Rapport TP1 - pthread
GIF-7104
Programmation parallèle et distribuée

par
Valentin Gendre & Adrien Turchini
Equipe 21

I. INTRODUCTION

L'objectif de ce premier TP est de paralléliser à l'aide de la librairie pthread un algorithme qui donne en sortie tous les nombres premiers des intervalles que l'on donne en entrée. Cette liste de nombre premier doit être sans doublon et ordonnée (dans l'ordre croissant). L'algorithme prends en arguments deux éléments :

- Le premier est un nombre entier et il correspond au nombre de thread que l'on utilisera dans l'algorithme.
- Le second est le nom du fichier texte qui contient un liste d'intervalles (chaque ligne contient deux nombres entiers de taille arbitraire : une borne inférieure et une borne supérieure séparées d'un espace).

II. METHODE PROPOSEE

Nous avons tout d'abord préparé les intervalles afin de pouvoir paralléliser le reste du programme et la recherche de nombre premier de manière efficace. Nous retrouvons ce code dans le fichier intervals.cpp. On effectue une lecture du fichier texte contenant les intervalles. On récupère ces derniers dans des tableaux et on stocke dans un tableau tous les nombres compris dans chacun des intervalles. Par la suite, une fois que les candidats possibles à tester sont récupérés, nous effectuons un tri fusion afin de les ordonner correctement. Puis nous supprimons tout simplement les doublons en passant notre liste dans une boucle for. On renvoie alors une struct dataNumbers contenant le tableau de tous les candidats à tester dans les intervalles, sans doublon et ordonnés par ordre croissant ainsi qu'un entier qui représente le nombre de candidats possibles.

Une fois le prétraitement des données terminé, nous passons au but de notre programme, la recherche de nombres premiers. Afin d'optimiser le parallélisme, nous avons créé un tableau de taille n_threads qui contient un tableau des candidats à tester. De ce fait, nous assignons à chaque thread un certain nombre de candidats, chaque thread ayant le même nombre d'entier à tester à plus ou moins 1 près afin que les threads aient une charge de travail similaire. De plus nous assignons des candidats se suivant à nos différents threads et non des plages de valeurs à tester afin qu'une plage qui comporte peu de nombres premiers et qu'une autre qui en comporte beaucoup ne donnent pas une quantité de travail différente à chaque thread et du coup que certains soient actifs quand d'autres non.

Afin de sortir la liste des nombres premiers. Pour chaque candidat, nous assignons une valeur true ou false selon le résultat de l'algorithme utilisé via la fonction mpz_proba_prime de la librairie gmp et et nous affichons les nombres premiers.

