# Rapport sur l'Optimisation du Crible d'Ératosthène

Léopold Chappuis - Macéo Anger | P24

Université de Technologie de Compiègne

## Introduction

Dans ce travail, nous avons pour objectif d'étudier et d'optimiser l'algorithme du Crible d'Ératosthène. Cet algorithme, utilisé pour trouver tous les nombres premiers jusqu'à un nombre donné $n$, sera d'abord implémenté de manière séquentielle. Ensuite, nous développerons une version parallèle en utilisant les threads POSIX. Enfin, nous comparerons les performances des versions séquentielles et parallèles et analyserons l'impact de différentes optimisations. Les tâches abordées sont déclinées en plusieurs étapes, chacune ayant des objectifs précis pour démontrer l'efficacité des méthodes et des optimisations mises en place.

## Tâche 1

**Question : Dérouler l'exécution de cet algorithme avec $n = 20$**

```
INPUT: n=20
```

Table 1 – Initialisation

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

Table 2 – i=2 : A[2] == VRAI

| | |
|---|---|
| j=4 | Faux |
| j=6 | Faux |
| j=8 | Faux |
| j=10 | Faux |
| j=12 | Faux |
| j=14 | Faux |
| j=16 | Faux |
| j=18 | Faux |
| j=20 | Faux |

TABLE 3 – État du tableau après i=2

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

TABLE 4 – i=3 : A[3] == VRAI

| | |
|---|---|
| j=9 | Faux |
| j=12 | Faux |
| j=15 | Faux |
| j=18 | Faux |

TABLE 5 – État du tableau après i=3

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

```
i=4: A[4] != VRAI => RIEN

i=5: 5 > 20 = 4,47 : FIN

OUTPUT: [2, 3, 5, 7, 11, 13, 17, 19] (Seulement les A[i] VRAI)
```

**Question : Pourquoi la boucle intérieure (la deuxième boucle) commence à $i^2$ et pas à 0 ou $i$ ?** Les multiples inférieurs à $i^2$ sont déjà traités par les précédentes valeurs de $i$.

Admettons que $i = 3$, nous avons donc $i = 2$ déjà parcouru donc 4, 6, 8 sont sur FAUX. Si $j$ commence à 0 : $j = 0$ n'est pas concerné car le premier nombre divisible par 3 est 3, cela revient donc à commencer par $i$. Si $j$ commence à $i$ : $j = 3$ est lui-même donc premier, $j = 6$ est multiple mais est déjà sur FAUX. Ainsi, nous pouvons déjà commencer à $j = i^2$ pour éviter de tester des valeurs déjà traitées.

**Question : Pourquoi la première boucle s'exécute jusqu'à $\sqrt{n}$ ? Que faites-vous si $\sqrt{n}$ n'est pas un entier ?** Soit un nombre $n = a \times b$, alors nécessairement $a \leq b$. On a soit $a$ soit $b$ qui est donc inférieur ou égal à $\sqrt{n}$. Ainsi en testant jusqu'à $\sqrt{n}$, si on trouve un multiple alors le nombre n'est pas premier.

Si $\sqrt{n}$ n'est pas entier, on va jusqu'à la partie entière la plus proche ($i \times i \leq \sqrt{n}$).

# Tâche 2

**Implémentation séquentielle du crible d'Ératosthène**

Comme première étape pour votre version parallèle du programme, implémenter une version séquentielle pour le crible d'Ératosthène.

Fichier : sequentiel.c

sieve_of_eratosthenes(int n) : Prend un entier n en argument, alloue un tableau d'entier de taille n et place l'ensemble des 2 à N case sur VRAI. Applique l'algorithme de Crible d'Ératosthène selon l'exemple du sujet et affiche le tableau des nombres premiers uniquement.

main() : Demande un entier à l'utilisateur appelle la fonction sieve_of_eratosthenes en mesurant le temps d'exécution.

