TP - ASR7 Programmation Concurrente
—
Première utilisation de threads

Sylvain Brandel, Guillaume Damiand, Laurent Lefèvre, Matthieu Moy, Grégoire Pichon

Printemps 2022

I Premiers pas avec les threads

Vous devez paralléliser un petit programme : `lancement.cpp` disponible sur le site de l'UE. Ce dernier répète un certain nombre de fois un calcul qui n'a pas d'importance. Le but de l'exercice est de savoir si vous pouvez paralléliser les appels à la fonction `fct()`, récupérer le résultat (qui est le temps écoulé pendant le calcul d'après la fonction) et comparer ce temps avec le temps réellement écoulé.

Pour cela :

Q.I.1) - Dans la fonction `main()` remplacez la boucle qui appelle `nbthreads` fois la fonction `fct()` par le lancement de `nbthreads` threads qui exécutent la fonction.

Q.I.2) - Si cela ne vous est pas déjà arrivé, faites une erreur d'argument entre le lancement du thread et la fonction afin de vous habituer à lire les messages d'erreur de compilation. Par exemple, essayez un passage par référence en oubliant `std::ref()` au moment de l'appel : le message d'erreur est rarement convivial !

Q.I.3) - Normalement, dès qu'il y a un nombre de threads important, vous pouvez observer que les messages qu'ils affichent se mélangent. Sachant que les fonctions système sont prévues pour fonctionner en environnement multithread, pourquoi y a-t-il ce mélange et comment l'éviter ?

> On observe le problème sur les appels chaînes
> `cout << num << ": lancement de la fonction pour "<< nbtours << "itérations"` et pas sur les `printf()`. En fait, chaque appel gère correctement les affichages, mais un `cout` fait plusieurs affichages séparés. En fait il y en a 1 par `<<`. Les appels à l'opérateur `<<` se mélangent d'autant plus que ce sont des appels système donc qu'ils sont lents.
>
> Pour éviter le problème, on peut construire la chaîne de caractères à part et tout afficher d'un seul coup. C'est ce que fait `fprintf()`. L'autre solution est d'utiliser les `ostringstream` pour construire la chaîne avant de l'envoyer sur `cout` [1].

Q.I.4) - À la fin des threads obtenez le résultat de la fonction et affichez le temps que chaque thread pense avoir mis pour s'exécuter ainsi que la somme de ces valeurs.

Une difficulté est que la valeur de retour de la fonction exécutée par un thread est perdue. Avec une construction de plus haut niveau comme `std::future`, on aurait pu la récupérer, mais c'est hors du périmètre de ce cours.

On va donc transformer la fonction qui renvoie une valeur via « return » en une fonction qui produit la valeur via un argument passé par référence ou par adresse. Attention, chaque fonction doit travailler sur une valeur différente (par exemple chaque fonction sur sa case de tableau), ou alors un mutex est nécessaire.

La commande `time` permet de voir les différents temps utilisés par une commande :

```
$ time ./lancement.exxb 20
Th principal : lancement de 10 fois la fonction
0 lancement de la fonction pour 5 itérations
...
Th principal : Le temps total de calcul est 9.87181

real    0m9.876s
user    0m9.863s
sys     0m0.002s
```

— *real* est le temps écoulé pendant le calcul.

— *user* est la somme des temps processeur utilisés par les threads en mode utilisateur (pour faire les calculs eux-mêmes).

— *sys* est la somme des temps processeur passés en mode noyau (pour faire les opérations système comme les affichages).

Q.I.5) - Faites plusieurs essais avec 2, 3, 4, 8, 10, 20 threads.

5(a) - Pourquoi le temps utilisateur est-il supérieur au temps réel ?

5(b) - Le somme des temps que vous calculez est-elle liée au temps réel ? Au temps utilisateur ? Au temps système ?

5(c) - Est-elle identique, et s'il y a une différence pourquoi ?

Pour la dernière question il faut comparer cela avec le nombre de processeurs/cores de l'ordinateur (`cat /proc/cpuinfo`).

Si votre ordinateur est multiprocesseur, les threads permettent un véritable parallélisme et le temps réel est donc inférieur à la somme des temps processeur nécessaires.

Le temps mesuré en utilisant `gettimeofday()` est le temps écoulé entre 2 moments. Il tient compte des autres threads ou processus qui fonctionnent sur la machine et donc peut être très supérieur au temps de calcul effectué (le temps user). C'est le cas si le thread doit attendre pour utiliser le processeur, donc s'il y a plus de threads qui calculent que de processeurs (n'oubliez pas qu'il y a d'autres programmes qui tournent sur votre ordinateur, notamment pour gérer l'affichage graphique).

Selon le nombre de processeurs de votre machine, vous devriez donc observer que la somme des temps vus par les threads est à peu près celle du temps utilisateur lorsqu'il y a moins de threads que d'unités de calcul. En effet, ces derniers trouvent toujours un processeur disponible pour faire les calculs. Ils n'attendent jamais.

Dès que le nombre de threads est supérieur au nombre de processeurs, le temps mesuré devient plus important que le temps de calcul car il tient compte des moments où les threads sont en attente du processeur (sans parler de la charge de la machine).

