Joris Placette, Clément Girard, Yann Kerkhof

TD8 – AI26 / SR02

Joris Placette, Clément Girard, Yann Kerkhof

Joris Placette, Clément Girard, Yann Kerkhof

Introduction

Ce TD a pour but de nous faire utiliser les thread POSIX en parallélisant une tache grace aux threads POSIX. Dans ce TD nous allons travailler sur une tache exécutant l'algorithme du crible d'Ératosthène. *Le crible d'Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un entier donné* (Wikipédia). L'algorithme est donné dans le cours. Nous allons travailler sur plusieurs taches : Tout d'abord nous allons analyser l'algorithme. Puis dans la tache 2 nous allons implémenter une première version, dite séquentielle de l'algorithme. Dans la tache 3 nous allons alors développer une version dite parallèle dans laquelle nous allons paralléliser les taches. Et enfin, dans l'étape 4 nous comparerons les performances des versions que nous avons rédigés.

Tache 1

Dans cette première tâche nous allons analyser l'algorithme donné dans le TD décrivant le crible d'Ératosthène. 3 questions nous sont alors posées :

1. **Dérouler l'exécution de cet algorithme avec n=20**

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	V	**F**	V	**F**	V	**F**	V	**F**	V	**F**	V	**F**	V	**F**	V	**F**	V	**F**

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	V	**F**	V	**F**	V	**F**	**F**	**F**	V	**F**	V	**F**	**F**	**F**	V	**F**	V	**F**

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	V	**F**	V	**F**	V	**F**	**F**	**F**	V	**F**	V	**F**	**F**	**F**	V	**F**	V	**F**

2. **Pourquoi la boucle de l'intérieure (la deuxième boucle) commence à i_2 et pas 0 ou i ?**

La boucle intérieure ne commence pas à 0 car le premier indice du tableau est 2. On ne commence pas non plus à i car c'est le nombre actuel. Or on veut garder sa valeur à Vrai pour indiquer qu'il est premier.

3. **Pourquoi la première boucle s'exécute jusqu'à √n? Que faites-vous si √n n'est pas un entier ?**

Continuer au-delà de racine(N) est inutile car pour tout nombre non premier a < N, il existe 2 diviseurs k et p de telle sorte que k < racine(N) < p < N. On a donc a = k*p. Or, comme tous les nombres inférieurs à p racine(N) ont déjà été traités par l'algorithme, ainsi, kxp a déjà été éliminé par le crible.

Conclusion : si h n'est pas premier, il possède des diviseurs < Racine(h) et donc dans le cas où n=h, la boucle peut s'arrêter à Racine(n). (on arrondit à l'inférieur).

Tache 2

Nous allons commencer par écrire un premier programme dans le but d'implémenter l'algorithme. Cette première implémentation a pour but de faire fonctionner une version du crible d'Ératosthène sans parallélisation des tache. Nous allons demander a l'utilisateur jusqu'a quel nombre il veut que l'algorithme fonctionne. Nous avons également implémenter une notion de temps d'execution grace à <time.h>.
Fichier : *sequentiel.c*
Commande de compilation : *gcc -o sequentiel sequentiel.c -lm*
Commande d'exécution : *./sequentiel*

Tache 3

Cette tâche est la tache dans laquelle nous allons implémenter la fonctionnalité la plus importante du TD, nous allons paralléliser les taches d'exécution de l'algorithme. Pour cela nous allons demander à l'utilisateur, comme pour la tache précédente, jusqu'à quel nombre il veut que l'algorithme fonctionne. Ensuite nous allons également demander sur combien de threads il souhaite paralléliser la tache. Une fois que ces informations sont renseignés, l'algorithme va créer des thread et leurs assigner une tache de calcul. Les threads sont gérés grâce à <pthread.h>.
Fichier : parallele.*c*
Commande de compilation : *gcc -o parallele parallele.c -lm -lpthread -DLinux -DREENTRANT*
Commande d'exécution : *./parallele*

Tache 4

Dans cette tâche nous allons comparer les performances d'exécution des deux programmes rédigés dans les 2 taches précédentes (sequentielle et parallele avec 1 à 7 threads) pour l'algorithme allant jusqu'à :
500 000, 1 000 000, 2 000 000 et 5 000 000.
Pour pouvoir effectuer cette comparaison nous avons rédigé un dernier programme : compare.c. Ce programme exécute les algorithmes avec les paramètres indiqué et calcul les temps d'exécutions grace à <time.h>.
Fichier : compare.*c*
Commande de compilation : *gcc -o compare compare.c -lm -lpthread -DLinux -DREENTRANT*
Commande d'exécution : *./compare*
En exécutant le programme compare, on obtient les résultats suivants :

