HAL
open science

PaSTeL. Une implantation parallèle de la STL pour les architectures multi-coeurs : une analyse des performances

Érik Saule, Brice Videau

To cite this version:

Érik Saule, Brice Videau. PaSTeL. Une implantation parallèle de la STL pour les architectures multi-coeurs : une analyse des performances. Proceedings des Rencontres Francophones du Parallélisme, RenPar'18, 2008, Fribourg, Switzerland. hal-00953634

HAL Id: hal-00953634

https://inria.hal.science/hal-00953634

Submitted on 25 Mar 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

PaSTeL
Une implantation parallèle de la STL pour les architectures multi-cœurs : une analyse des performances

Erik Saule (MOAIS) et Brice Videau(MESCAL)
Laboratoire d'Informatique de Grenoble
Email : {erik.saule, brice.videau}@imag.fr

Résumé

Dans cet article, nous proposons la bibliothèque PaSTeL, implémentation parallèle d'une partie de la STL, bibliothèque standard du langage C++. PaSTeL propose à la fois un modèle de programmation pour la construction d'algorithmes parallèles, mais également un modèle d'exécution basé sur du vol de travail. Une attention toute particulière a été portée sur l'utilisation de mécanismes optimisés de synchronisation et d'activation des threads. Les performances de PaSTeL sont évaluées sur une machine de bureau avec un processeur à deux cœurs, mais également avec une machine disposant de 16 cœurs. On notera que les performances de PaSTeL sont supérieures à celles d'autres implémentations de la STL même pour des petites exécutions sur des petits jeux de données.

1. Introduction

L'émergence des architectures de processeurs multi-cœurs a eu pour principale conséquence que les ordinateurs fixes et même les portables sont rapidement devenus des machines multiprocesseurs. La concurrence entre les grands constructeurs (AMD, Intel) est particulièrement vive et chacun d'eux a cherché à être le premier à sortir le processeur quadri-cœur. La société Intel est même allée jusqu'à annoncer qu'elle développait dans ses laboratoires un prototype de processeur à 80 cœurs qui atteindrait la puissance crête d'un téraflops. De son côté, l'utilisateur qui passe une bonne partie de son temps à lire ses courriers électroniques ou bien à surfer sur l'Internet, se sent finalement relativement peu concerné puisqu'il n'utilise toujours qu'un seul cœur. Les programmeurs d'applications, débutants et mêmes confirmés, n'ont que très rarement les connaissances en algorithmique parallèle et le temps nécessaire à la mise au point de tels algorithmes, leur permettant d'exploiter les différents cœurs d'un processeur.

Depuis une bonne dizaine d'années, les approches et méthodologies de développements basées sur l'objet rencontrent un vif succès. Plusieurs langages de programmation, comme C++ et plus récemment encore Java, sont largement utilisés et même enseignés aux étudiants. Ces langages s'appuient sur des bibliothèques standards (ou des paquetages) dont l'objectif est de faciliter et raccourcir les phases de développement. Un exemple bien connu est celui de la bibliothèque STL[1] qui définit un ensemble de structures de données classiques et génériques ainsi que les différents algorithmes qui s'y appliquent. Les programmeurs peuvent ainsi se focaliser sur l'application qu'ils ont à implémenter en s'appuyant sur des structures et des algorithmes déjà connus. Les temps d'exécution de ces algorithmes peuvent être relativement importants, par exemple lorsque le programmeur a besoin de trier un ensemble d'éléments de grande taille. Pour de tels exemples, il est certain que les programmeurs et les utilisateurs aimeraient vivement disposer d'algorithmes capables de s'exécuter sur les différents cœurs d'un processeur.

