Rapport Tp n°1 - Pthread
GIF-7104 programmation parallèle et distribuée

Equipe 9

Sébastien Marsaa-poey

Pierre Snell

11 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 Pthread.
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 Méthode

Nous avons réalisé deux versions de notre implémentation, une avec un mutex et l'autre sans.

2.a Version avec mutex

Pour les détails de l'algorithme, cf : algorithme 3
Dans un premier temps, le fichier qui contient les intervalles est lu. Les intervalles sont stockés dans une liste de structures globale, interval_t, contenant les deux bornes de chaque intervalle.
Un pré-traitement est appliqué à la liste d'intervalles. Si les bornes sont inversées, elles sont remises dans l'ordre (fonction swap_intervalle). Puis les intervalles sont triés dans l'ordre croissant des bornes. Enfin, les recouvrements entre intervalles sont supprimés en comparant deux à deux les bornes d'intervalles successifs (fonction sort_and_prune cf algo :2).
On créée ensuite les threads. Ils prennent chacun en argument un pointeur sur une structure qui leur est propre, dans laquelle ils vont écrire la liste des nombres premiers qu'ils trouveront.
Les threads récupèrent un intervalle à traiter dans la liste globale d'intervalles gIntervalles, tant que tous les intervalles n'ont pas été récupérés. Pour éviter que deux threads distincts n'accèdent au même intervalle, la liste est protégée par un mutex. Puis les threads apppliquent la fonction mpz_probab_prime_p à chaque élément de l'intervalle. Si l'élément testé est premier, il est ajouté à la liste de nombres premiers du thread.

Le main attend la fin de tous les threads avec un appel à la fonction blocante pthread_join pour chacun d'eux. Puis il recupere la liste des nombres premiers de chaque threads dans leur structure et les concatène dans la list finalList. Il ne reste qu'à trier cette liste et à l'afficher dans stdout.

2.b Version sans mutex

Pour la version sans mutex : cf algorithme 1
On lit le fichier contenant les intervalles et on les stocke dans une variable locale.
On trie ensuite les intervalles afin de les avoir par ordre croissant et sans recoupement. (cf algo 2 : sort_and_prune)
On assigne ensuite à chaque thread, de façon uniforme une partie de la liste d'intervalle par ordre de création.
Le thread 1 a donc la première partie le thread 2 la deuxième etc...
Chacun calcule donc une partie de l'intervalle total.
On joint ensuite tout les résultats des threads. Puisque on les joint dans le même ordre que leur création, les résultats sont ordonnés.
Pour terminer on affiche tous les nombres premiers.

