



École Nationale Supérieure d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, FRANCE

TP n°3

Niveau	3 ^{ème} année
Parcours	Électronique et Physique Appliquée
Unité d'enseignement	2E1AC2 - Architectures pour le calcul [Parallélisme]
Responsables	Emmanuel Cagniot emmanuel.cagniot@ensicaen.fr Hugo Descoubes hugo.descoubes@ensicaen.fr

1 Exercice

Ce TP aborde la parallélisation d'un algorithme de la bibliothèque standard C++ en OPENMP. Il suppose la connaissance de la notion d'itérateur et sa traduction en C++.

L'algorithme `count_if` (module `algorithm` de la bibliothèque standard C++) permet de comptabiliser les éléments d'un conteneur satisfaisant un prédicat unaire. La figure 1 présente la définition de cet algorithme.

Le prédicat unaire `pred` est invoqué sur tous les éléments de l'intervalle `[first, last[`, `first` et `last` étant deux itérateurs de types `InputIterator`. Rappelons que les opérations permises par ce type d'itérateur sont la pré-incrémentation (`++ first`), la post-incrémentation (`first ++`), le dé-référencement `*first`, le test d'égalité (`first == last`) et celui de l'inégalité (`first != last`).

La parallélisation de cet algorithme en OPENMP pose plusieurs problèmes :

1. la condition de continuation utilise un test d'inégalité (`first != last`). Comme les itérateurs de classe `InputIterator` ne proposent que des tests d'égalité ou d'inégalité, il n'est à priori pas possible de transformer la boucle `while` en une boucle `for` puis de paralléliser cette dernière via une directive de type `omp for` ;
2. le nombre d'éléments de l'intervalle `[first, last[` ne peut être calculé directement (il faut parcourir les éléments correspondants afin de les comptabiliser) ;
3. la nature du prédicat unaire `pred` est inconnue (s'agit-il d'une fonction dont le temps de calcul est fixe quel que soit son argument ou au contraire d'une fonction dont le temps de calcul dépend de son argument ?).

```

1 template < typename InputIterator , typename UnaryPredicate >
2 typename iterator_traits< InputIterator >::difference_type
3 count_if(InputIterator first , InputIterator last , UnaryPredicate pred) {
4     typename iterator_traits< InputIterator >::difference_type ret = 0;
5     while (first != last) {
6         if (pred(*first)) ret ++;
7         first ++;
8     }
9     return ret;
10 }

```

FIGURE 1 – Algorithme `count_if` de la bibliothèque standard C++.

Nous sommes à priori mal partis mais tout n'est cependant pas si sombre. En effet, l'algorithme `count_if` peut être instancié par des itérateurs de classe `InputIterator` mais également pas des itérateurs de classes dérivées, notamment `RandomAccessIterator`. Ces derniers sont l'apanage des structures de données où l'accès à un élément via son rang s'effectue en temps constant grâce à l'utilisation d'une zone d'adresses en mémoire contiguë. Sont concernés les tableau de capacité fixe (classiques ou de classe `array`) et ceux de capacité dynamique (classe `vector`).

Outre les possibilités des `InputIterator`, les `RandomAccessIterator` proposent, entre autres, les opérateurs relationnels manquants (<, <=, > et >=) ainsi que la différence entre deux itérateurs (`last - first`) dont le résultat est le nombre d'éléments de l'intervalle semi-ouvert correspondant. Par conséquent, il est possible de proposer une spécialisation relativement simple de l'algorithme `count_if` pour les `RandomAccessIterator`. Pour toutes les autres classes d'itérateurs (le cas général), nous allons devoir nous torturer sérieusement les méninges.

Deux possibilités d'implémentation s'offrent à nous :

1. sous forme de fonctions génériques (implémentation analogue à celle du module `algorithm` de la bibliothèque standard C++);
2. sous forme de méthodes de classes (mot-clé `static`) génériques d'une classe non générique (implémentation analogue à celle des classes `Arrays` ou `Collections` de la bibliothèque standard JAVA).

Nous optons pour la seconde solution qui offre (selon nous) d'avantage de sécurité grâce à l'utilisation des droits d'accès.

L'archive `tp3.tar.gz` contient un squelette incomplet de l'application à réaliser. Sa structure est la suivante :

`src/include/Metrics.hpp` : fichier en-tête contenant la déclaration d'une classe `Metrics` permettant de calculer des facteurs d'accélération et d'efficacité;

`src/include/CountIf.hpp` : fichier en-tête contenant le squelette incomplet de notre version parallèle de l'algorithme `count_if`. Cette classe non générique propose une méthode générique `apply` qui opère en deux temps :

1. elle instancie une sous-classe générique `Impl` avec le type des itérateurs reçus (il existe une sous-classe `Impl` dédiée au cas général des `InputIterator` et une autre dédiée au cas particulier des `RandomAccessIterator`);
2. elle invoque la méthode générique `apply` de cette sous-classe.