[1] Standard Template Library

Cet article présente PaSTeL, une bibliothèque d'algorithmes parallèles en mémoire partagée, qui peut se substituer partiellement à la STL. Nous nous sommes fixés comme objectifs que PaSTeL puisse répondre aux critères d'efficacité sur des jeux de données de grande taille, mais également de petite taille. Le modèle d'exécution est basé sur la technique du "vol de travail" qui garantit une bonne répartition de la charge sur les différents processeurs. En utilisant des mécanismes efficaces de synchronisation et d'activation des threads, nous montrons que PaSTeL affiche de très bonnes performances, même pour des jeux de données de petite taille. Les performances de PaSTeL ont été comparées avec celles obtenues avec la bibliothèque MCSTL [7] développée à l'université de Karlsruhe.
L'organisation de cet article est la suivante : la section 2 fait un état de l'art des travaux menés autour de la parallélisation de la STL. Nous y décrivons en particulier le mécanisme de vol de travail qu'on retrouve dans plusieurs environnements et que nous avons retenu pour PaSTeL. Les principaux choix de conception et d'implémentation sont décrits dans la section 3. Nous détaillons quelques algorithmes de base de la STL que nous avons parallélisés dans PaSTeL. La dernière section du papier décrit les principales expérimentations que nous avons menées, d'abord avec un ordinateur portable et son processeur à deux cœurs, puis ensuite avec une machine multiprocesseurs à 16 cœurs.

2. Des approches de parallélisation de la STL

La parallélisation des algorithmes de base de la STL est un sujet abordé par plusieurs auteurs. La Section 2.1 dresse un tableau des différentes bibliothèques ajoutant à la STL une sémantique parallèle. Les méthodes de vol de travail ne sont pas nouvelles non plus, de nombreux travaux étudient ce problème. La Section 2.2 discute les moteurs existants. Finalement, la Section 2.3 fait le point sur les différentes approches présentées.

2.1. Parallélisation de la STL

STXXL [4] est une bibliothèque dérivée de la STL qui a pour but de gérer de très grands volumes de données. Entre autres, STXXL propose des conteneurs qui permettent une gestion intelligente des disques durs. STXXL repose sur une architecture à mémoire distribuée pour stocker ces grands volumes de données. Les processeurs distribués servent également à traiter les opérations sur les données qu'ils possèdent. En un sens, STXXL est l'approche duale de PaSTeL.
STAPL [2] est une bibliothèque redéfinissant les concepts de la STL (conteneurs, itérateurs et algorithmes) pour les architectures parallèles à mémoire partagée comme distribuée. Bien que l'algorithmique utilisée soit adaptative, STAPL n'obtient pas de performances intéressantes sur les données de petites tailles [2].
MCSTL [7] est une implantation parallèle de la STL pour les architectures multi-cœurs utilisant openMP pour la parallélisation. Les algorithmes *embarassingly parallel* utilisent un moteur de vol de travail tandis que les autres utilisent des algorithmes parallèles ad hoc. MCSTL est à notre connaissance la seule STL qui vise directement les machines à mémoire partagée multi-cœurs. C'est pourquoi MCSTL sera notre point de comparaison lors de notre étude expérimentale.
Notons également que des travaux visant à obtenir des accélérations sur des petits jeux de données existent. [5] étudie comment utiliser les opérations vectorielles des processeurs modernes pour trier plus rapidement des ensembles de l'ordre d'une centaine d'éléments.

2.2. Moteur pour le vol de travail

Cilk [6] est une extension du langage C permettant programmation parallèle pour machine à mémoire partagée. L'efficacité du calcul parallèle est assurée par des méthodes de vols de travail

