instruction atomique *compare and swap* coûteuse pour le possesseur de la file, et donc de réduire ce surcoût sur le travail. Bien que les performances obtenues soient supérieures à celles obtenues par les autres protocoles de vol, la sémantique proposée permet qu'une même tâche soit extraite plusieurs fois de la file. Dans le cas général pour la plupart des algorithmes, le coût engendré pour rendre unique l'exécution d'une tâche masque les gains apportés.
La seconde approche consiste à créer les tâches à la volée lors des requêtes de vol [7, 16, 15]. Outre les gains sur les opérations arithmétiques de l'algorithme exécutées pour exhiber le parallélisme [7], cette approche permet d'éviter les coûts des instructions de création des tâches. La structure de parcours du calcul avec cette approche est organisée en nid de boucles [6]. Pour éviter de complexifier la conception des algorithmes, les bibliothèques KaSTL [16] ou PaSTeL [13] offrent un cadre simple de développement d'algorithmes parallèles qui ne permet pas au compilateur d'entrelacer efficacement la partie arithmétique du calcul et l'itération sur le travail à exécuter.

3. Notations

Étant donné un algorithme séquentiel $\mathcal{A}$, nous noterons par T_{seq} son temps d'exécution sur un cœur de notre architecture multicœur. Nous supposerons qu'il s'agit d'un bon, voir du meilleur, algorithme qui

[4] Terme désignant le nombre d'opérations exécutées.

résout le problème traité. L'algorithme parallèle $\mathcal{A}_{//}$ est a priori capable de s'exécuter sur un nombre quelconque de cœurs. En notant par T_1 le temps d'exécution de l'algorithme $\mathcal{A}_{//}$ sur un cœur, le temps $T_1 - T_{seq}$ représente le surcoût d'utilisation de l'algorithme parallèle par rapport à l'algorithme séquentiel. En notant par T_p le temps d'exécution sur p cœurs, le temps $pT_p - T_1$ représente alors le surcoût d'ordonnancement comprenant le surcoût dû à la création des tâches, le surcoût de gestion de la file de priorité, et le surcoût de l'ordonnancement et sa capacité à bien réguler la charge de calcul. Nous noterons par T_∞ le temps nécessaire à l'exécution sur un nombre infini de cœurs ou encore le temps du plus long chemin dans le graphe de calcul parallèle, *i.e.* le chemin critique.

4. Vol de travail coopératif

Afin de réduire davantage ces coûts, nous proposons un protocole de vol non concurrent entre le voleur et la victime : le principe étant que la victime teste régulièrement la présence de requêtes de vol et les traite si nécessaire. Le gain est double : d'une part il n'y a plus de concurrence entre les voleurs et la victime et cela permet de n'utiliser que des lectures et écritures en mémoire sans primitives de synchronisation coûteuse (verrou ou même une instruction atomique du type *compare and swap*) ; d'autre part une partie de la logique du traitement du vol est expansée dans le code source de l'application ce qui permet une compilation plus efficace de l'algorithme.
Pour cela, nous considérerons une machine multicœur où les seules communications ont lieu par une mémoire partagée. L'algorithme par vol de travail est un algorithme de liste : dès qu'un cœur devient inactif, il cherche à voler du travail chez un autre cœur qui en posséderait. Les deux éléments à définir pour implémenter un tel algorithme sont : ce que représente le travail et déterminer la responsabilité entre le voleur et la victime pour exécuter le traitement associé au vol. Excepté pour PaSTeL, dans toutes les implémentations des bibliothèques citées précédemment, le traitement associé se déroule en concurrence au calcul effectué par la victime. L'idée initiale d'une telle responsabilité était qu'il ne fallait surtout pas interrompre un cœur actif.
En pratique sur des applications à grain très fin (une tâche = une opération arithmétique) et suivant les architectures considérées, nous avons mesuré un ratio $T1/T_{seq}$ entre 20 à 25 pour Cilk et entre 10 à 15 pour Kaapi. Ceci signifie que le coût pour créer une tâche est de plusieurs dizaines de fois plus important qu'un appel de fonction (C ou C++). Pour des programmes parallèles à grain fin qui génèrent un grand nombre de tâches ce coût est important et représente une grande fraction α du travail T_1. Ce coût est donc noté αT_1 avec $0 \leq \alpha \leq 1$. Pour ces mêmes programmes, le nombre de requêtes de vol générées par cœur est en $O(T_\infty)$, qui est faible devant le nombre de tâches créées.
L'idée de X-Kaapi est donc de déplacer du voleur à la victime le traitement des requêtes de vol, que l'on supposera borné par τ dans un premier temps. Dès lors, le nombre de requêtes traitées par la victime sera $O((p-1)T_\infty)$ qui demandera un effort en travail de $O(\tau(p-1)T_\infty)$ qu'il faudra amortir vis-à-vis du gain potentiel que l'on pourra obtenir en $O(\alpha T_1)$. Dans le cas des algorithmes à grain fin, la victime testera suffisamment fréquemment la présence de requêtes de vol et un voleur ne restera pas inactif très longtemps.