La sous-classe `Impl` dédiée aux `InputIterator` propose trois méthodes génériques `strategyA`, `strategyB` et `strategyC`, chacune implémentant une stratégie de parallélisation spécifique. Sa méthode `apply` invoquera l'une des ces trois méthodes afin d'évaluer la pertinence de chaque.

`src/Metrics.cpp` : définition de la classe `Metrics`;

src/testCountIf.cpp : programme de démonstration de l’algorithme parallèle **CountIf**. Cette application teste (calcul des facteurs d’accélération et d’efficacité) la version parallèle de notre algorithme sur deux types de conteneurs (listes et tableaux de taille fixe) et deux types de prédicats unaires. Le temps de calcul du premier (**estPair**) est fixe tandis que celui du second (**pgcd21Vaut3**) varie très fortement ;

CMakeLists.txt : script permettant de générer le makefile de l’application via l’utilitaire **cmake** ;

Lisezmoi.txt : fichier texte décrivant la procédure de génération du makefile et celle de la configuration du fichier **CMakeCache.txt** produit par **cmake**.

1.1 Question

Comme elle représente une spécialisation totale, la définition de la sous-classe **Impl** dédiée au cas particulier des **RandomAccessIterator** est placée à l’extérieur de la classe **CountIf** (ainsi que l’exige la norme C++). Complétez la définition de sa méthode **apply** via une boucle **for** parallèle et la clause la plus adéquate pour la répartition des itérations (clause que vous devez pouvoir justifier).

1.2 Question

Démontrez que les performances obtenues pour cette première version parallèle seront au rendez-vous quel que soit le type des éléments de l’intervalle **[first, last[** et quelle que soit la nature du prédicat **pred** (en d’autres termes, cette première version peut être mise en service).

Concernant les **InputIterator**, une première stratégie de parallélisation consiste, dans un premier temps, à constituer un tableau (de taille fixe) dont les éléments contiennent des pointeurs vers les éléments de l’intervalle **[first, last[**. Dans un second temps, les éléments de ce tableau sont traités via une boucle parallèle **for** et la clause de répartition des itérations la plus adéquate.

Le problème posé par cette stratégie est que le nombre d’éléments de l’intervalle **[first, last[** ne peut être calculé a priori à moins de le parcourir (ce qui s’avérerait exorbitant en termes d’*overhead*). Par conséquent, il est impossible d’utiliser un tableau de taille fixe. Envisager un tableau de taille dynamique (classe **vector**) serait suicidaire en raison des opérations de ré-allocation-recopie enclenchées lorsque la taille du tableau vient à excéder sa capacité.

La solution à ce problème consiste à tronçonner l’intervalle **[first, last[** en segments d’une taille arbitraire fixe (éventuellement calculée en fonction du nombre de threads disponibles). Notre stratégie de parallélisation revient alors à constituer un segment de pointeurs, à traiter ce segment via une boucle parallèle **for** et à recommencer ainsi jusqu’à avoir traité l’intégralité des éléments de l’intervalle **[first, last[**.

1.3 Question

Complétez la définition de la méthode générique **strategyA** qui implémente la stratégie de parallélisation décrite ci-dessus. Dans un premier temps, la boucle de tronçonnage sera séquentielle tandis que seul le traitement du tronçon fera l’objet d’une région parallèle (**pragma omp parallel for schedule(...)**).

1.4 Question

Faites en sorte que la méthode **apply** invoque la méthode **strategyA** et observez ses performances en faisant éventuellement varier la taille des tronçons. Qu’en concluez-vous ?

L'écriture de la méthode générique **strategyA** demandée précédemment vous place à la merci de votre compilateur (n'oubliez pas qu'OpenMP n'est pas une bibliothèque mais un standard). En effet, il peut traduire ce code de façon intelligente ou particulièrement stupide :

- s'il est intelligent, il ré-écrit l'intégralité de la boucle de tronçonnage pour l'incorporer dans une région parallèle. Par conséquent, les threads créés en entrée de cette région seront disponibles tout le temps d'exécution de la boucle de tronçonnage ;
- s'il est stupide, il traduira votre code mot pour mot c'est à dire que des threads risquent d'être créés en entrée de la boucle parallèle **for** puis détruits à l'issue et ce à chaque itération de la boucle de tronçonnage (c'est pour cette raison que la norme OpenMP recommande aux développeurs de toujours maximiser leurs régions parallèles).

1.5 Question

Ré-écrivez la définition de la méthode générique **strategyA** afin que la boucle de tronçonnage soit placée à l'intérieur d'une région parallèle (une attention très particulière devra être apportée à la synchronisation de vos différents threads).

