Rapport Tp n°2 - OpenMp
GIF-7104 programmation parallèle et distribuée

Equipe 9

Sébastien Marsaa-poey

Pierre Snell

24 février 2019

Table des matières

Table des figures

Liste des tableaux

List of Algorithms

1 Introduction

L'objectif de ce TP est de réaliser un programme multifilaire qui permet de trouver tous les nombres premiers compris dans une série d'intervalles. Le programme est écrit en C++, et la parallélisation repose sur la librairie OpenMP.
Le programme doit gérer des intervalles de précision arbitraire, (réalisé grâce à GMP) qui peuvent se recouvrir. On souhaite également que la sortie, envoyée dans stdout, soit constituée des nombres premiers trouvés dans l'ordre croissant.

2 Méthodes

Pour ce deuxième Tp, nous avons voulu comparer deux méthodes de parallélisation.
Nous sommes reparti de la base du code du premier Tp, ainsi, une approche simple à été envisagée.
D'autres approches (Producteur consommateurs, utilisation des tâches d'OpenMP) auraient pu être faites mais étaient plus difficilement réalisable après notre premier travail.

Pour les sections suivantes, trois méthodes de répartition des intervalles sont envisagées :

— Méthode de répartition "static"
Cette méthode coupe en Nombre_Thread partie égale l'intervalle et chaque partie est roulée par un thread indépendant.

— Méthode de répartition "dynamic"
Cette méthode coupe en éléments de 1 l'intervalle. Chaque petit élément est ensuite calculé par un thread et dès qu'il à fini openMP lui donne le suivant.

— Méthode de répartition "guided"
Cette méthode commence comme en statique avec un groupe en Taille / Nombre_Thread puis pour chaque groupe la taille est recalculée avec Taille_Restante / Nombre_Thread ce qui amène à avoir des blocs de plus en plus petits

2.a Méthode en parallélisant les intervalles eux mêmes (Intra Intervalles)

Pour cette méthode, nous prenons les intervalles du fichier un par un et divisons l'unique intervalle en différentes parties qui sont ensuite assignées à chaque threads.
En faisant cela, la charge de travail devrait être plus répartie parmi les threads.
En effet, c'est le calcul de probabilités qu'un nombre soit premier qui demande beaucoup de ressources : on réparti donc mieux la charge directement à sa base.

2.b Méthode en parallélisant les groupes d'intervalles (Extra Intervalles)

Pour cette méthode, nous divisons l'ensemble des intervalles soumis dans un fichier en sous intervalles qui seront ensuite traité par différents threads.
En faisant cela nous prenons le risque, en créant de plus gros bloc, que la charge de travail soit moins répartie parmi les différents threads.
En effet, chaque bloc va s'exécuter en parallèle mais le traitement lourd de la vérification de la primauté d'un nombre va se faire de façon séquentielle au sein du thread.
En jouant sur la taille de division de l'intervalle on peut espérer tout de même arriver à de bonnes performances.

La compilation a été optimisée avec l'option -O3.

3 Algorithmique

Algorithm 1 Version Extra Intervalle

Entrées : Fichier d'intervalles, Nombre de threads
Sorties : Affiche les nombre premier des intervalles du fichier

$\mathcal{N} \leftarrow$ Nombre de thread
Assigner $\mathcal{N}$ au nombre de threads OpenMP
Vérifie le fichier d'intervalles
while (Lis ligne) **do**
    insère l'intervalle dans $\mathcal{I}$
debut $\leftarrow$ démarre chrono
swap_intervalle($\mathcal{I}$) ▷ Si inter_bas $<$ inter_haut $\rightarrow$ swap
sort_and_prune($\mathcal{I}$) ▷ Cf algo : 2
alloue les structures custom
for sous_intervalle dans $\mathcal{I}$ **do** ▷ OMP for variables privées : nb
    for nb dans sous_intervalle **do** ▷ for normal sur un mpz_t
        if mpz_proba_prime_p(nb) egal 1 ou 2 **then** ▷ Algorithme de miller-rabin
            Ajoute nb à $\mathcal{L}$ ▷ Omp Critical
$fin \leftarrow$ arrête chrono
Trie $\mathcal{L}$
Affiche $\mathcal{L}$.
Affiche $fin - debut$

Algorithm 2 Sort and Prune

for i<size intervalle **do**
    if Intervalle_bas(i+1) <= intervalle_haut(i) **then**
        if Intervalle_haut(i+1) > intervalle_haut(i) **then**
            intervalle_haut(i) $\rightarrow$ Intervalle_bas(i+1)-1
        else
            erase intervalle

Algorithm 3 Version Intra Intervalle

Entrées : Fichier d'intervalles, Nombre de threads
Sorties : Affiche les nombre premier des intervalles du fichier

$\mathcal{N} \leftarrow$ Nombre de thread
Assigner $\mathcal{N}$ au nombre de threads OpenMP
Vérifie le fichier d'intervalles
while (Lis ligne) **do**
    insère l'intervalle dans $\mathcal{I}$
debut $\leftarrow$ démarre chrono
swap_intervalle($\mathcal{I}$) ▷ Si inter_bas $<$ inter_haut $\rightarrow$ swap
sort_and_prune($\mathcal{I}$) ▷ Cf algo : 2
alloue les structures custom
for sous_intervalle dans $\mathcal{I}$ **do** ▷ for normal mais OpenMP veut un int pas un MPZ_t
    offset $\leftarrow$borne_inf - borne_sup ▷ Pour cast dans un unsigned long
    for nb dans sous_intervalle **do** ▷ OMP for, privé : nb
        nb $\leftarrow$ nb + offset
        if mpz_proba_prime_p(nb) egal 1 ou 2 **then** ▷ Algorithme de miller-rabin (cf lien sous section 3)
            Ajoute nb à $\mathcal{L}$ ▷ Omp Critical
$fin \leftarrow$ arrête chrono
Trie $\mathcal{L}$
Affiche $\mathcal{L}$.
Affiche $fin - debut$