4.1. Déclaration d'un point de traitement des requêtes de vol

Nous supposerons un ensemble de p threads utilisant les p cœurs de notre architecture. Chaque thread est responsable de régulièrement insérer un point dans son flot d'exécution pour tester la présence d'une requête de vol. Un thread inactif poste une requête de vol dans un descripteur de requêtes du thread victime : ce descripteur est appelé *contexte de traitement des requêtes de vol*, ou plus simplement *context*. Chaque thread possède un tel descripteur.
Un thread victime traitera non pas une requête de vol mais toutes les requêtes ajoutées dans son descripteur. Le gain potentiel est un meilleur équilibrage du travail. Cette capacité est une différence importante vis-à-vis des découpes binaires utilisées dans les autres bibliothèques citées ci-dessus.
L'instruction[5] `kaapi_stealpoint( thread, splitter [, liste args] )` permet de placer un tel point. La paramètre *thread* représente le thread qui s'exécute et *splitter* une fonction qui sera appelée s'il y a effectivement au moins une requête de vol postée à destination du thread. Cette fonction *splitter* possède un ensemble de paramètres fixes et peut accepter un ensemble de paramètres supplé-

[5] Les extraits de code sont en pseudo C/C++.

```
1.  void std::transform( InputIterator first,
      InputIterator last, OutputIterator result,
      UnaryOperation unary_op)
2.  {
3.    for (; first != last; ++first, ++result)
4.    {



5.      *result = unary_op(*first);
6.    }


7.    return result;
8. }
```

(a) code séquentiel

```
1.  void std::transform( InputIterator first,
      InputIterator last, OutputIterator result,
      UnaryOperation unary_op)
2.  {
3.    for (; first != last; ++first, ++result)
4.    {
        // Passe le contexte courant de calcul (first, last, result, op)
5.      kaapi_stealpoint( thread, splitter,
              first, &last, result, op );
6.      *result = unary_op(*first);
7.    }
      // Terminaison du calcul et préemption éventuelle des voleurs
8.    kaapi_finalize_steal( context, 0,  (kaapi_reducer_function_t)reducer, 0 );
9.    return result;
10. }
```

(b) code parallèle à grain fin

FIG. 1 – Exemple d'utilisation sur le code STL std : :transform

mentaires (`[liste args]`) passés par l'utilisateur lors de la déclaration du point de traitement de vol. La signature de la fonction *splitter* est la suivante :

```
void splitter(kaapi_stealcontext* context, int count,
              kaapi_request** request  [, liste args] );
```

Celle-ci est appelée avec une valeur de `count` strictement supérieure à 0. Le tableau de pointeurs vers des requêtes de vol `request[i]` contient au moins `count` éléments non nuls.

Si aucune requête de vol n'est postée, le coût de cette instruction est comparable à un test à zéro. S'il y a au moins une requête de vol alors la fonction *splitter* est appelée afin de répondre aux différentes requêtes de vol.