dont la clé de voûte est le *work-first principle* qui affirme que le surcoût lié au parallélisme ne doit apparaître que lorsque le parallélisme est rendu effectif. Le but recherché par Cilk est la parallélisation de l'intégralité d'une application ; le moteur de parallélisation est donc très générique. En effet, chaque processus Cilk maintient à jour une liste à double sens du travail à effectuer. Le processus ajoute et consomme les tâches par un bout tandis que les voleurs consomment du travail par l'autre bout. De ce fait, le parallélisme de l'application est exposé de façon explicite aux autres processus en cours d'exécution. Le modèle de programmation de Cilk est fortement basé sur la récursivité. En effet, dans Cilk, ce sont les fonctions qui sont exécutables de façon asynchrone.
Récemment, Intel a proposé des outils appelés *Threading Building Blocks* [1] d'aide à la parallélisation d'algorithmes dont l'architecture est calquée sur celle de Cilk.
KAAPI/ATHAPASCAN [3] est un moteur d'exécution pour la programmation adaptative par vol de travail. Il présente de nombreux points de ressemblance avec Cilk. Les principales différences sont que KAAPI est un moteur en C++ et qu'il fonctionne efficacement sur les grandes architectures à mémoire distribuée de type grille de calcul.

2.3. Bilan des approches connexes

Aucune implantation parallèle de la STL ne vise à obtenir des performances intéressantes sur des petites structures de données.
L'exposition du travail parallèle disponible est une opération qui a un impact non négligeable lorsque l'on cherche à traiter de petits volume de données. Cette exposition est nécessaire lorsque l'on parallélise une application entière ce qui en fait un choix pertinent pour les moteurs de vol de travail comme Cilk ou KAAPI. Cependant, dans le cadre de l'écriture d'une boite à outils d'algorithmes parallèles à forte réactivité comme PaSTeL, cette exposition entraîne un surcoût que l'on ne peut pas se permettre.

3. PaSTeL

Nous présentons ici PaSTeL[2] en détail. Nous commençons par détailler le modèle de programmation et d'exécution des algorithmes. Ensuite, nous discuterons de l'implantation de ce modèle.

3.1. Modèle de programmation

La programmation d'un algorithme dans PaSTeL demande de fournir des structures de données et des fonctions particulières.
Tout d'abord, il faut fournir une structure de données qui sera globale à l'exécution de l'algorithme et une structure de données qui sera propre à chaque processeur. La structure globale peut servir à effectuer une synchronisation globale entre les processeurs. Tandis que la structure propre à chaque processeur sert à savoir où en sont les calculs de chaque processeur. Il faut également fournir les fonctions d'initialisation et de libération de ces structures.
Un algorithme est également définit par des fonctions qui seront appelées par le moteur PaSTeL. La première fonction, TravailGlobal sert à détecter la fin de l'algorithme. Tandis que, TravailLocal sert à savoir si un processeur particulier a actuellement du travail. Le travail est effectué par morceau dans la fonction FaireUniteTravailLocal, la taille de cette unité de travail, aussi appelée grain, est à définir par l'utilisateur de PaSTeL. L'état global de l'algorithme, contenant son avancement, est mis à jour à l'aide de la fonction CommitLocalDansGlobal. Finalement, lorsqu'un processeur n'a pas de travail à effectuer, il vole le travail d'un autre processeur avec la fonction Vol.

[2] PaSTeL est disponible sur https ://gforge.inria.fr/projects/pastel/

3.2. Modèle d'exécution

Lorsqu'un programme appelle une fonction de PaSTeL, le processeur exécute l'Algorithme 1. Chaque processeur qui se rajoutera au calcul utilisera lui aussi le même algorithme qui terminera quand la fonction TravailGlobal indiquera que le calcul est terminé. Tant qu'il dispose de travail à effectuer (indiqué par la fonction TravailLocal) le processeur exécute par bout le travail qu'il possède à l'aide de la fonction FaireUniteTravailLocal. Il vérifie après chaque unité de travail si un processeur cherche à lui voler du travail. Si oui, il met à jour l'état global (à l'aide de CommitLocalDansGlobal) et se place dans un état d'attente jusqu'à ce qu'aucun processeur ne cherche à le voler.