1.6 Question

Vous constatez que les performances de cette nouvelle version de **strategyA** restent décevantes (si elles sont identiques à celles de la version précédente alors votre compilateur avait bien fait les choses). quoiqu'un peu meilleures dans le cas de **pgcd21Vaut3** : expliquez les raisons de cet échec.

Une seconde stratégie de parallélisation consiste à exploiter la notion de tâche introduite par la norme 3.0 d'OpenMP pour tenter de reproduire la clause de répartition des itérations **schedule(dynamic, size)** des boucles parallèles **for**. Une taille **size = 1** n'est bien sûr pas envisageable puisque le coup de création et de synchronisation d'un nombre de tâches trop important serait alors largement supérieur aux gains produit par la parallélisation. Nous allons donc fixer la taille du paramètre **size** à une valeur arbitraire (pouvant éventuellement dépendre du nombre de threads disponibles) et ainsi tronçonner l'intervalle **[first, last[**. Chaque tâche ainsi créée invoquera l'algorithme **count_if** de la bibliothèque standard sur son sous-intervalle. Le résultat de cet appel sera ensuite cumulé dans un compteur global commun à toutes les tâches (attention : la clause **reduction** n'est malheureusement applicable qu'aux threads mais pas aux tâches : il faut utiliser **atomic**).

1.7 Question

Complétez la définition de la méthode générique **strategyB** qui implémente la stratégie de parallélisation décrite ci-dessus. Le tronçonnage de l'intervalle **[first, last[** et la création des tâches correspondantes seront l'œuvre d'un seul et unique thread.

1.8 Question

Faites en sorte que la méthode **apply** invoque la méthode **strategyB** et observez ses performances en faisant éventuellement varier la taille des tronçons. Qu'en concluez-vous ?

Il est peut-être possible d'optimiser la méthode **strategyB** comme suit :

1. les tâches étant créées à l'intérieur d'un bloc **single**, tous les autres threads sont placés en attente sur la barrière de synchronisation implicite placée à la sortie de ce bloc. Par conséquent, ces derniers sont obligés d'attendre que toutes les tâches soient créées pour commencer à les traiter. Une clause **nowait** placée derrière le mot-clé **single** permet de lever cette barrière ;
2. lorsqu'une tâche commence son exécution avec un thread et qu'elle est mise en sommeil pour une raison particulière, elle ne peut reprendre son exécution qu'avec le même thread. Par conséquent, tant que ce thread n'est pas disponible, elle reste en attente même si elle est prête à reprendre son exécution. Toute implémentation prévoit un nombre maximum de tâches en attente dans une file

de tâches à exécuter. Lorsque ce plafond est atteint, le thread chargé de créer les tâches suspend le processus et commence lui-même à traiter les tâches en attente. Quand un certain nombre de tâches ont été sorties de la file, il reprend le processus de création. Par conséquent, si ce thread particulier a commencé à traiter des tâches puis repris le processus de création, toutes les tâches qu'il a commencées mais qu'il n'a pas terminées restent en attente. La clause **untied** placée derrière le mot clé **task** permet de briser le lien entre une tâche et son thread : une tâche qui a commencé son exécution avec un thread puis qui a été mise en sommeil peut être reprise par un autre thread.

1.9 Question

Incorporez les optimisations décrites ci-dessus à votre code.

1.10 Question

Vous constatez que malgré les optimisations mises en place, les performances de **strategyB** restent décevantes : expliquez les raisons de cet échec.

Une troisième stratégie de parallélisation consiste à reproduire la clause de répartition des itérations **schedule(static, 1)** des boucles parallèles **for**, c'est à dire une répartition cyclique des itérations de l'intervalle **[first, last[** sur l'ensemble des threads disponibles. Pour cela, chaque thread doit d'abord se placer sur son premier élément. Ensuite, si \mathcal{P} désigne le nombre de threads disponibles alors chaque thread doit avancer de \mathcal{P} positions dans l'intervalle **[first, last[**, traiter l'élément correspondant et recommencer ainsi jusqu'à dépasser la position **last**.

1.11 Question

Complétez la définition de la méthode générique **strategyC** qui implémente la stratégie de parallélisation décrite ci-dessus.

1.12 Question

Faite en sorte que la méthode **apply** invoque la méthode **strategyC** et observez ses performances. Qu'en concluez-vous ?

1.13 Question

Les trois stratégies proposées ont échoué, ce qui impose de ne fournir qu'une version parallèle pour les itérateurs de type tableau (c'est d'ailleurs ce qui est fait lorsque vous activez l'option « mode expérimental » du compilateur **g++**). Cependant, la stratégie implémentée par **strategyC** est de loin la meilleure : expliquez pourquoi et indiquez dans quelle contexte celle-ci devient une stratégie de parallélisation gagnante.