TP 6

Victor Boone

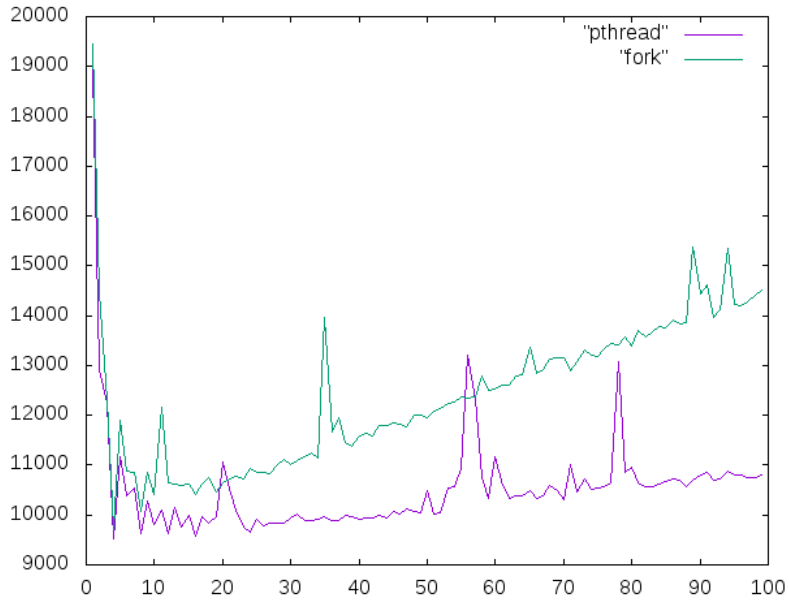
Vous pouvez télécharger le code du TP à cette adresse :
Voici quelques informations utiles sur la machine sur laquelle ont été fait les tests :

```
Intel(R) Core(TM) i3-6006U CPU @ 2.00GHz
CPU(s) : 4
On-line CPU(s) list : 0-3
Thread(s) par cœur : 2
```

Exercice 1 On utilise une représentation linéaire des matrices. Ainsi, dans le code, une matrice sera représentée par un `int*`, et deux `int` représentant sa taille. On peut rajouter un argument aux fonctions afin qu'elles renvoient le temps d'exécution en microsecondes plutôt que la matrice résultat.

Question 1 - 2 - 3 - 4 Cf code.

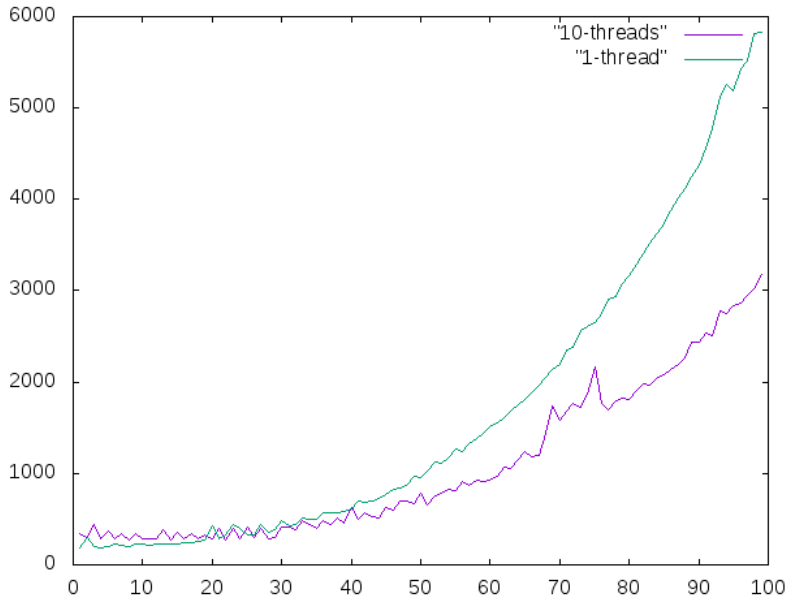
Question 5



Ci-dessus : comparaison des codes `ex1_matmul_fils` et `ex1_matmul_proc`. On compare les performances d'un programme multithreadé utilisant `fork` et un autre utilisant des `pthread` à nombre de threads égaux. Le benchmark a été fait sur des multiplications de matrices aléatoires de taille 150×150 à valeurs dans $[\![-20;\ 20]\!]$. L'échelle de temps est en microsecondes.

On remarque que la version utilisant `pthread` est plus efficace que la version utilisant `fork()`. C'est parce que `fork()` est plus lourd. Autre remarque : on a une accélération linéaire jusqu'à 4 threads. Sur certaines simulations, 8 threads étaient plus performant que 4, car mon processeur possède 2 cœurs gérant naturellement 2 threads chacun (mais accélération sous-linéaire entre 4 et 8). De manière générale, il semble qu'Ubuntu gère mieux un nombre de thread multiple de 4 (mon nombre de cœurs).