# Tâche 3

**Développement d'une version parallèle**. On départ, on initialise nous structure pour gérer et faire fonctionner nos Pthread avec notre problème actuel. On définit également une fonction qui parcourt un ensemble de valeurs pour chercher les nombres divisibles et changer leur valeur à faux (ThreadFunction). On a également une fonction qui initialise les thread au préalable pour éviter de compromettre la mesure du temps d'exécution de notre algorithme. Enfin notre main demande la borne n pour calculer les nombres premiers ainsi que le nombre de thread en parallèle utilisées et lance la mesure et l'exécution.

# Tâche 4

**Comparaison des performances des versions séquentielles et parallèles** Nous avons mesuré le temps nécessaire pour calculer tous les nombres premiers inférieurs à 500,000 avec différentes configurations de threads.

TABLE 6 – Temps d'exécution moyen (en secondes) sur 100 exécutions

| n | Séquentiel | Paral k=1 | Paral k=2 | Paral k=3 | Paral k=4 | Paral k=5 | Paral k=6 | Paral k=7 |
|---|---|---|---|---|---|---|---|---|
| 500000 | 0.00152966 | 0.00429126 | 0.00608381 | 0.01236699 | 0.01390039 | 0.02606561 | 0.04321577 | 0.05927598 |
| 1000000 | 0.00315051 | 0.00634327 | 0.00885678 | 0.01760949 | 0.01894415 | 0.03539035 | 0.06079985 | 0.07766984 |
| 2000000 | 0.00711209 | 0.00891900 | 0.01309375 | 0.02368481 | 0.02800150 | 0.04744994 | 0.07900796 | 0.09737328 |
| 4000000 | 0.01678269 | 0.02475063 | 0.02350763 | 0.03977924 | 0.04556235 | 0.07297162 | 0.10795205 | 0.14072581 |

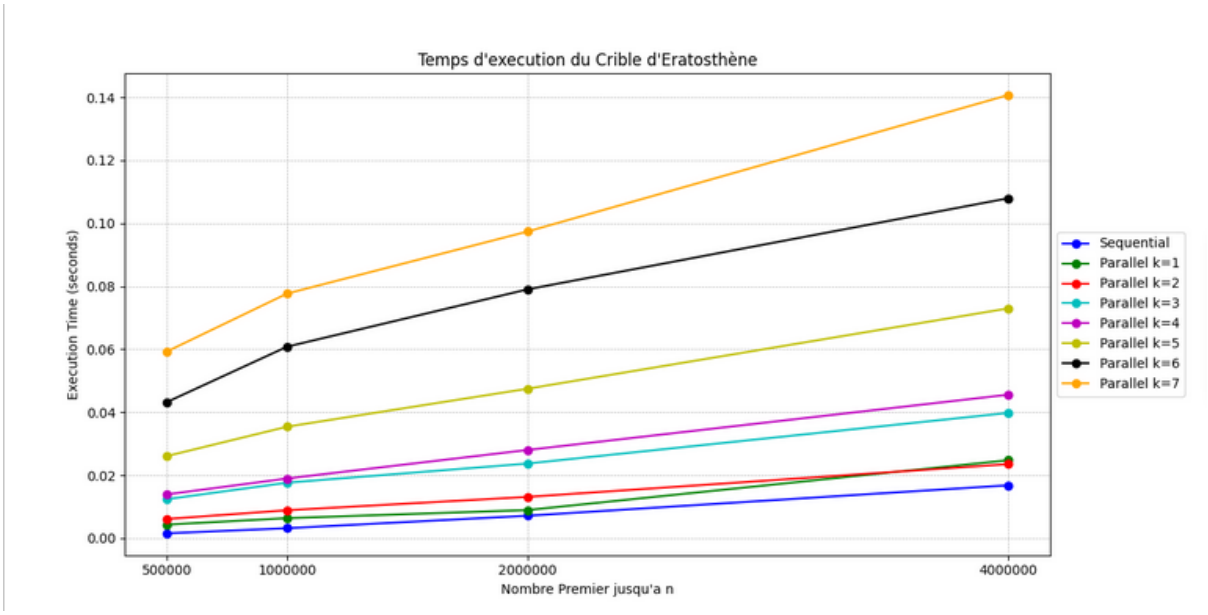


FIGURE 1 – Graphique du temps d'exécution des différentes versions sur 100 exécutions

