TP 2 : OpenMP
GIF 7104 - Programmation Parallèle et distribuée

Valentin Gendre et Adrien Turchini
Equipe 21

February 2021

1 Introduction

Dans ce TP, nous cherchons à mettre en place un programme capable de trouver la liste de nombres premiers situés dans une liste d'intervalle qui nous est fournie par fichier .txt. La liste des nombres premiers situés dans ces intervalles doit être sans doublon et triée par ordre croissant. Le programme doit également être capable de traiter des nombres très grands de l'ordre de 1e100.

Le programme compile grâce à un makefile réalisé à la main par les commande :
$ make clean main
$ ./main

Notre programme prend 2 arguments, le nombre de threads a utiliser lors du traitement des nombres premiers et le chemin du fichier texte qui contient la liste des intervalles à traiter.

2 Méthode proposée

2.1 Pré-traitement des données

Nous nous sommes tout d'abord occupé de traiter les intervalles afin de pouvoir paralléliser la recherche des nombres permiers de la façon la plus efficace possible. On retrouve la partie du programme s'occupant du pré-traitement dans le fichier intervalles.cpp. Nous effectuons tout d'abord la lecture du fichier texte contenant les intervalles à traiter. Par la suite nous récupèrons tous les intervalles à traiter dans un tableau et nous stockons dans un aute tableau tous les nombres compris dans ces intervalles. Nous nous retrouvons donc avec une liste de potentiels candidats.

Par la suite nous effectuons un tri fusion donc de complexité O(n log(n)) en temps et 0(n) en espace de façon séquentielle. Une fois la liste de candidats triée par ordre croissant nous éliminons les doublons en passant cette liste dans une boucle afin de vérifier que le nombre suivant n'est pas le même nombre que celui qui est traité actuellement et nous l'éliminons si c'est le cas. Nous renvoyons donc de notre fonction de pré-traitement une struct dataNumbers contenant un tableau des candidats à tester dans les intervalles, sans doublons et en ordre croissant ainsi qu'un entier qui représente le nombre de candidats.

2.2 Librairie GMP pour les très grands nombres

Notre programme est capable de traiter de très grands nombres grâce à l'utilisation de la librairie gmp. Lors de la lecture de nos intervalles, ces derniers sont lus en tant que string. Par la suite les intervalles sont convertis en tant que mpz_class pour être stockés dans un vecteur de mpz_class. Les intervalles pouvant ne pas être pris en charge sinon car certains dépassant la taille maximale qu'un unsigned long long puisse prendre. Les candidats lors du pré-traitement sont également sous la forme d'un mpz_class de la librairie gmp.

2.3 Recherhe des nombres premiers via OpenMP

Une fois le pré-traitement séquentiel terminé notre recherche de nombre premiers se fait de façon parrallèle. Pour oprimiser notre programme, ce dernier étant parralèle il devait répondre à certaines contraintes supplémentaires qu'un programme séquentiel, nottament en terme d'accès mémoire ainsi que de la bonne répartition de la charge de travail entre les threads. Afin d'avoir une bonne répartition de la charge de travail nous avons créé un tableau de taille n_threads qui contient lui-même un tableau de candidats à tester. Nous assignons par la suite n_candidats/n_threads candidats à chaque threads, chaque threads se retrouvant donc avec le même nombre de candidats à tester à +/- 1 candidat près. Ils auront donc tous une charge de travail similaire et nous maximisons utilisation du multifilaire dans notre programme. De plus on sait que certains intervalles comprennent plus de nombres premiers que d'autres et comme il est plus rapide de calculer si un nombre n'est pas premier que de vérifier qu'il l'est nous avons donné à chaque thread des candidats qui se suivent pour que tous les threads soient actifs en même temps et que le travail soit bien reparti.

Afin de trouver la liste des nombres premiers nous avons une liste de booléens de la même taille que la liste des candidats de chaque thread. Pour chaque candidat nous testons si ce dernier est permier ou pas. Nous réalisons ceci directement via la librairie gmp et la fonction mpz_proba_prime. Si le candidat est premier nous assignons la valeur true à l'index équivalent de la liste de booléens. Notre méthode a été pensée de manière à minimiser l'usage du mutex. En effet comme chaque thread à son propre tableau de candidats et de booléens il n'est

pas nécéssaire de faire l'usage du mutex car la mémoire n'est pas partagée entre les différents threads. De plus comme les candidats sont triée par ordre croissant et les threads différents ont des candidats qui se suivent, nous récupérons la liste des nombres permiers déjà triée sans devoir faire appel au mutex ou sans devoir réaliser un deuxième tri. Nous affichons par la suite la liste des nombres premiers trouvés.

