

# ISIMA 3<sup>ème</sup> année - MODL/C++

## TP 6 : Multithreading

### Exercice 1 - Couche bas niveau

L'objectif ici est de manipuler les éléments fondamentaux qui permettent la programmation parallèle par les threads depuis C++11, à savoir les classes `std::thread` et `std::mutex`. Notre exemple consiste à paralléliser le calcul de la multiplication de deux vecteurs. Le code séquentiel (`sequentiel.cpp`) est fourni et devra être progressivement adapté en une version parallèle plus rapide.

Pour percevoir les problématiques du calcul parallèle, les vecteurs ne contiennent pas d'éléments de type `double` mais des objets de la classe `Nombre` qui est volontairement lente dans ses opérations de construction, d'affectation et de calcul (opérateurs `+`, `-`, `*` et `/`). Mais avant de commencer, il vous faut tester le fonctionnement des threads, des mutex et des lambdas.

- 1) Reprendre un à un les exemples du cours pour bien comprendre le fonctionnement des threads et des mutex. Commencer avec le lancement de plusieurs threads, d'abord en version appel de fonction (*slide 343*), puis en version expression lambda (*slide 344*). L'affichage est buggé (relancer éventuellement plusieurs fois votre programme pour bien identifier le souci), donc compléter le code pour synchroniser l'utilisation de la sortie standard (*slide 347*) à l'aide d'un mutex.
- 2) Intéressons-nous maintenant à notre code séquentiel. Pour le préparer à la version parallèle, nous commençons par abstraire le mécanisme de boucle dans une fonction : écrire `for_sequentiel(x,y,callable)` qui exécute un «*callable*» (i.e. fonction, foncteur ou lambda, cf. cours) à chaque itération d'une boucle allant d'une valeur `x` à une valeur `y`. Cette fonction sera appelée de la manière suivante :

```
for_sequentiel(0,taille, [&] (unsigned i) { a[i] = ++compteur; } );
```

qui est équivalente au code :

```
for (unsigned i = 0; i<taille; ++i) a[i] = ++compteur;
```

Remplacer les 3 boucles `for` de l'exemple par l'appel à cette fonction (conserver le fichier `sequentiel.cpp` et mettre la version modifiée dans `parallelele.cpp`). Remarque : cette fonction est générique car le type du callable doit être un paramètre.

- 3) Ecrire une version parallèle de la boucle `for`, par exemple pour  $N = 4$  threads, où la boucle initiale est découpée en 4 sous-boucles, chacune affectée à l'un des threads. Votre fonction `for_parallelele`, qui aura le même prototype que la fonction `for_sequentiel`, devra réaliser l'équivalent du code suivant :

```
- Thread 1 : for (unsigned i = 0; i<taille/4; ++i) a[i] = ++compteur;
- Thread 2 : for (unsigned i = taille/4; i<2*taille/4; ++i) a[i] = ++compteur;
- Thread 3 : for (unsigned i = 2*taille/4; i<3*taille/4; ++i) a[i] = ++compteur;
- Thread 4 (principal) : for (unsigned i = 3*taille/4; i<taille; ++i) a[i] = ++compteur;
```

pour l'appel à la fonction de la manière suivante :

```
for_parallelele<4>(0,taille, [&] (unsigned i) { a[i] = ++compteur; } );
```

Remplacer les 3 boucles `for_sequentiel` par l'appel à cette fonction. Remarque : cette fonction est générique avec deux paramètres, le nombre de threads et le type du callable.

- 4) Vous avez dû remarquer que le compteur a un comportement chaotique, lié au fait qu'il est partagé par les threads. Il faut donc synchroniser l'accès à cette ressource par un mutex. Ecrire une fonction qui permet un accès contrôlé à l'incrément du compteur. Essayer d'abord sans verrou (utilisation directe d'un mutex), puis avec un verrou (`std::lock_guard`).

Vous pouvez comparer les temps d'exécutions des deux versions (séquentiel et parallèle) du code à l'aide de la commande `time` sous UNIX.

## **Exercice 2 - Couche intermédiaire**

L'objectif maintenant est de manipuler la couche intermédiaire du parallélisme en C++ qui permet de masquer le mécanisme des threads. Nous allons considérer un exemple similaire au précédent mais avec un calcul plus complexe :  $g[i] = a[i]*b[i] + c[i]/d[i] - e[i]*f[i]$ . Avant de commencer, il est nécessaire de tester le fonctionnement de la fonction `std::async` et de la classe `std::future`.

- 1) Reprendre un à un les exemples du cours pour bien comprendre le fonctionnement de la couche intermédiaire. Commencer par la fonction `async` (*slides 356-357*), puis avec la classe `future` (*slide 359*).
- 2) Créer et initialiser les 6 vecteurs  $a, b, \dots, f$  de la même manière que les vecteur  $a$  et  $b$  de l'exemple précédent. Ecrire la boucle qui permet de calculer les valeurs du vecteur  $g$ . Vous comparerez les temps d'exécution des versions séquentielle (utilisant `for_sequentiel`) et parallèle (utilisant `for_parallele`, 6 threads).
- 3) On expérimente maintenant une autre manière de paralléliser le calcul. Chaque boucle d'initialisation d'un vecteur est exécutée en séquentiel sur un thread différent (donc 6 threads s'exécuteront en parallèle). Utiliser la fonction `async` pour lancer chaque tâche en parallèle et des objets `future` pour attendre les threads.
- 4) Lancer chacun des calculs  $a[i]*b[i]$ ,  $c[i]/d[i]$  et  $e[i]*f[i]$  sur un thread différent, également à l'aide de la fonction `async`. Comme pour la question précédente, chaque thread sera chargé de traiter tous les éléments des vecteurs. Seulement 3 threads ont été lancés en parallèle, il est donc possible d'exécuter leur tâche à l'aide d'une boucle parallèle sur 2 threads, afin d'obtenir un total de 6 threads :

- Thread 1 :  $a[i] = a[i]*b[i]$  (de 0 à  $taille/2$ )
- Thread 2 :  $a[i] = a[i]*b[i]$  (de  $taille/2$  à  $taille$ )
- Thread 3 :  $c[i] = c[i]/d[i]$  (de 0 à  $taille/2$ )
- Thread 4 :  $c[i] = c[i]/d[i]$  (de  $taille/2$  à  $taille$ )
- Thread 5 :  $e[i] = e[i]*f[i]$  (de 0 à  $taille/2$ )
- Thread 6 (principal) :  $e[i] = e[i]*f[i]$  (de  $taille/2$  à  $taille$ )

Remarque : on réutilise les vecteurs  $a, c$  et  $e$  pour stocker les calculs intermédiaires.

Attendre la fin de tous les threads avant de lancer le calcul final ( $g[i] = a[i] + c[i] - e[i]$ ) sur une boucle parallèle (6 threads). Le temps de calcul devrait être sensiblement équivalent à celui de la première version parallèle.