Lorsqu'un processeur n'a plus de travail, il met à jour l'état global et vérifie si l'algorithme est terminé. S'il ne l'est pas, un processeur qui possède du travail est choisi et est notifié qu'il va être volé. Lorsque le processeur volé est en attente d'être volé, le processeur voleur utilise la fonction Vol pour récupérer du travail.

Algorithme 1 Algorithme

```
1:  tant que TravailGlobal faire
2:    si TravailLocal alors
3:      tant que TravailLocal faire
4:        FaireUniteTravailLocal
5:        si Tentative de Vol alors
6:          CommitLocalDansGlobal
7:          Attendre que les voleurs soient partis
8:        fin si
9:      fin tant que
10:     CommitLocalDansGlobal
11:   sinon
12:     Vol
13:   fin si
14: fin tant que
```

Le modèle d'exécution de PaSTeL est actuellement implanté à l'aide de *threads* POSIX. Pour pouvoir constater du parallélisme, il est nécessaire que le noyau du système associe les *threads* POSIX à des *threads* noyaux. La création de *thread* étant coûteuse, des *threads* sont crées au début de l'exécution du programme et se placent en attente d'un algorithme à effectuer. Le *thread* qui appelle une fonction de PaSTeL commence par se déclarer auprès des *threads* de calculs. Puis, il exécute l'algorithme de vol de travail présenté précédemment. La synchronisation entre les *threads* peut être assurée à l'aide de *mutex* et conditions. Elle peut également être faite avec des *spinlocks*. Ces derniers provoquent une utilisation massive des ressources CPU et de la bande passante mémoire mais devraient permettre une plus grande réactivité et donc de meilleures performances et une accélération intéressante sur des données de plus petites tailles.

Remarquons que le mécanisme de vol de travail dans PaSTeL est synchrone. Lorsqu'un processeur veut en voler un autre, le voleur notifie le processeur cible qu'il va être victime d'un vol en mettant à jour l'état du processeur volé. Le processeur volé se place ensuite dans un état d'attente jusqu'à ce que tous les vols aient été commis. Un processus de vol asynchrone permet d'assurer une exécution rapide des calculs lorsqu'il n'y a pas de vol. De plus, savoir si un processeur veut nous voler peut se faire sans synchronisation, seul la mise à jour de l'état néccessite un verrouillage

de la mémoire. Les vols nécessitent alors une synchronisation entre voleur et volé. Mais tout algorithme à base de vol de travail essaye de minimiser le nombre total de vols, et ainsi minimise le surcoût lié à ce vol synchrone.

3.3. Algorithmes implantés

Pour implanter un algorithme dans PaSTeL, il suffit de fournir les fonctions utilisées dans l'Algorithme 1. Trois classes d'algorithmes sont actuellement implantées : *min_ element*, *merge* et *stable_ sort*. Chacune fonctionne sur n'importe quelle structure de données à accès aléatoire (supportant les *RandomAccessIterator* suivant le jargon de la STL). Nous présentons maintenant l'algorithme *two way merge* implanté en PaSTeL.

Chaque *thread* a une partie des tableaux à interclasser et interclasse un nombre fixe d'éléments dans la fonction FaireUniteTravailLocal. Elle maintient à jour le nombre d'éléments qui ont été interclassés par ce *thread* et modifie les parties des tableaux qui restent à interclasser. La fonction TravailLocal se contente de vérifier que le *thread* dispose encore de travail à effectuer. Pour détecter la terminaison de l'algorithme, chaque *thread* compte le nombre d'éléments qu'il a interclassé et met à jour l'état global dans CommitLocalDansGlobal. L'algorithme se termine lorsque tous les éléments ont été interclassés, ceci est vérifié par la fonction TravailGlobal.

L'opération de vol de travail effectuée dans la fonction Vol consiste à séparer le premier tableau en deux et à chercher le point de séparation dans le deuxième à l'aide d'une recherche dichotomique. Le voleur fusionnera les premières parties des deux tableaux tandis que le volé fusionnera les dernières parties des deux tableaux. Chaque *thread* calcule facilement la position à laquelle il doit interclasser les tableaux. Le vol s'effectue alors en temps logarithmique.