4 Résultats expérimentaux

OS	Linux mint 19.1	Archlinux (Noyeau linux 4.20.11)
Architecture :	x86_64	
Processeur(s) :	8	
Thread(s) par cœur :	2	
Cœur(s) par socket :	4	
Socket(s) :	1	
Nom de modèle :	Intel i7-7700HQ 2.80GHz	Intel i7-4700HQ 2.4GHz
Vitesse maximale du processeur en MHz :	2800	3400
Vitesse minimale du processeur en MHz :	800	
Cache L1d L1i L2 L3 :	32K 32K 256K 6144K	
Ram :	16Go DDR4 2400MHz	

TABLE 1 – Spécifications des machines utilisée pour les tests

Les programmes ont été testés sur une liste de 100 intervalles de longueur 10^5 et de précision arbitraire (10 à 100 digits). Les programmes parallèles utilisent un nombre de threads allant de 1 à 16.

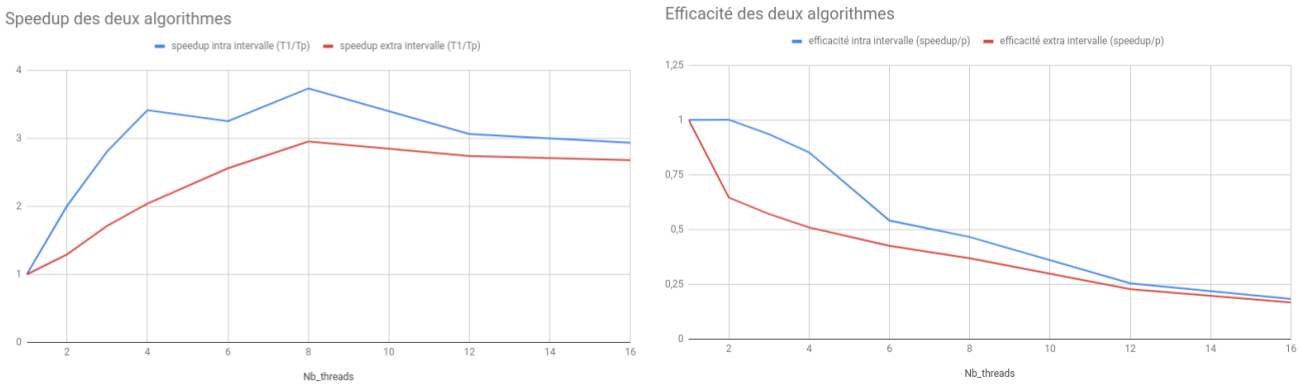

FIGURE 1 – Speedup et efficacité des deux algorithmes
Parallélisation dans l'intervalle Parallélisation par partage des intervalles

5 Discussion

5.a Analyse des résultats

Pour les deux algorithmes, le temps d'exécution minimal est atteint pour 8 threads. Cela correspond au nombre de coeurs de la machine de tests. En effet, à un thread par coeur, on atteint la performance maximale, car toutes les ressources de calculs sont utilisées au maximum et ne sont pas saturées.
Le speedup est meilleur pour la méthode intra intervalle, qui garde une bonne efficacité (>0.8) jusqu'à 4 threads. La methode extra intervalle a quant à elle une efficacité médiocre.
Cela s'explique par le fait que dans le cas extra-intervalle, la charge de travail n'est pas toujours bien répartie entre les threads. Il arrive régulièrement qu'un thread ait fini les calculs sur ses intervalles alors que les autres threads travaillent encore sur leur dernier intervalle.
La modification du schedule pour la version intra intervalle, comme présentée dans la méthode (static, dynamic et guided) n'a pas donné de modification sensible dans les résultats. C'est peut-être parce que la tache partagée est la détermination du caractère premier d'un nombre ; cette tache a toujours la même durée, donc la répartition des

taches ne change pas le temps d'exécution.
Chacune des deux méthode n'utilise qu'un seul mutex, pour écrire dans la liste des résultats. L'écriture est courte devant le temps nécessaire pour déterminer si un nombre est premier, donc peu de temps est perdu à attendre après le mutex.
La version extra-intervalle est encore meilleure sur ce point puisqu'elle n'attend après le mutex uniquement après qu'un intervalle entier ait été traité.
Hormis ce mutex, les threads sont complètement indépendants.

Aussi, les intervalles étant stockés dans un tableau, l'accès aux variables proches est rapide, chaque thread invalide le moins possible la cache des autres (Car le C++ alloue les valeurs des tableaux côte à côte).

5.b Améliorations possibles

La principale source d'amélioration possible serait de diminuer la partie séquentielle de notre code. En particulier, il aurait été possible de trouver une méthode pour paralléliser le tri de nos valeurs à la fin du programme. Cela aurait amélioré le speedup et l'efficacité de la méthode. Aussi, grâce à pthread nous pouvions joindre les thread dans l'ordre voulu et ainsi conserver les nombres dans l'ordre sans avoir reccours à un mutex. Ici, nous n'avons pas accès à la logique d'OpenMP et devons retrier l'intervalle en fin de programme.

5.c Difficultés

La parallélisation avec OpenMP simplifie l'écriture du code par rapport à la version bas niveau de pthread. Cependant, OpenMP ne gère efficacement que les boucles for dont l'itérateur est un entier. Pour itérer sur des objets de type mpz, il a fallu créer un itérateur de type int incrémenté dans une boucle for, et l'ajouter à un offset de type mpz.