Joris Placette, Clément Girard, Yann Kerkhof

```
yann@yann:~/Documents/AI26/TD8$ ./compare
n = 500000 :
Résultats simples :
seq=0.009136,
par1=0.007663,
par2=0.009981,
par3=0.013001,
par4=0.014095,
par5=0.020625,
par6=0.024697,
par7=0.028236
Résultats (moyenne de 10 lancements) :
seq=0.004494,
par1=0.008143,
par2=0.010607,
par3=0.012743,
par4=0.015513,
par5=0.020196,
par6=0.024358,
par7=0.028001


----------------------------------------------------------------------
n = 1000000 :
Résultats simples :
seq=0.008122,
par1=0.015075,
par2=0.020747,
par3=0.025622,
par4=0.027470,
par5=0.037765,
par6=0.037007,
par7=0.040299
Résultats (moyenne de 10 lancements) :
seq=0.008030,
par1=0.013487,
par2=0.021156,
par3=0.024714,
par4=0.028429,
par5=0.036833,
par6=0.040101,
par7=0.041578
```

Joris Placette, Clément Girard, Yann Kerkhof

```
--------------------------------------------------------------------------
n = 2000000 :
Résultats simples :
seq=0.024455,
par1=0.031652,
par2=0.035500,
par3=0.038077,
par4=0.043789,
par5=0.065612,
par6=0.065280,
par7=0.077119
Résultats (moyenne de 10 lancements) :
seq=0.016357,
par1=0.031661,
par2=0.038827,
par3=0.041859,
par4=0.044534,
par5=0.073565,
par6=0.067157,
par7=0.076879


--------------------------------------------------------------------------
n = 4000000 :
Résultats simples :
seq=0.061013,
par1=0.118318,
par2=0.093412,
par3=0.114681,
par4=0.133937,
par5=0.164960,
par6=0.170251,
par7=0.198908
Résultats (moyenne de 10 lancements) :
seq=0.046949,
par1=0.116127,
par2=0.096243,
par3=0.116818,
par4=0.139108,
par5=0.161191,
par6=0.182342,
par7=0.195369
```

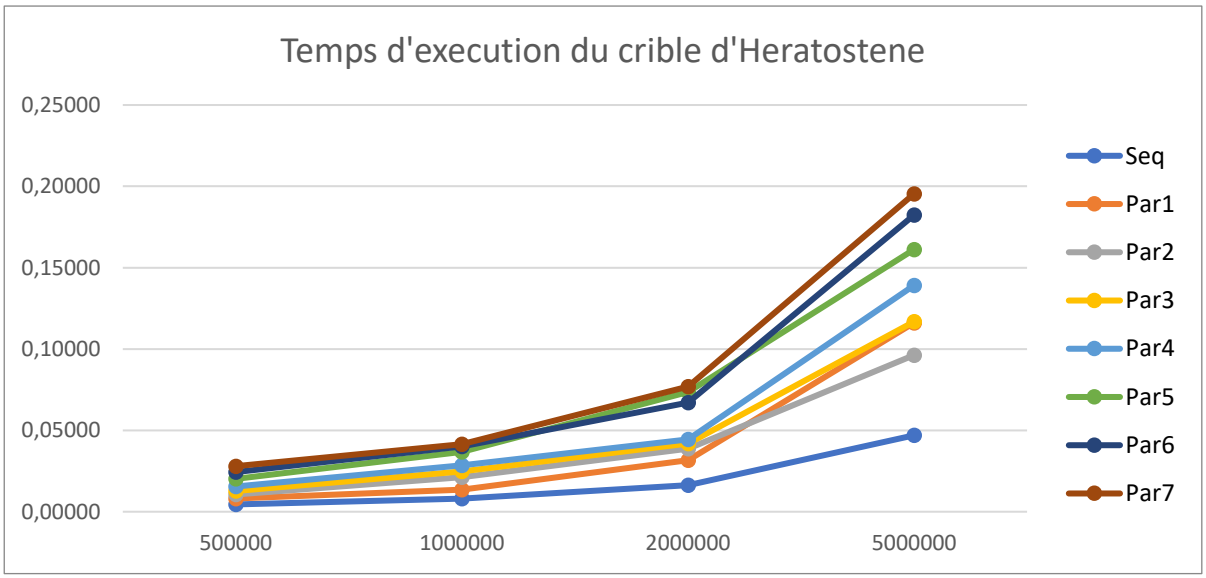
On peut observer les 4 analyses avec le paramètre n, représentant le chiffre jusqu'au quel exécuter l'algorithme pour chacune des analyses. Ensuite les résultats, en temps permettent de comparer les différents temps pour les 2 algorithmes et le nombre de threads engagés.

A chaque fois nous avons exécuté une fois l'algorithme (simple) et puis une moyenne de 10 exécutions (moyenne).