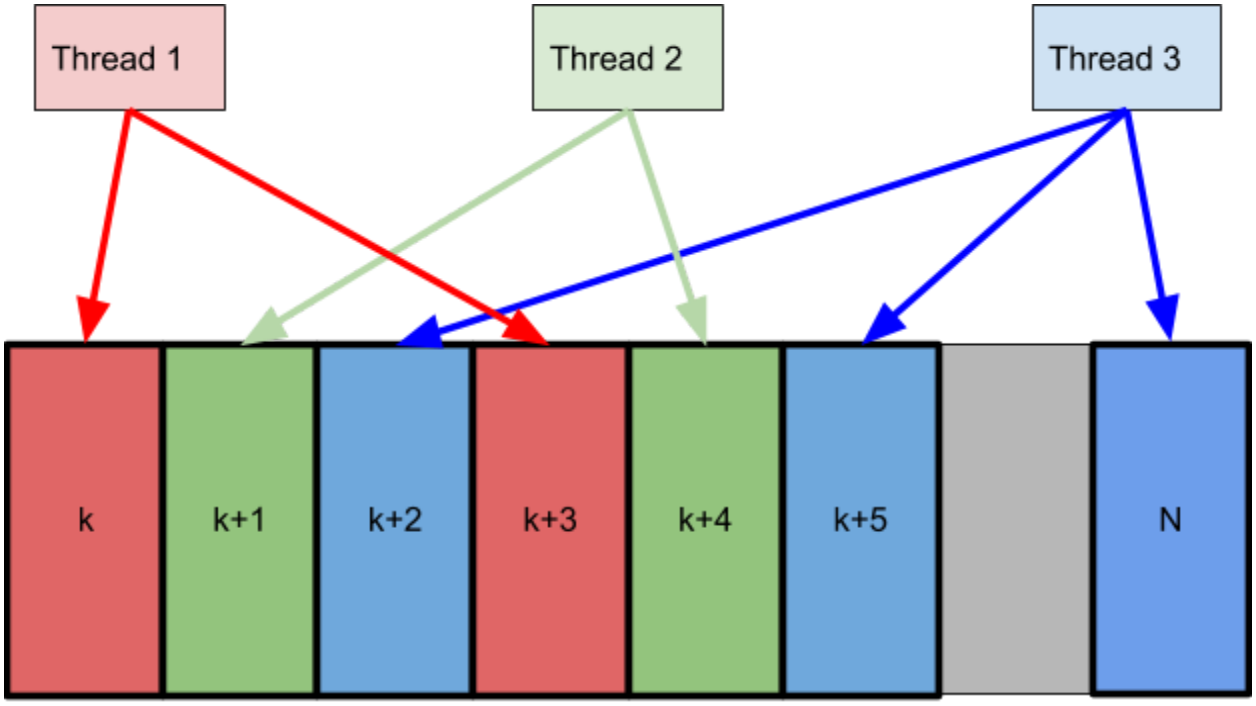
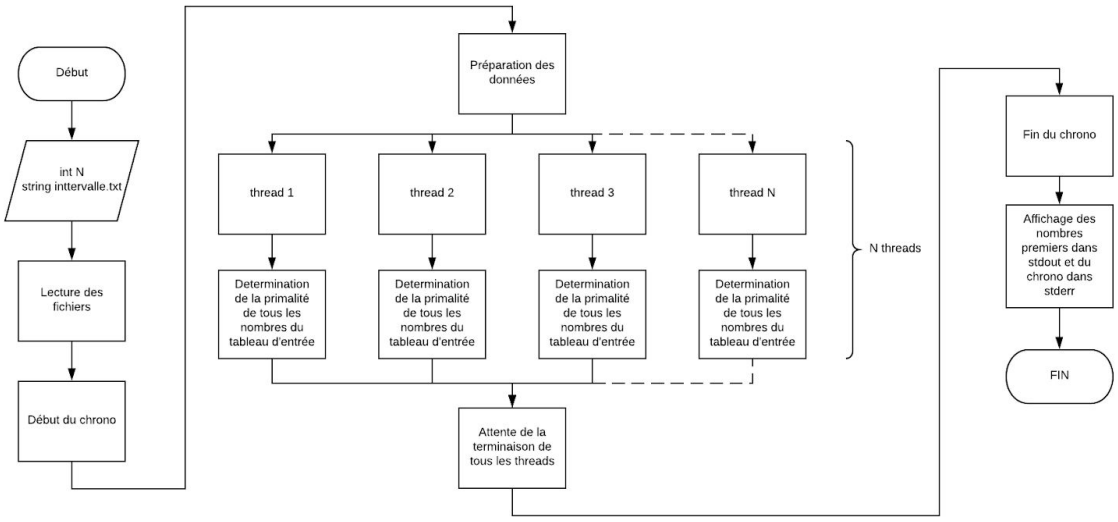