La figure 1 illustre l'utilisation à grain fin de cette instruction sur le code de l'algorithme `transform` de la STL. En pratique il conviendra d'amortir le coût de l'instruction `kaapi_stealpoint` en effectuant quelques itérations de la boucle. Dans [6, 15] cette boucle est appelé la *nano loop*.

KaSTL et PaSTeL sont basées sur une décomposition du calcul séquentiel dans une *nano loop* écrite dans les bibliothèques et qui fait appel à une fonction utilisateur pour chaque unité de calcul séquentiel. À la différence de ces bibliothèques, dans l'interface X-Kaapi, le flot de contrôle séquentiel de l'application contient les appels nécessaires à certains points précis du calcul afin de traiter le vol. Puisque le compilateur à une vision globale de la *nano loop* et du code utilisateur, nous pouvons espérer de meilleures optimisations. Le second intérêt est d'autoriser le passage d'un contexte d'exécution séquentiel (des variables dans le code séquentiel) directement à la fonction de traitement du vol. Par exemple, à la ligne 5 de la figure 1b, nous passons en paramètre de la fonction l'ensemble des itérateurs décrivant l'état du calcul en cours. En retour l'itérateur de fin de boucle *last* est potentiellement muté suite à une requête de vol. La terminaison du calcul, ligne 8 de la figure 1b, est décrite à la section 4.3 et représente un coût d'un test à 0 si aucun vol n'a été effectué. Le surcoût global du code parallèle est l'ajout de deux instructions.

4.2. Traitement des requêtes de vol

La figure 2 montre le code de la fonction `splitter` appelée lors d'une ou plusieurs requêtes de vol. La fonction `splitter` est en charge du découpage du travail en au plus autant de requêtes que reçues par le thread courant. Une fois le travail découpé, elle doit signifier à chacun des cœurs voleurs la disponibilité de son travail. Pour cela la fonction doit allouer les arguments nécessaires à chacun des voleurs (ligne 9). Cette allocation s'effectue par le thread recevant les requêtes de vol. Pour signifier au voleur un succès (ligne 16) ou un échec (ligne 19) de sa requête de vol, la fonction `splitter` répond en lui précisant un point d'entrée (une fonction à appeler `thief_entrypoint`) pour exécuter le travail volé et alloué précédemment.

À la différence de toutes les autres bibliothèques, l'interface que nous proposons permet de traiter `k` requêtes de vol en même temps. Le principal avantage est une meilleure répartition du travail envers plusieurs cœurs inactifs, en particulier au début du calcul.

4.3. Terminaison du calcul parallèle

En fin de calcul (ligne 8 de la figure 1b), le code parallèle appelle l'instruction `kaapi_finalize_steal` chargée de terminer le calcul parallèle. En réutilisant les schémas d'algorithme adaptatif [15, 16], le thread qui a commencé un calcul et qui s'est fait voler arrête ses voleurs, dans l'ordre inverse à celui de

```
1.  void splitter( kaapi_stealcontext* context, int count, kaapi_request** request,
                   InputIterator first,  InputIterator* last,
                   OutputIterator result, const UnaryOperation& unary_op)
2.  {
3.      size_t bloc, size = (*last - first);
4.      bloc = size / (1+count);
5.      while (count >0)
6.      {
7.        if (request[i] !=0)
8.        {
9.          if (kaapi_steal_context_alloc_result( context, request[i], (void**)&data_thief,
                    2*sizeof(InputIterator) + sizeof(OutputIterator) + sizeof(UnaryOperator)  ) ==0)
10.          {
11.            data_thief->last = *last;
12.            data_thief->first = *last-bloc;
13.            *last -= bloc;
14.            data_thief->result = result + (data_thief->first - first);
15.            data_thief->op   = op;

               /* reply ok (1) to the request */
16.            kaapi_request_reply( request[i], context, &thief_entrypoint, 1 );
17.          }
18.          else {
               /* reply failed (=last 0 in parameter) to the request */
19.            kaapi_request_reply( request[i], context, 0, 0 );
20.          }
21.          --count;
22.        }
23.        ++i;
24.      }
25. }
```