4. Expérimentation

Nous présentons dans cette Section une évaluation des performances de PaSTeL. Après avoir présenté les plate-formes d'expérimentation en Section 4.1, nous comparerons les performances de PaSTeL et de MCSTL sur un ordinateur portable en Section 4.2 et sur une machine de calcul en Section 4.3.

4.1. Préliminaires

La première machine qui sert à nos expériences est *nedni*. La machine est un ordinateur portable disposant d'un processeur Intel Core2 Duo T7100 cadencé à 1.8 GHz et de 2Go de mémoire vive. Le bus mémoire est cadencé à 667Mhz. La seconde machine qui sert à nos expériences est *idkoiff*. Cette machine de calcul est composée de huit processeurs Dual Core AMD Opteron 875, soit 16 cœurs au total (8 sont utilisés), à 2,2 Ghz et dispose de huit bancs de 4 Go de mémoire.

Sur les deux plate-formes, le noyau utilisé est Linux 2.6.22 en 64 bits. Le compilateur utilisé est gcc 4.2.1. Les options de compilation sont *-02* pour optimiser le code et *-DNDEBUG* pour désactiver les assertions.

4.2. Performances globales sur Core2 Duo

La première expérience a pour but de comparer les performances de MCSTL et de PaSTeL. Pour ce faire, les trois algorithmes présentés précédemment ont été exécutés sur *nedni* sur des instances aléatoires de taille fixe comprise entre un et 10^7 éléments sur les types de données *int*, *float* et *double* (rappelons que sur les machines 64bits, les *int* et les *float* font 4 octets alors que les *double* font 8 octets). Quatre méthodes sont exécutées pour comparaison : la STL fournie avec gcc, MCSTL, PaSTeL synchronisé avec des *mutexs* et PaSTeL synchronisé avec des *spinlocks*. Les méthodes parallèles ont été exécutées sur les deux cœurs. Chaque mesure est répétée 20 fois et à