fig : Répartition des nombres à traiter pour chacun des threads

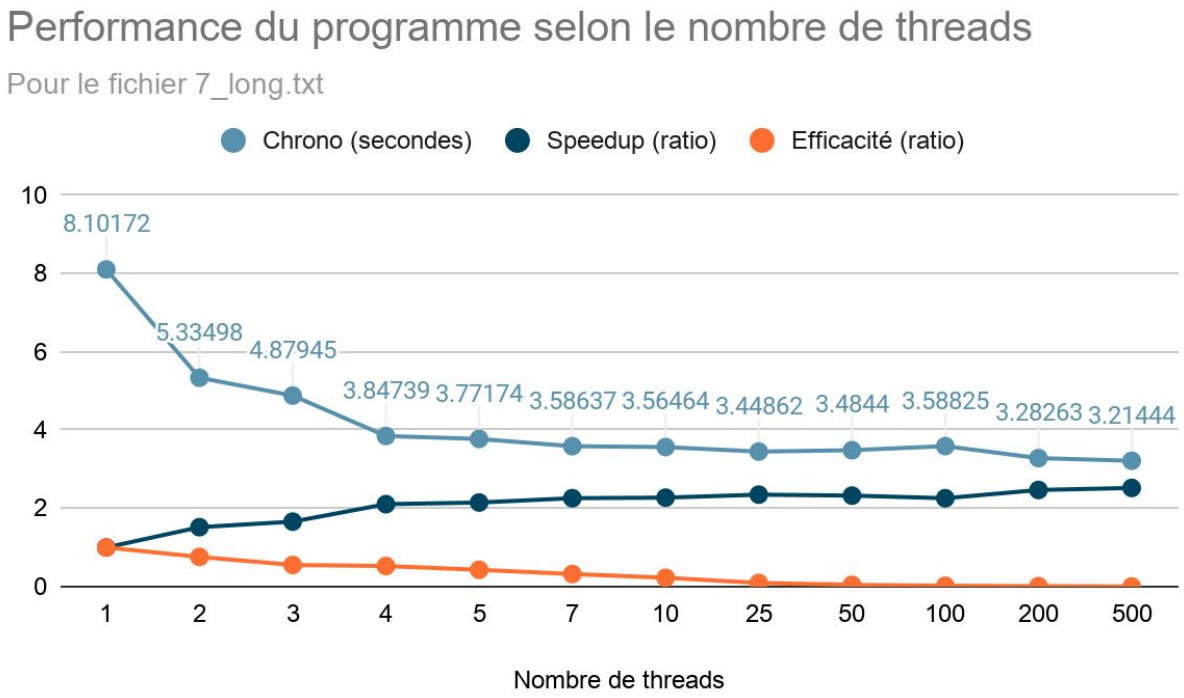


III. RÉSULTATS ET ANALYSE

1. Spécification de la machine

Nous testons notre programme sur un mac sous macOS Big Sur ayant un processeur 2,7 GHz Intel Core i7 quatre cœurs et une mémoire 16 Go 2133 MHz LPDDR3.

2. Résultats



Nous avons choisi de rapporter le speedup et l'efficacité du programme selon le nombres de threads pour le fichier 7_long.txt car ce dernier comprend un grand nombre d'intervalles, que ces derniers sont de grands chiffres, il semblait donc pertinent de tester notre programme avec ce fichier.

Nous remarquons que le chrono diminue fortement lors du passage de 1 thread à 4 threads, principalement lors du passage de 1 thread en multithreading à 2 threads. L'ordinateur utilisé afin de tester le programme comporte 4 cœurs, il est donc normal mais intéressant de voir l'amélioration en termes de vitesse du programme lors de l'utilisation d'un à 4 cœurs de l'ordinateur. En effet nous avons un speedup d'environ 1,5 avec 2 threads, 1,66 avec 3 threads

et on arrive à un speedup de plus de 2 avec 4 threads. On remarque que lorsqu'on augmente le nombre de threads utilisés au-delà de 4, le speedup continue à augmenter mais de manière très légère, cela est due à l'utilisation de l'hyperthreading de l'ordinateur. Cependant l'amélioration est donc négligeable par thread, et avec 500 threads par exemple, le speedup est d'environ 2,5, alors qu'il est déjà de 2 en utilisant seulement 4 threads.