FIG. 2 – Exemple simplifié de découpe du travail pour le code STL std : :transform en k

ses réponses aux requêtes de vol. Cet arrêt permet de terminer en séquentiel le calcul : cette opération s'appelle la *préemption* du voleur. Lorsqu'un voleur est préempté, une fonction de réduction est appelée afin de permettre une fusion entre les résultats calculés par la victime et ceux calculés par le voleur. Cette fonction est précisée en paramètre d'appel à `kaapi_finalize_steal`. En retour d'appel, l'ensemble du calcul parallèle est terminé et la mémoire modifiée par les voleurs peut être lue.

4.4. Protocole de communication entre threads voleurs et threads victimes

Que ce soit pour poster une requête de vol auprès d'un thread victime ou lorsque la victime répond aux voleurs, nous utilisons une communication à travers des zones mémoire de communication. Du fait d'un possible ré-ordonnancement des écritures ou des lectures en mémoire par les cœurs ou par un compilateur, nous utilisons le protocole suivant utilisant une "barrière mémoire". Le cœur qui émet le message est appelé le *producteur* du message. Celui qui reçoit le message est le *consommateur*.

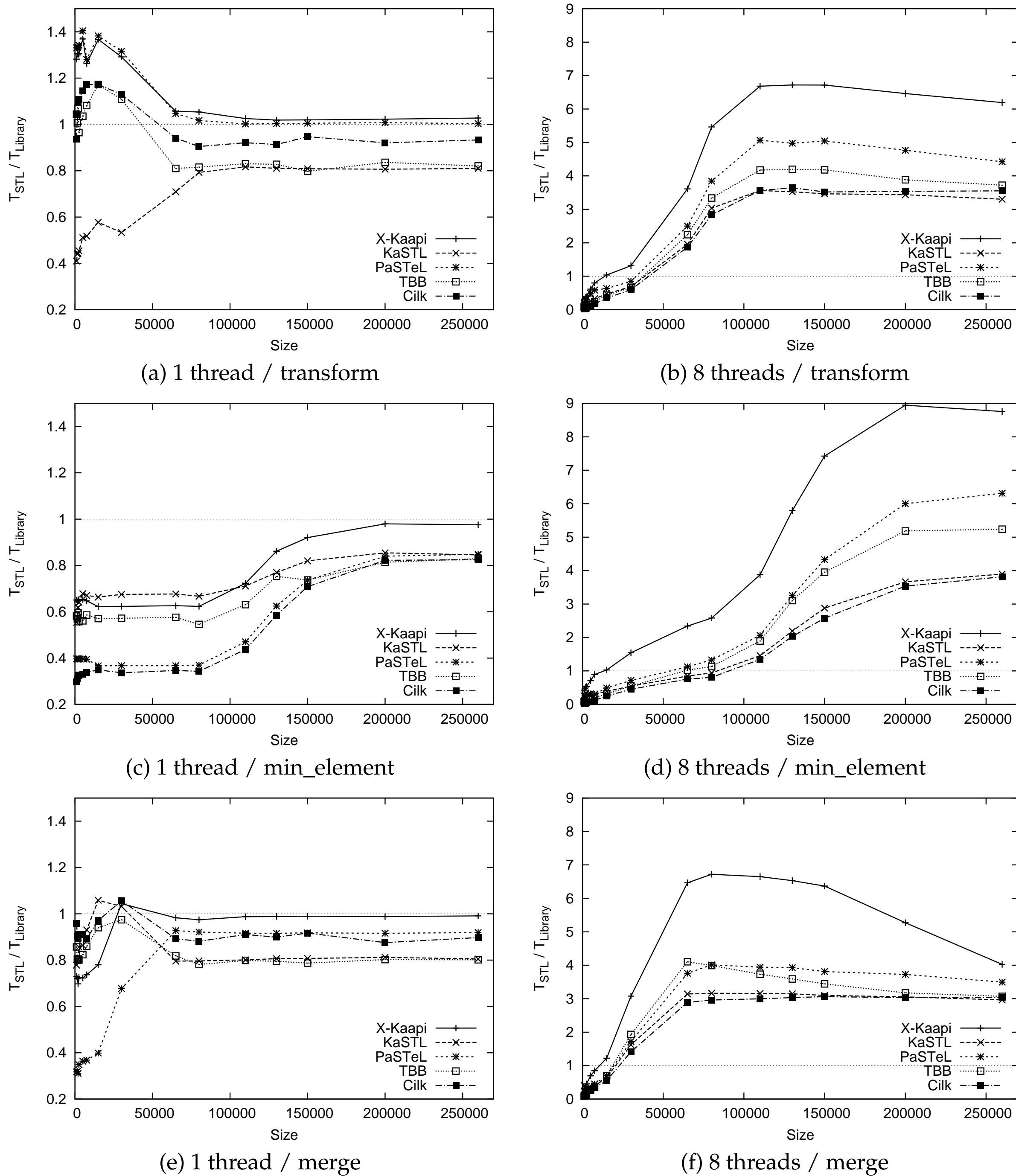
code producteur : 1/ les données du message sont écrites dans la zone mémoire ; puis 2/ une barrière de type écriture est appelée pour garantir que l'écriture suivante ne sera pas réordonnancée avant la barrière ; 3/ enfin une variable d'état de la zone mémoire est changée atomiquement pour indiquer que les données sont écrites.

