TP HPC 2

Master Informatique - 1ère année

Année 2020-2021

L'objectif de ce TP est d'expérimenter la programmation multi-threads au travers de petits exemples.

Préliminaires

Récupérez la version initiale du fichier source (`exo1.cpp`) joint à cet énoncé. Vous aurez à le compléter au fur et à mesure des exercices.

Exercice 1

1. Compilez (la commande de compilation est fournie en commentaire dans le fichier) et exécutez une première fois l'application `exo1`. A ce stade, seuls des messages issus de la fonctions `main` apparaissent et il est donc difficile de savoir si les threads ont bien été créés et exécutés.
2. Ajouter, dans chaque thread, le code nécessaire à l'affichage d'un message indiquant son démarrage et son identifiant. Après exécution (lancez l'application plusieurs fois), vous devez noter les points suisvants :
   — la création et le lancement des threads par le processus principal ne sont pas bloquants : le message `''Milieu du main''` apparaissant souvent avant les messages issus des threads ;
   — la méthode `join` est bien bloquante pour le processus principal, puisque le message `''Fin du main''` apparaît bien toujours après la fin de tous les threads ;
   — le texte des différents messages est parfois mélangé à l'affichage. Ce point sera traité dans l'exercice suivant.
3. le calcul effectué dans chaque thread étant ici très rapide, ajoutez une mise en sommeil d'une seconde à la fin de chacun d'entre-eux et vérifiez que le comportement attendu, par rapport à l'affichage des messages, reste cohérent.

Exercice 2

Dans cet exercice, vous allez tout d'abord modifier le code de votre fonction `main` de telle sorte qu'elle crée un tableau de 10 threads, les lance et attend la fin de leur exécution.

Après avoir testé votre application (plusieurs fois), vous noterez que davantage de messages sont mélangés au moment de l'affichage. Ceci provient du fait que le flot de sortie est une ressource partagée par tous les processus. Leur accès concurrent à cette ressource se fait sans aucune protection, conduisant à un mélange des messages affichés. Pour résoudre ce problème, il sera nécessaire de protéger les opérations sur `cout` par un sémaphore d'exclusion mutuelle :

— déclarez une variable globale de type `mutex` ;
— modifiez votre code de telle sorte que, là où c'est nécessaire, l'accès au flot de sortie `cout` soit protégé par ce sémaphore.

Exercice 3

Dans un premier temps, recopiez le code utilisé dans les deux exercices précédents, dans un fichier source nommé `exo3.cpp`, ceci afin de conserver une copie des développements réalisés. Modifiez ensuite ce nouveau code de la manière suivante :

— ajoutez-y une fonction permettant de calculer la somme des **racines carrées** des valeurs entières comprises entre deux entiers fournis en paramètres (à l'image de la fonction `somme` fournie) ;
— complétez votre code par une fonction permettant de calculer la somme des **racines carrées** multipliées par le **logarithme** des valeurs entières comprises entre deux entiers fournis en paramètres ;

— complétez les 3 fonctions de telle sorte qu'elles affichent le résultat de la somme dont elles ont la charge à la fin de leur exécution. A titre d'indication, vous devez obtenir les résultats suivants, pour une valeur de $N = 100000000$:

```
somme - res = 5e+15
somme racines - res = 6.66667e+11
somme (racines x log) - res = 1.1836e+13
```

— modifiez votre fonction `main` de telle sorte qu'elle lance le calcul des trois fonctions de sommation dont vous disposez désormais, l'une après l'autre. Vous mesurerez le temps d'exécution de cette version séquentielle de votre application, à l'aide de la bibliothèque <**chrono**> et vous l'afficherez sous la forme suivante :

```
tps de calcul sequentiel : 0.672272 s
```

— dans la mesure où les 3 fonctions semblent effectuer des opérations de même complexité, vous allez compléter votre application pour que, à la suite du code séquentiel, elle exécute à nouveau les 3 fonctions de calcul, mais cette fois dans 3 threads distincts (une fonction par thread). L'idée est alors d'accélérer le temps global d'exécution, en effectuant les 3 sommes sur 3 coeurs distincts. Vous mesurerez et afficherez le temps d'exécution obtenu après complétion des 3 threads. A titre d'indication, vous devez obtenir les résultats équivalents au résultats présentés ci-dessous (les temps obtenus différeront évidemment selon le processeur sur lequel le code aura été exécuté), pour une valeur de $N = 100000000$:

```
tps de calcul sequentiel : 0.672272 s
tps de calcul parallèle : 0.474243 s
```