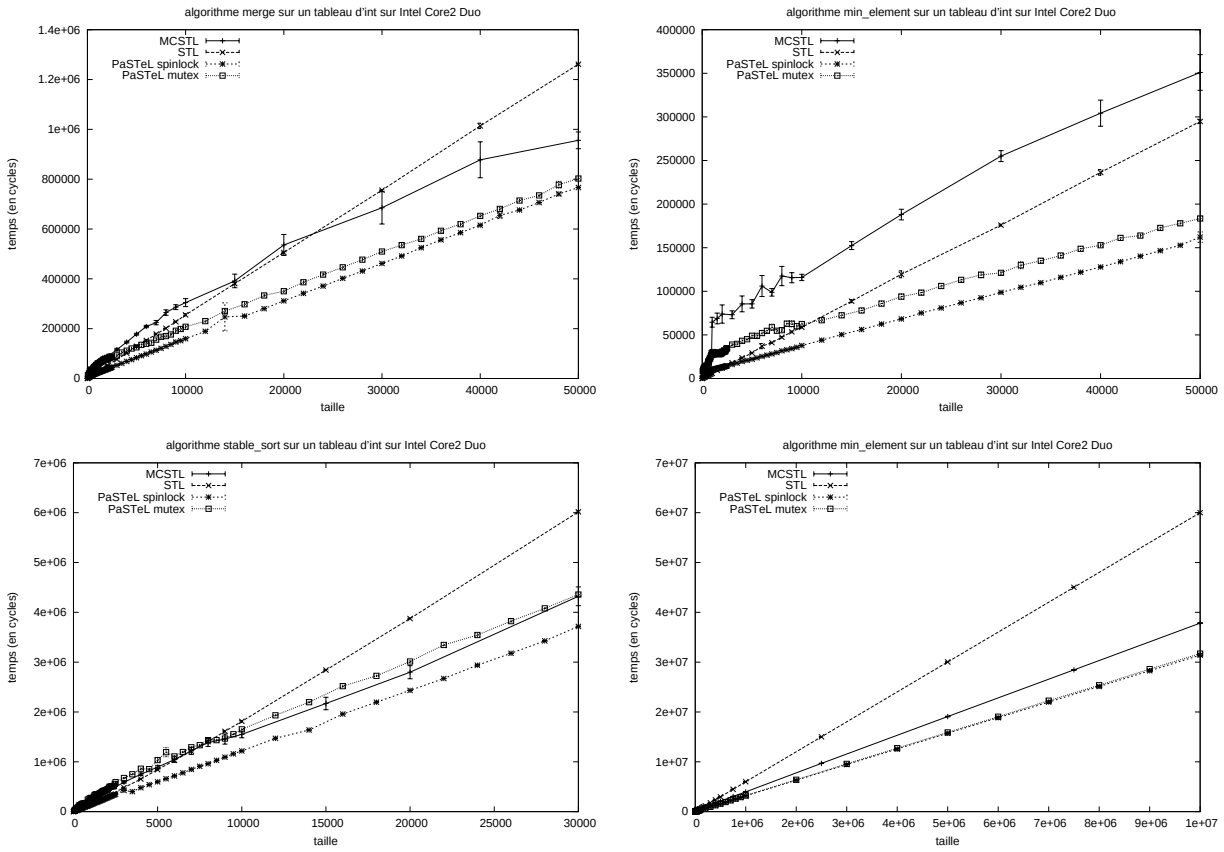

FIG. 1 – Performances comparées de PaSTeL, MCSTL et STL sur Core2 Duo (2 cœurs)

chaque résultat correspondent 20 exécutions. Les algorithmes de PaSTeL utilisent un paramètre supplémentaire, le grain de l'algorithme parallèle. Le grain a été fixé manuellement de la meilleure façon possible (soit à 50 itérations pour *min_ element*, 100 pour *merge* et 400 pour *stable_ sort*). La Figure 1 présente les moyennes et écart-types des temps d'exécution (en cycles) de chaque méthode en fonction de la taille des données pour des tailles inférieures à 50000 éléments pour le type de donnée *int* ainsi que les résultats de *min_ element* pour des tailles inférieure à 10^7. Les résultats obtenus sur les deux autres types de données sont comparables.

La première remarque est que PaSTeL synchronisé à l'aide de *spinlocks* est toujours plus rapide que PaSTeL synchronisé à l'aide de *mutexs*. Sur les trois algorithmes, MCSTL obtient des temps de calculs plus important que PaSTeL synchronisé en *spinlock*. On remarque également que PaSTeL a un surcoût fixe bien moins important que celui de MCSTL. De plus, on remarque à l'aide des intervalles de confiance que PaSTeL fournit des résultats bien plus stables que MCSTL. Finalement, synchronisé à l'aide de *mutexs*, PaSTeL obtient de meilleures performances que MCSTL sur les algorithmes *merge* et *min_ element*. Cependant, le cas de *stable_ sort* est différent. Le surcoût apporté par les *mutexs* rend l'algorithme légèrement plus lent que celui de MCSTL.