Nous avons résumé les résultats dans le tableau suivant, sur une moyenne de 10 exécutions du programme compare :

N	Seq	Par1	Par2	Par3	Par4	Par5	Par6	Par7
500 000	0,004494	0,008143	0,010607	0,012743	0,015513	0,020196	0,024358	0,028001
1 00000	0,00803	0,013487	0,021156	0,024714	0,028429	0,036833	0,040101	0,041578
2 00000	0,016357	0,031661	0,038827	0,041859	0,044534	0,073565	0,067157	0,076879
5 00000	0,046949	0,116127	0,096243	0,116818	0,139108	0,161191	0,182342	0,195369

On peut représenter ces résultats dans les graphes suivants :



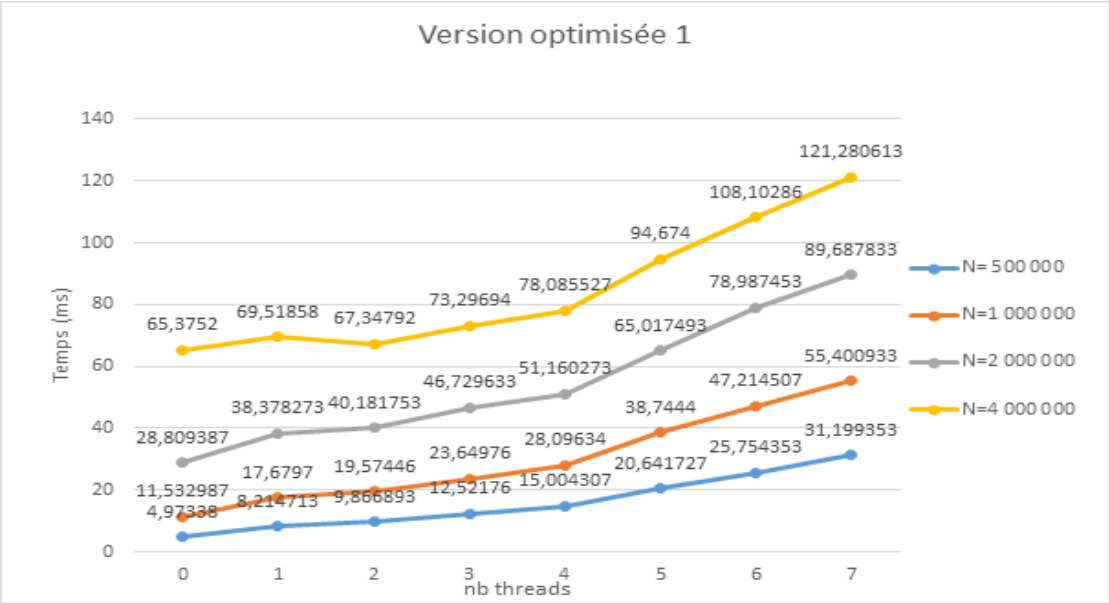
L'analyse de l'exercice se base sur les résultats obtenus ci-dessus. On peut étonnamment voir que le programme qui a un temps d'exécution le plus faible est le programme séquentiel. Et que plus on partitionne l'exécution du programme dans des threads, plus le temps d'exécution est lent. Ce schéma se répète pour les 4 cas étudiés. On peut potentiellement expliquer cela par le fait que la vitesse de commutation de contexte est possiblement trop lente. Aussi, comme on a pu le voir en TD, plus on augmente le nombre de threads, plus le programme peut être lent au début car le système d'exploitation met du temps pour initialiser les threads. Augmenter le nombre de threads peut donc avoir un effet négatif sur le rendement du programme.

Tache 5

Accélération de la boucle interne :

Joris Placette, Clément Girard, Yann Kerkhof

Pour optimiser la vitesse d'exécution, on peut utiliser un pas de 2i car tous les nombres pairs sauf 2 ne sont pas premiers par définition. On peut modifier la boucle pour obtenir de nouvelles valeurs, cependant il est important de noter que les tests on étés fait sur un autre ordinateur :



On remarque que plus les données en entrée sont grandes, plus le gain de temps est important.

Réduction de l'espace mémoire :

Avec cette méthode, on retire tous les nombres pairs du tableau (qui ne sont pas premiers Excepté 2) ainsi que 0 et 1 qui sont premiers. On va effectuer les calculs uniquement pour les entiers impairs supérieurs à 3.

Cela nous fait économiser en espace mémoire de la moitié de la taille du tableau. En termes de temps de calcul, si l'optimisation précédente et appliquée, cela n'aura pas un grand impact.

Conclusion

Dans ce Td nous avons mis en place un algorithme de façon séquentielle puis nous l'avons parallélisé grâce aux threads POSIX. Enfin nous avons pu calculer le temps d'exécution pour chacun des cas pour pouvoir ensuite les analyser.