Vous pouvez consulter les images obtenues sur un ordinateur avec 8 cœurs de calcul (attention, il n'y a qu'un test à chaque fois, les résultats sont assez imprécis).

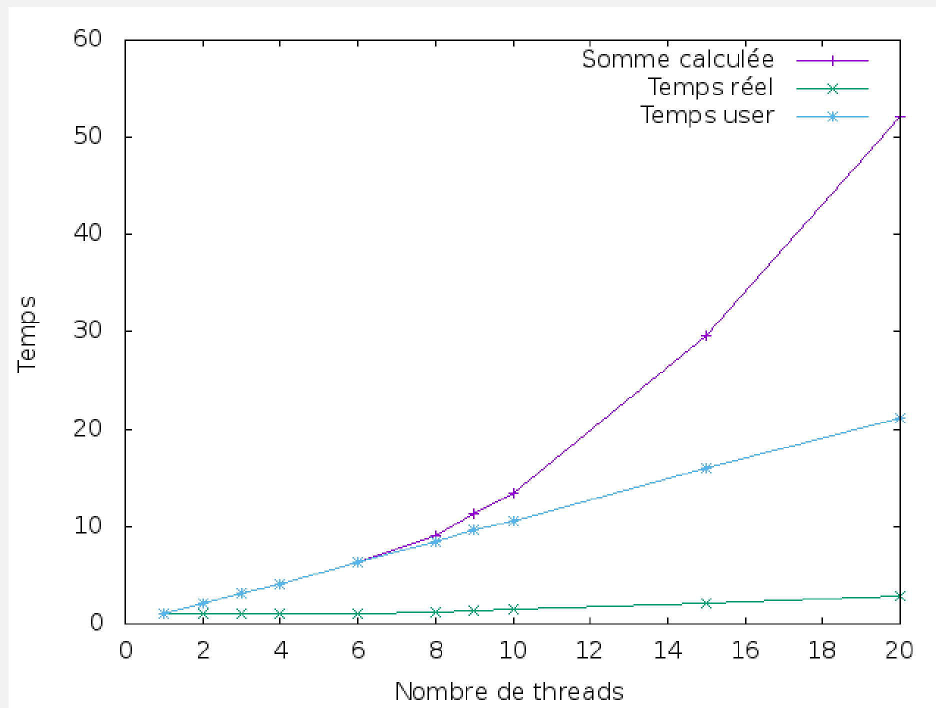

Figure 1 – Temps mesuré en fonction du nombre de threads

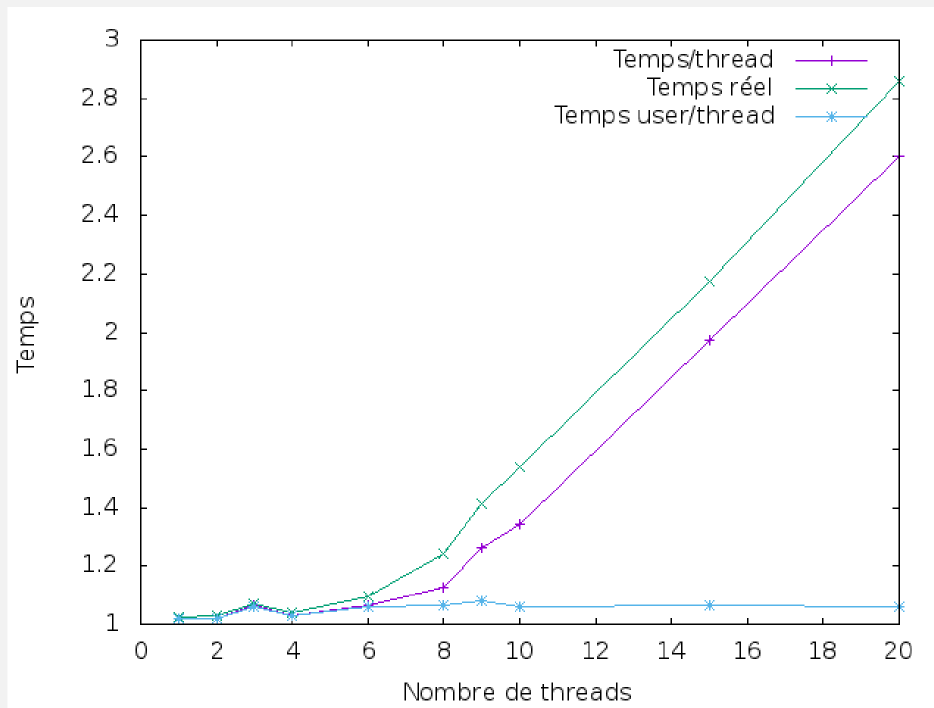

Figure 2 – Rapport du temps et du nombre de threads

Des éléments de corrections seront mis sur la page du cours

II Début du TP2

Vous devriez avoir terminé ce sujet au bout d'1h30 / 2h. Quand cela sera fait, vous devez commencer le sujet de TP2.