Revenons sur la différence entre PaSTeL et MCSTL sur les algorithmes. Commençons par *min_ element*. MCSTL implémente cet algorithme avec une technique de vol de travail proche de la notre. Bien que PaSTeL ait été configuré avec un grain optimal, cela ne suffit pas à expliquer la différence de performances. De trop nombreuses synchronisations ralentissent l'exécution de

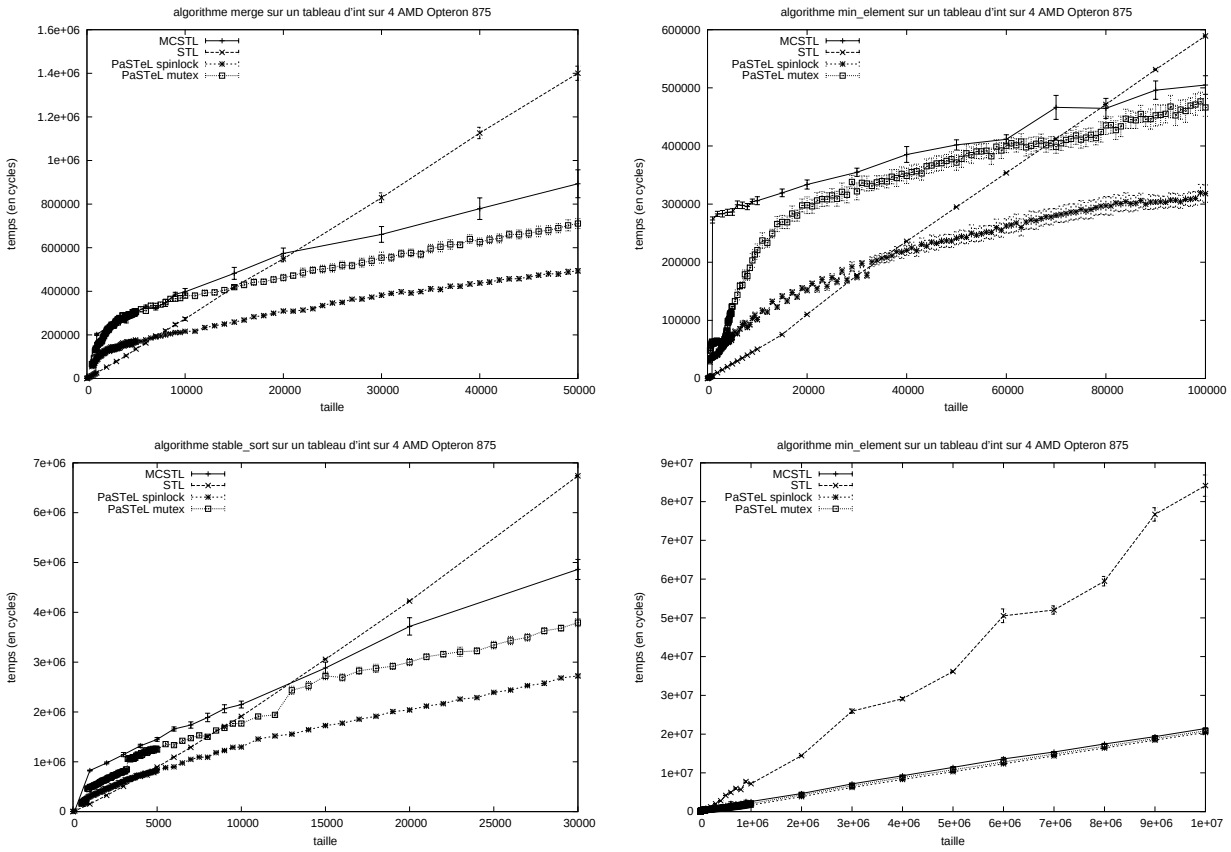

FIG. 2 – Performances comparées de PaSTeL, MCSTL et STL sur Opteron 875 (8 cœurs)

MCSTL. La supériorité de PaSTeL est plus visible sur l'étude de *merge*. En effet, le moteur de vol de travail de MCSTL ne permet pas d'exprimer cet algorithme. La travail est alors découpé statiquement. Cependant, cette découpe peut ne pas être équilibrée. Dans PaSTeL, *merge* s'écrit très simplement et fournit de bonnes performances.