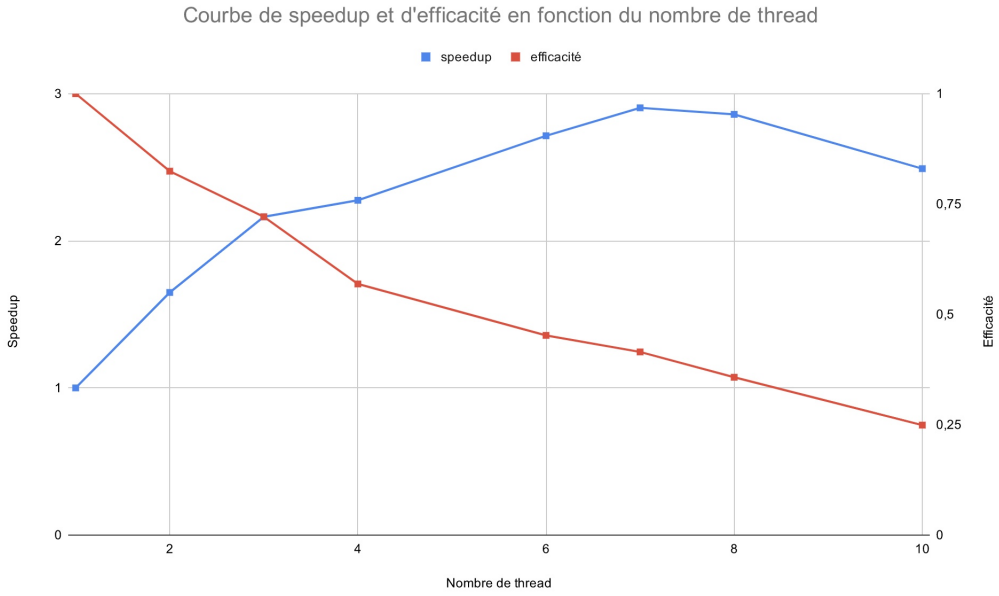
3 Résultats et Analyse

3.1 Spécifications de la machine

Afin de tester notre programme, nous utilisons une machine virtuelle Ubuntu 64 bits avec un processeur Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz de 4 coeurs et 9 threads et 8Go de RAM 2667MHz.

Malheureusement nous avons du changer de machine entre le TP 1 sur pthread et le TP 2 sur OpenMP, nous avons donc refait des tests avec pthreads sur cette nouvelle machine pour pouvoir comparer nos résultats.

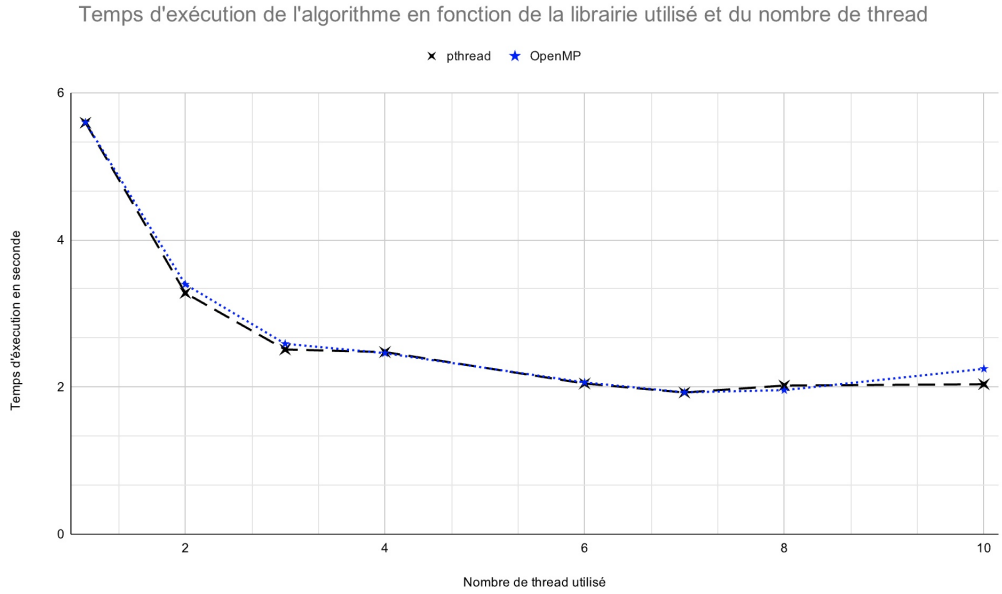
3.2 Résultats



Nous avons rapporter ci-dessus la courbe d'efficacité et de speedup pour le fichier 7_long.txt, ce dernier comportant 983 intervalles différents de très grands nombres. Le challenge étant de pouvoir traiter tous types d'intervalles de façon

parrallèle, ce fichier est alors parfait pour faire nos mesures.

On peut voir que notre speedup augmente fortement du passage de 1 à 7 threads, puis stagne en utlisant 8 threads et diminue légrèrement par la suite. Cela est explicable car l'ordinateur à 4 coeurs et 8 threads. Nous observons donc une grande amélioration lorsque le nombre de threads augmente dans cet intervalle mais après ce n'est plus le cas. Il serait toujours possible d'améliorer le speedup au-délà grâce à l'hyperthreading par exemple mais cela reste négligeable comparé à l'utilisation d'un coeur en plus.



Ci-dessous on peut retrouver le temps d'execution du programme, on peut voir une bonne diminution de 1 à 3 threads suivi d'une baisse moins important jusqu'à l'utlisation de 7 threads, le 8e étant un peu plus lent, cela étant peut-être du aux performances de l'ordinateur au moment ou nous avons réalisé les tests. Après 8 threads notre temps d'execution stagne.

Nous pouvons donc affirmer que la parralèlisme marche bien dans notre programme, nous avons un speedup qui atteint presque 3 et donc un temps d'ecextution d'environ 2 secondes en utlisant 8 threads contre preque 6 secondes avec un seul thread ce qui nous sert de référence pour le speedup.

Nous avons également affiché la courbe de temps execution avec pthread sur le même jeu de donnée. Nous retrouvons des résultats vraiment très similaires de ceux obtenus avec OpenMP, pthread étant un peu plus rapide mais OpenMP reste très optimisé pour autant et marchant presque tout aussi bien pour peut-être un peu plus de simplicité.

3.3 Améliorations

Nous pourrions améliorer notre programme en parallélisant le tri réalisé. Cependant cela semble compliqué pour le tri fusion qui marche bien de façon séquentielle et qui reste d'une complexité acceptable. Il serait interessant de mettre en place un autre type de tri parallèle ayant une complexité temporelle et en espace égale ou moins importante que le tri fusion.