code consommateur : 1/ le lecteur teste atomiquement la valeur de la variable d'état et si celle-ci change d'état alors 2/ une barrière de type lecture est utilisée afin de garantir que les lectures suivantes ne sont pas ordonnancées avant la barrière ; puis 3/ les données du message peuvent être lues.

Le voleur émet un message "requête de vol" en utilisant ce protocole, puis attend (de manière actif) une réponse de la part de la victime. Celle ci émet un message "réponse de la requête" en utilisant le même procédé.

5. Expérimentations

Nous avons comparé les accélérations de chacune des bibliothèques Cilk, TBB, PaSTel, KaSTL et notre proposition X-Kaapi avec les algorithmes de la STL `merge`, `min_element` et `transform`. Le type C++ utilisé est le `double` pour tous les algorithmes. L'opération arithmétique effectuée par l'algorithme `transform` est l'opération `*=2`. Les codes sont tous écrits en C++ et compilés avec la version 4.3 du compilateur `g++` avec option d'optimisation `-O2`. La machine d'expérience est une architecture multi-cœur NUMA composée de 8 processeurs dual-cœurs AMD 875 à 2.2 Ghz. À chaque processeur est associé un banc mémoire de 4 Go, soit 32 Go de mémoire au total. Chaque expérience qui utilise K cœurs (K=1, 2, 4, ou 8) est réalisée en figeant le thread noyau POSIX de calcul sur un cœur des K processeurs utilisés. La stratégie d'allocation des tableaux est contrôlée en utilisant `numactl` afin d'allouer chaque page des tableaux sur chacun des K bancs mémoire utilisés de manière cyclique (stratégie `--interleave` de `numactl`).



(a) 1 thread / transform (b) 8 threads / transform

(c) 1 thread / min_element (d) 8 threads / min_element

(e) 1 thread / merge (f) 8 threads / merge

FIG. 3 – Accélérations des bibliothèques par rapport à la STL

Chaque valeur correspond à la moyenne des mesures de 30 exécutions du programme avec les mêmes entrées. L'intervalle de confiance calculé est dans tous les cas très petit (de deux ordres de grandeur plus petit que les valeurs). Le programme commence par faire une série du même calcul afin de "chauffer" les caches. Les temps reportés étant les temps après "chauffe", l'objectif premier étant de mesurer les coûts arithmétiques sans tenir compte des effets de cache.