Notre objectif est de démontrer qu'il est possible de paralléliser des algorithmes de façon à être efficace sur de petits volumes de données. Sur *nedni*, PaSTeL synchronisé à l'aide de *spinlocks* obtient des temps de calcul plus faibles que la STL sur des tableaux d'au moins 4000 *int* pour l'algorithme *min_ element*, d'au moins deux fois 800 *int* pour l'algorithme *merge* et d'au moins 1000 *int* pour l'algorithme de tri. La mécanique de PaSTeL est donc fortement réactive.

4.3. Performances globales sur Opteron 875

La Figure 2 présente les mêmes expériences sur la machine *idkoiff* en utilisant 8 cœurs. Le but de cette expérience est de vérifier le comportement de PaSTeL dans une architecture hiérarchique avec plusieurs processeurs. Le grain des algorithmes parallèles a ici été fixé à 400 itérations.

Nous remarquons que globalement les résultats sont similaires à ceux obtenus sur Core2 bien que les ordres de grandeur soient différents. La mécanique de PaSTeL semble alors robuste au passage à l'echelle. Cependant, le cas du tri stable est particulier. Bien que cela n'apparaisse que pour des données de tailles supérieur à 10^6 éléments, la présence des barrières de synchronisation dans l'algorithme PaSTeL semble diminuer significativement les performances globales lorsque le nombre de cœurs utilisés est important. MCSTL devient alors un meilleur choix.

On remarque que sur *idkoiff*, les trois mécaniques parallèles obtiennent des performances similaires sur l'algorithme *min_ element* pour des données de taille supérieure à 10^6 éléments. L'accélération est alors de 4 alors que 8 cœurs sont utilisés. Bien qu'une étude plus approfondie soit nécessaire, on peut raisonnablement conclure que la bande passante mémoire a été saturée.

5. Conclusion

Dans cet article, nous avons présenté PaSTeL, une bibliothèque d'algorithmes parallèles de la STL. Pour les programmeurs utilisateurs de la STL, elle permet à coût quasiment nul en développement d'exploiter le parallélisme offert par les cœurs des processeurs. PaSTeL s'inscrit dans la même lignée que d'autres travaux menés sur la parallélisation des algorithmes de la STL. PaSTeL repose sur un modèle de programmation simple et un modèle d'exécution basé sur du vol de travail. Pour la mise en œuvre du vol de travail, PaSTeL se distingue des bibliothèques concurrentes par des threads réactifs aux évènements de synchronisation. Des expériences ont été faites et ont montré la pertinence de cette approche, en particulier pour des exécutions parallèles de courte durée.
Ces premiers résultats nous encouragent à poursuivre nos investigations vers plusieurs pistes. D'abord celle consistant à fixer la granularité pendant l'exécution et non à la compilation comme cela est fait actuellement. Au début de l'algorithme, il faudrait ainsi pouvoir effectuer quelques mesures pour ensuite calibrer la taille de l'unité de travail local. Une autre piste consiste à estimer à l'exécution s'il est intéressant d'utiliser PaSTeL ou non. Ces deux points nécessitent de modéliser finement le comportement des machines parallèles.

Bibliographie

1. Threading Building Blocks. http ://threadingbuildingblocks.org/, 2007.
2. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL : An Adaptive, Generic Parallel C++ Library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 193–208, August 2001.
3. 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 publishing, editor, *Parallel Symbolic Computation'07 (PASCO'07)*, July 2007.
4. R. Dementiev, L. Kettner, and P. Sanders. STXXL : standard template library for XXL data sets. *Software : Practice and Experience*, August 2007.
5. T. Furtak, J. N. Amaral, and R. Niewiadomski. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the 19th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 348–357, 2007.
6. J. Singler, P. Sanders, and F. Putze. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, June 1998. Vol. 33, No. 5.
7. J. Singler, P. Sanders, and F. Putze. MCSTL : The Multi-Core Standard Template Library. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer, 2007. http ://algo2.iti.uni-karlsruhe.de/singler/mcstl/.