Observations : La méthode séquentielle est la plus rapide et, dans l'ordre croissant des $k$ de 1 à 7, le multithreading est de moins en moins rapide. On pourrait supposer que

la parallélisation influe sur le temps d'exécution d'un programme donné (à cause de la synchronisation des threads par exemple). Ainsi, plus de threads s'exécutent en parallèle, plus ils mettent de temps à compléter leurs tâches.

# Tâche 5

**Optimisations et comparaison des temps d'exécution**

Nous avons implémenté deux optimisations supplémentaires pour ressortir un tableau de temps moyens sur 100 exécutions.

TABLE 7 – Temps d'exécution moyen après optimisations (en secondes) sur 100 exécutions

| n | Séquentiel | Paral k=1 | Paral k=2 | Paral k=3 | Paral k=4 | Paral k=5 | Paral k=6 | Paral k=7 |
|---|---|---|---|---|---|---|---|---|
| 500000 | 0.00068864 | 0.00329551 | 0.00463381 | 0.00946241 | 0.01038308 | 0.01998259 | 0.03033674 | 0.043456 |
| 1000000 | 0.00140375 | 0.00460076 | 0.00626131 | 0.01385045 | 0.01465861 | 0.02610984 | 0.04662356 | 0.05973589 |
| 2000000 | 0.00294878 | 0.00653874 | 0.00916059 | 0.01938938 | 0.02113912 | 0.03906360 | 0.06582857 | 0.08209812 |
| 4000000 | 0.00604159 | 0.00957254 | 0.01357674 | 0.02594838 | 0.02997924 | 0.05520867 | 0.09262788 | 0.11263526 |

**Différences de temps avant et après optimisations :**

TABLE 8 – Différences de temps avant et après optimisations (en secondes)

| n | Séquentiel | Paral k=1 | Paral k=2 | Paral k=3 | Paral k=4 | Paral k=5 | Paral k=6 | Paral k=7 |
|---|---|---|---|---|---|---|---|---|
| 500000 | -0.00084102 | -0.00099575 | -0.00145 | -0.00290458 | -0.00351731 | -0.00608302 | -0.01287903 | -0.01581998 |
| 1000000 | -0.00174676 | -0.00174251 | -0.00259547 | -0.00375903 | -0.00428554 | -0.00928051 | -0.01417629 | -0.01793395 |
| 2000000 | -0.00416331 | -0.00238026 | -0.00393316 | -0.00429543 | -0.00686239 | -0.00838634 | -0.01317939 | -0.01527517 |
| 4000000 | -0.0107411 | -0.01517809 | -0.00993089 | -0.01383085 | -0.01558311 | -0.01776295 | -0.01532417 | -0.02809055 |

Toutes les différences sont négatives, ce qui signifie que les optimisations ont eu un effet positif sur les temps d'exécution. De plus, on remarque que plus le threading est parallélisé, plus la différence est importante, donc plus l'optimisation a d'impact.

# Conclusion

À travers ce travail, nous avons exploré et optimisé l'algorithme du Crible d'Ératosthène en utilisant des techniques de parallélisation avec les threads POSIX. Nous avons observé que bien que la parallélisation puisse augmenter le temps d'exécution en raison des coûts de synchronisation, des optimisations spécifiques permettent de réduire significativement ces temps. En particulier, les optimisations visant à réduire le nombre de vérifications et à diminuer l'espace mémoire utilisé se sont avérées efficaces. Ce projet nous a permis de mieux comprendre les défis et les avantages de la parallélisation, ainsi que les stratégies pour optimiser les algorithmes de manière efficace.