Ensuite, on décélère de manière sous-linéaire au-delà d'une dizaine de cœurs, car ceux-ci sont inutilement nombreux et l'OS perd du temps à switcher de thread en thread.



Ci-dessus : comparaisons des codes `ex1_matmul_fils` dans les versions utilisant 1 thread et une autre 10. En ordonnée le temps en microsecondes, et en abscisse la taille des matrices. Les matrices sont à valeurs dans $\mathbb{Z} \cap [-20;\ 20]$. Autrement dit, on compare la méthode naïve à la méthode utilisant plusieurs threads.

Globalement, la version utilisant 10 threads est plus lente au début : car lancer les threads et switcher de thread coûte cher par rapport à la taille de la matrice, c'est quand la matrice grossit que la solution devient intéressante, et est entre 1.5 à 3 fois plus rapide en moyenne, sur de grosses matrices.

Exercice 2

Question 1 Qu'est-ce que j'observe ? Je suis choqué. Dans le code, `res` est une variable globale que tous les threads vont s'amuser à incrémenter en même temps. Il va donc y avoir plein d'accès concurrentiels, car l'opération "`res += a[i]`" n'est pas atomique : voici le bout de code assembleur qui fait cet opération :

```
0x00000000004008f8 <+50>:    mov    0x2007a9(%rip),%rax        # 0x6010a8 <a>
0x00000000004008ff <+57>:    mov    -0x10(%rbp),%rdx
0x0000000000400903 <+61>:    shl    $0x2,%rdx
0x0000000000400907 <+65>:    add    %rdx,%rax
0x000000000040090a <+68>:    mov    (%rax),%eax
0x000000000040090c <+70>:    movslq %eax,%rdx
0x000000000040090f <+73>:    mov    0x20077a(%rip),%rax        # 0x601090 <res>
0x0000000000400916 <+80>:    add    %rdx,%rax
0x0000000000400919 <+83>:    mov    %rax,0x200770(%rip)        # 0x601090 <res>
```

Finalement, voilà ce qu'il se passe si on lance plusieurs fois `sum.c` sur l'entrée `10000 2` :

```
Sum=9511 (in 0.000617 sec)
Sum=7344 (in 0.000558 sec)
Sum=7345 (in 0.000562 sec)
Sum=6864 (in 0.000696 sec)
```

Le programme est sensé renvoyer `10000`. Ici, il a un comportement non-déterministe. Pour résoudre le problème, on peut utiliser un `mutex`, et il n'y aura alors plus d'accès concurrentiel. Cependant, cette méthode est absolument désastreuse car on perd tout l'intérêt de lancer des threads ; mais ça, c'est la question suivante.

La solution corrigée se nomme `fixed-sum.c`.

Question 2 Voici un tableau de comparaison de performances :

`./a.out 10000 2`	`./a.out 10000 1`
`Sum=10000 (in 0.002477 sec)`	`Sum=10000 (in 0.001542 sec)`
`Sum=10000 (in 0.006801 sec)`	`Sum=10000 (in 0.001639 sec)`
`Sum=10000 (in 0.005925 sec)`	`Sum=10000 (in 0.001809 sec)`
`Sum=10000 (in 0.005782 sec)`	`Sum=10000 (in 0.001604 sec)`
`Sum=10000 (in 0.003077 sec)`	`Sum=10000 (in 0.001569 sec)`
`Sum=10000 (in 0.005916 sec)`	`Sum=10000 (in 0.001580 sec)`
`Sum=10000 (in 0.010376 sec)`	`Sum=10000 (in 0.001679 sec)`
`Sum=10000 (in 0.009105 sec)`	`Sum=10000 (in 0.001135 sec)`
`Sum=10000 (in 0.007642 sec)`	`Sum=10000 (in 0.000839 sec)`

Ainsi, le programme est beaucoup plus rapide avec 1 thread qu'avec 2. En fait, les threads passent leur vie bloqués à `lock` / `unlock`.

Remarque : On pourrait aussi critiquer l'utilisation de `clock()`. Il est cependant clair que le threading est très mal géré ici.

Pour améliorer ça, je propose de se débarrasser des `mutex`. Les threads vont faire leur calcul en interne, et renvoyer leur bout de somme en le castant en `void*` [1]. De plus, j'utilise `clock_gettime()` par la suite plutôt que `clock()` qui m'a donné quelques mesures de temps complètement erronées sur l'exercice 1 lorsqu'on augmente grandement le nombre de threads [2].

On obtient alors un programme qui se comporte comme on le souhaite : `improved-sum.c`. Ses performances par rapport au nombre de threads se comportent de la même manière que dans l'exercice 1, donc je n'en parlerai pas.

1. C'est une astuce vue en cours et bien connue
2. c'est-à-dire que `clock()` mesure la consommation du processeur (proche du nombre de tours d'horloge) qui lui, ne peut que grandir avec le nombre de threads, bien que le programme aille plus vite (c'est pour ça que ce dernier n'est pas approprié pour mesurer le temps en MT)