5.1. Accélération de l'exécution

La figure 3 montre l'accélération $T_{seq}/T_{library}$ pour chacune des bibliothèques testées en fonction de la taille du tableau d'entrée et pour les trois algorithmes. À gauche les graphes montrent cette accéléra-

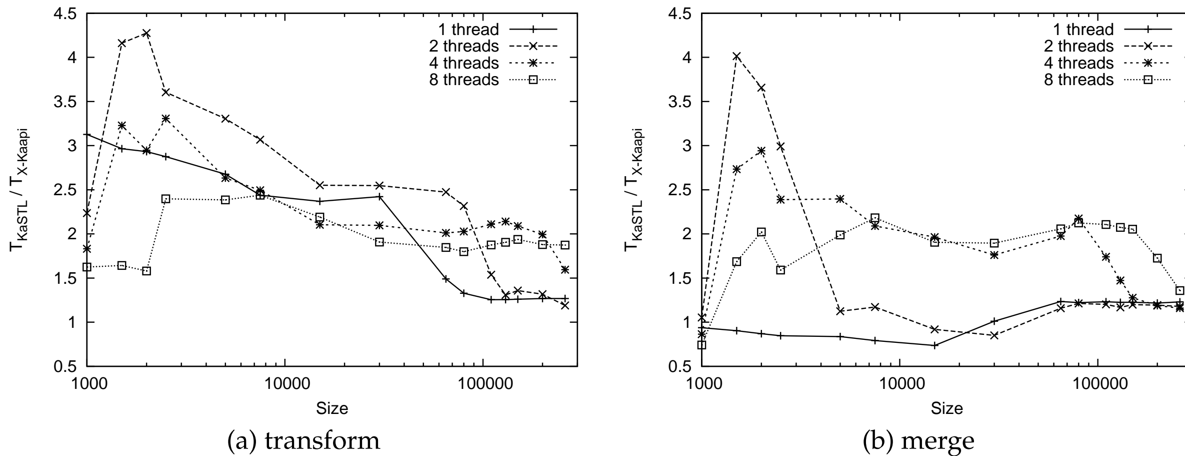
tion lorsqu'un seul thread est utilisé ; à droite, lorsque 8 threads sont choisis. Le premier phénomène à noter est le gain à utiliser PaSTeL ou X-Kaapi avec un seul thread par rapport à la STL sur l'algorithme `transform`. Ce comportement n'est pas encore expliqué. Les accélérations de X-Kaapi et PaSTeL sont similaires dans ce cas, mais PaSTeL traite le cas d'utilisation avec un thread de manière optimisée en court-circuitant la version parallèle.

Sur les figures de droite, on remarque que sur 8 cœurs, X-Kaapi permet un gain d'au plus 6.68 pour `transform` et 6.72 pour `merge`. Pour `min_element`, le gain maximum observé est 8.95. Ce gain supérieur à 8 sur peut s'expliquer par le fait qu'en utilisant 8 cœurs, on bénéficie d'un cache 8 fois plus grand que la STL qui ne tourne que sur un cœur. Dans le cas de `min_element`, le gain représente environ 49% de plus que PaSTeL, 73% de plus que TBB et 153% de plus que Cilk++.

Enfin le dernier point à noter concerne la taille à partir de laquelle le code X-Kaapi est plus rapide que le code séquentiel de la STL sur 8 threads : elle est d'environ 15000 pour X-Kaapi, plus de 30000 pour les autres bibliothèques.