L'accélération obtenue ici est bien loin du facteur 3 que l'on pouvait s'attendre à obtenir. La raison en est que les 3 fonctions ont un coût de calcul très différent. L'accélération obtenue dépend alors du thread qui est le plus lent ... Pour vérifier cette assertion, modifiez le code de vos 3 fonctions, de manière à ce qu'elles mesurent le temps nécessaire à l'évaluation de chaque somme (et uniquement les sommes ...), puis affiche ce temps à la fin de leur exécution. A titre d'information, vous devez obtenir un affichage similaire à celui figurant ci-après (les temps obtenus différeront évidemment selon le processeur sur lequel le code aura été exécuté) :

```
tps de calcul sequentiel : 0.672272 s
fin thread somme - res = 5e+15 en 0.0838422 s
fin thread somme racines - res = 6.66667e+11 en 0.125692 s
fin thread somme (racines x log) - res = 1.1836e+13 en 0.474096 s
tps de calcul parallèle : 0.474243 s
```

Exercice 4

Recopiez le code de l'exercice précédent dans une fichier source nommé `exo4.cpp`, afin de conserver les développements déjà effectués.

Dans cet exercice, vous allez mettre en place une stratégie d'**équilibrage de charge statique**, qui va consister à faire exécuter par chaque thread créé, une partie des calculs de chacune des sommes. Le principe sera de découper chacune des 3 sommes en N sous-sommes, N étant le nombre de threads à utiliser. Chaque thread exécutera une sous-somme pour chacune des 3 sommes initiales, et ajoutera les résultats obtenus aux variables partagées qui cumuleront les sous-sommes.

— écrire le code de la fonction `void sommes(int i0, int i1)`, dont le rôle est de calculer successivement chacune des 3 sommes précédentes, entre les indices `i0` et `i1` ;

— déclarez 3 variables globales, de type `double`, qui permettront de cumuler les différentes sous-sommes. Elles seront initialisées dans la fonction `main` ;

— complétez le code de la fonction `sommes` pour qu'elle ajoute chacune des sous-sommes calculées aux 3 variables déclarées ci-dessus. Vous prendrez garde à protéger l'accès en écriture à ces variables par un sémaphore ...

— Complétez votre fonction `main` de telle sorte qu'elle déclare un tableau de `NBT` threads (constante à définir et initialiser dans un premier temps à 1), puis lance le calcul sur `NBT` threads, chacun d'entre-eux ayant la charge d'un intervalle disjoint différent (il s'agit alors de découper l'intervalle $[1, N]$ en `NBT` sous-intervalles disjoints ;

— après avoir vérifié que les 3 sommes finales obtenues sont bien conformes à ce que vous obteniez dans l'exercice précédent, procédez à des mesures de temps (i) dans la fonction `main` et (ii) dans les threads, afin de vérifier que les temps obtenus correpondent à l'objectif visé ;

— dressez ensuite un tableau des accélérations obtenues en fonction du nombre de threads utilisés, limité au nombre de coeurs disponibles sur votre machine. Prenez garde à ne pas avoir trop d'applications en cours, afin de limiter les biais de mesure.

Exercice 5 (optionnel)

Après avoir recopié le code de l'exercice 4 dans un fichier nommé `exo5.cpp`, modifiez le code de la fonction `sommes` pour l'accélérer à l'aide de routines vectorielles. On précise les points suivants :

- il existe un *intrinsic* pour le calcul d'une racine carrée, mais pas pour le log (disponible uniquement pour le compilateur intel) ;
- afin d'être cohérent avec la version précédente, il ne faut pas réutiliser des calculs déjà faits (par exemple la recine carrée) ;
- contrairement à ce qui a été fait lors du premier TP, les données à utiliser ne sont pas présentes en mémoire. Elles doivent donc être générées à la volée, lorsque leur calcul est requis. Ceci implique un surcoût non négligeable, pour lequel il convient d'étudier la manière la plus rapide de les obtenir.