3. Discussion

a. Parallélisation des fils

Pour démarrer les calculs, les fils d'exécutions parallèles nécessitent le résultat du fil d'exécution principal pour obtenir les données d'entrée (tri des données, élimination des redondances). Ces fils d'exécutions sont complètement indépendants entre eux.

b. Utilisation du mutex

Chacun des fils d'exécution travaille dans une partie de la mémoire qui leur est exclusive et réservée. En effet, chaque processus possède son tableau de nombre à tester et son tableau de booléen à remplir. Nous n'avons donc pas besoin d'utiliser un mutex.

c. Options d'optimisation

Nous n'avons ni tenu compte des contraintes de mémoire cache, ni des options d'optimisation du compilateur. Nous avons fait le Makefile "à la main".

d. Améliorations possibles

En plus de se pencher sur les contraintes de mémoire cache et d'option d'optimisation du compilateur, nous pourrions essayer de davantage paralléliser notre algorithme. Une partie de notre algorithme qui n'est pas parallèle est le tri et la fusion des données à traiter. Il est possible de paralléliser le tri fusion par exemple :

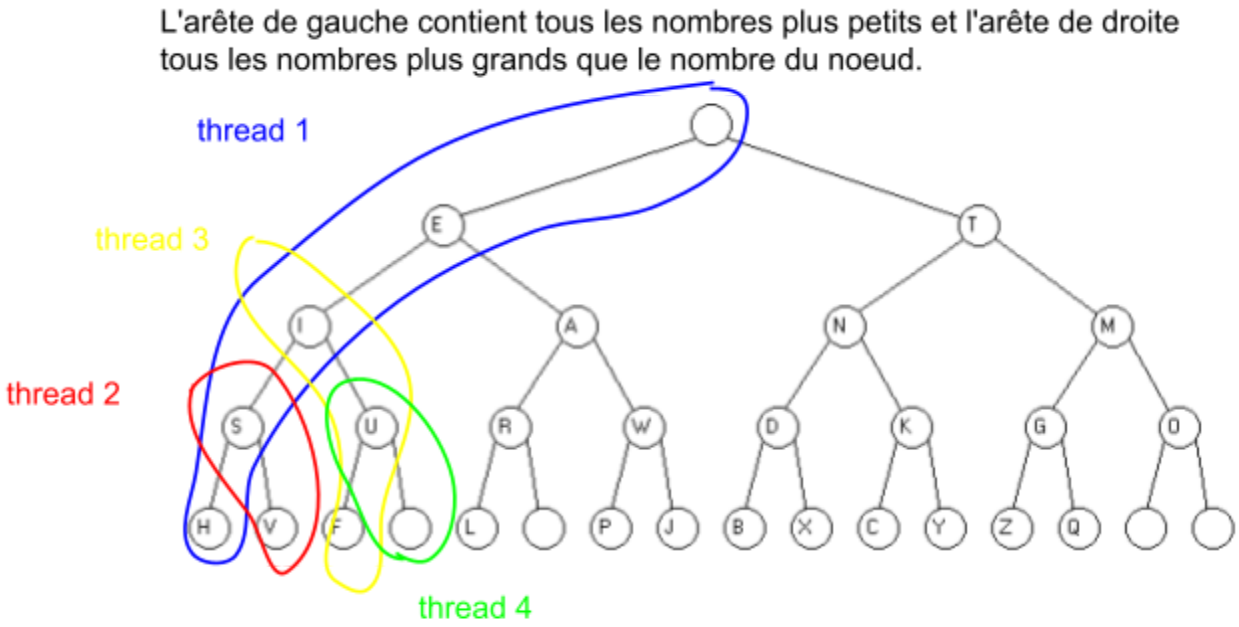


fig : Exemple de traitement parallèle d'un tri fusion à 4 thread

Lors de la reconstruction du tableau chaque thread devra attendre le résultat de thread du dessous (donc il y a dépendance entre plusieurs fils d'exécution) ainsi le gain en temps serait assez faible. En plus des difficultés d'implémentation que représente cette parallélisation, nous ne jugions pas pertinent d'effectuer cette tâche.