5.2. Gain par rapport à KaSTL

Historiquement X-Kaapi a été conçu suite aux performances observées de KaSTL au dessus de Kaapi : cette série de mesures cherche à confirmer les choix effectuées dans X-Kaapi à travers les gains obtenus, en particulier pour des tableaux de petites tailles (quelques milliers d'éléments). Pour cela, la figure 4 montre l'accélérations de X-Kaapi par rapport à KaSTL sur les algorithmes `transform` (à gauche) et `merge` (à droite). Notons que les temps pour la plus petite taille est $4.48e^{-06}$s pour l'algorithme `transform` et $5.97e^{-06}$s pour `merge` avec X-Kaapi. Globalement X-Kaapi est à un facteur >1 plus rapide que KaSTL, si ce n'est quelques cas dégénérés avec 1 thread, ou sur les plus petits tableaux. Les gains sont plus importants lorsque le nombre de threads augmente.



(a) transform (b) merge

FIG. 4 – Gain de X-Kaapi par rapport à KaSTL

6. Conclusions

Nous avons présenté la problématique à partir de laquelle nous avons créé X-Kaapi. Nous avons donné une analyse des coûts lors d'un ordonnancement par vol de travail pour en déduire une nouvelle approche originale à deux points de vue : d'une part X-Kaapi permet de traiter K requêtes de vol en même temps ce qui permet un meilleur équilibrage de la charge en cas de vol. D'autre part, l'implémentation de ce protocole de vol coopératif s'effectue uniquement avec des écriture / lecture atomique en mémoire. Les expérimentations confirment les choix d'implémentations. Notons que les gains mesurés avec cette approche peuvent aussi être transposés aux cas des algorithmes récursifs à grain fin.

Ces travaux se poursuivent sur deux directions : d'une part essayer d'établir une garantie théorique sur les performances obtenues pour des algorithmes à grain fin ; d'autre part lever la limitation sur la granularité de l'algorithme, soit en mixant des approches classiques d'implémentation de protocole de vol avec une approche coopérative, soit en utilisant des mécanismes d'interruption du thread victime.

Bibliographie

1. Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr. Project fortress : A multicore language for multicore processors. *Linux Magazine*, september 2007.
2. Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98 : Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM.
3. Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5) :720–748, 1999.
4. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10 : an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05 : Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.
5. David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05 : Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM.
6. Vincent Danjean, Roland Gillard, Serge Guelton, Jean-Louis Roch, and Thomas Roche. Adaptive loops with kaapi on multicore and grid : Applications in symmetric cryptography. In ACM publishing, editor, *Parallel Symbolic Computation'07 (PASCO'07)*, London, Ontario, Canada, July 2007.
7. El Mostafa Daoudi, Thierry Gautier, Aicha Kerfali, Rémi Revire, and Jean-Louis Roch. Algorithmes parallèles à grain adaptatif et applications. *Technique et Science Informatiques (TSI)*, 24 :1–20, 2005.
8. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99 : Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285, Washington, DC, USA, 1999. IEEE Computer Society.
9. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5) :212–223, 1998.
10. Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI : a thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Parallel Symbolic Computation'07 (PASCO'07)*, number 15–23, London, Ontario, Canada, 2007. ACM.
11. A. Kukanov and M. Voss. The foundations for scalable multi-core software in intel® threading building blocks. *Intel Technology Journal*, november 2007.
12. Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *PPoPP '09 : Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, New York, NY, USA, 2009. ACM.
13. Érik Saule and Brice Videau. PaSTeL. une implantation parallèle de la STL pour les architectures multi-coeurs : une analyse des performances. In *Proceedings des Rencontres Francophones du Parallélisme, RenPar'18*, Fribourg, Suisse, feb 2008.
14. Horst D. Simon, Hans W. Meuer, Hans W. Meuer, Erich Strohmaier, Erich Strohmaier, Jack J. Dongarra, Jack J. Dongarra, and D. Simon. Top500 supercomputer sites. Technical report, http ://www.top500.org, november 2008.
15. Daouda Traore. *Algorithmes parallèles auto-adaptatifs et applications*. PhD thesis, INPG, dec 2008.
16. Daouda Traore, Jean-Louis Roch, Nicolas Maiilard, Thierry Gautier, and Julien Bernard. Adaptive parallel algorithms and applications to stl. In Springer-Verlag, editor, *EUROPAR 2008*, Las Palmas, Spain, August 2008.