3 Algorithmique

cf : doc gmp fonction miller rabin

Algorithm 1 Version sans mutex

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

$\mathcal{N} \leftarrow$ Nombre de thread
Vérifie le fichier d'intervalles
while (Lis ligne) **do**
    insère l'intervalle dans $\mathcal{V}$
debut $\leftarrow$ démarre chrono
swap_intervalle($\mathcal{V}$) ▷ Si inter_bas $<$ inter_haut $\rightarrow$ swap
sort_and_prune($\mathcal{V}$) ▷ Cf algo : 2
for $i < \mathcal{N}$ **do**
    $\mathcal{I} \leftarrow size(\mathcal{V})/\mathcal{N}$
    $\mathcal{F} \leftarrow i * \mathcal{I}$
    $\mathcal{L} \leftarrow ((i+1) * \mathcal{I}) - 1$
    $\mathcal{W} \leftarrow \mathcal{V}(\mathcal{F}, \mathcal{L})$
    Lance le thread avec la fonction **compute_intervalles** et le paquet d'intervalle $\mathcal{W}$ ▷ (cf algo : 5)
for $i < \mathcal{N}$ **do**
    Joins les threads
    Insère sortie du thread dans la liste nombre premier.
debut $\leftarrow$ arrête chrono
Affiche liste nombre premier.
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) → Intervalle_bas(i+1)-1
        else
            erase intervalle
```

OS	Linux mint 19.1
Architecture :	x86_64
Processeur(s) :	8
Thread(s) par cœur :	2
Cœur(s) par socket :	4
Socket(s) :	1
Nom de modèle :	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Vitesse du processeur en MHz :	1000.091
Vitesse maximale du processeur en MHz :	2800,0000
Vitesse minimale du processeur en MHz :	800,0000
Cache L1d L1i L2 L3 :	32K 32K 256K 6144K
Ram :	16Go DDR4 2400MHz

TABLE 1 – Spécifications de la machine utilisée pour les tests

Algorithm 3 Version avec mutex

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

$\mathcal{N} \leftarrow$ Nombre de thread
Vérifie le fichier d'intervalles
while (Lis ligne) **do**
    insère l'intervalle dans $g\mathcal{V}$
debut $\leftarrow$ démarre chrono
swap_intervalle($g\mathcal{V}$)
sort_and_prune($g\mathcal{V}$)
for $i < \mathcal{N}$ **do**
    Lance le thread avec la fonction **compute_intervalles_thread** ▷ Cf aglo : 4
for $i < \mathcal{N}$ **do**
    Joins les threads
    Insère sortie du thread dans la liste nombre premier.
Trie la liste de nombres premiers
debut $\leftarrow$ arrête chrono
Affiche liste nombre premier.
Affiche $fin - debut$

Algorithm 4 Compute_intervalles_Threads

▷ Fonction intermédiaire pour le mutex

Verrouille le mutex
l_num_intervalle $\leftarrow$ g_num_intervalle
g_num_intervalle += 1
Déverrouille le mutex
if l_num_intervalle $<$ size(g$\mathcal{V}$) **then**
    $\mathcal{I}$ = g$\mathcal{V}$(l_num_intervalle)
    compute_intervalles($\mathcal{I}$) ▷ Cf aglo : 5

Algorithm 5 compute_intervalle

$\mathcal{N} \leftarrow$ intervalle_bas
while $\mathcal{N}$ <intervalle_haut **do** ▷ While car les types mpz_t sont embêtant
    is_prime $\leftarrow$ **mpz_probab_prime_p**($\mathcal{N}$) ▷ Algorithme de miller-rabin (cf lien sous section 3)
    if is_prime == 1 ou is_prime ==2 **then**
        output_list $\hookleftarrow \mathcal{N}$
    $\mathcal{N}$+ = 1

4 Résultats expérimentaux

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 128.
la version séquentielle sert de comparaison pour calculer le speedup.

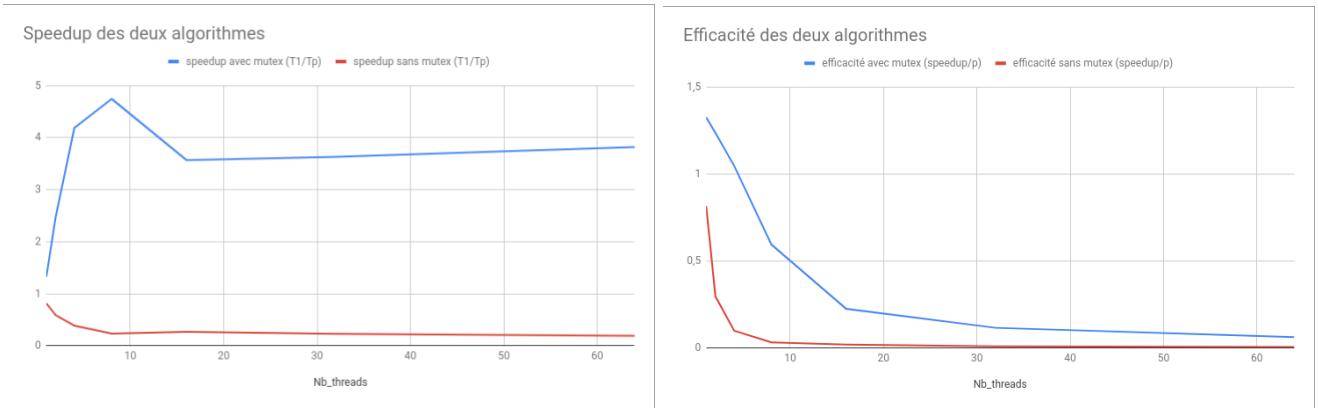

FIGURE 1 – Speedup et efficacité des deux algorithmes
Avec Mutex Sans Mutex

5 Discussion

5.a Analyse des résultats

Les courbes de speedup représentées sont calculés à partir du temps d'exécution de la version séquentielle, qui ne créée pas de threads. Elles auraient aussi pu être calculées à partir du temps d'exécution de la version parallèle à un thread. La différence n'est pas énorme et représente juste un écrasement des courbes mais les allures restent les mêmes.
On remarque que le speedup de la version ave cmutex est maximal pour 8 thread. Cela correspond au cas où il y a exactement un thread par coeur du processeur.
La comparaison des temps d'exécution des deux programmes est à l'avantage de la version avec mutex pour un nombre de threads inférieur à 8, et à l'avantage de la version sans mutex au delà. Les bons résultats avec un faible nombre de threads font de la version avec mutex celle qui maximise le speedup et l'efficacité.
Attention toutefois, aucun des deux programmes n'est suffisamment robuste pour fonctionner si il a trop d'intervalles à traiter. Dans ce cas, le tableau pour stocker les structures qui représentent les intervalles n'est plus allouable en mémoire, la RAM se remplit entièrement et le PC finit par planter.

En conclusion, le programme fonctionne correctement puisqu'il renvoie la liste triée des nombres premiers dans les intervalles donnés en entrée.
Il est capable, au pire, de traiter 500 intervalles de largeur 10^5 dans un temps inférieur à une minute sur l'ordinateur utilisé pour les tests.
Les fils d'exécution ne communiquent pas, et possèdent leurs propres paramètres. Il sont donc totalement indépendants, excepté pour l'utilisation du mutex dans la version qui l'utilise.
Celle-ci n'en utilise qu'un seul, qui protège la liste d'intervalles. Le temps de blocage est cependant extrêmement court, puisqu'il ne correspond qu'à la copie et à l'incrémentation d'une variable de type int. De plus, une version du programme a été écrite spécifiquement pour ne pas utiliser de mutex. On notera que la compilation a été effectuée avec l'option -O3 du compilateur, qui permet d'optimiser la rapidité d'exécution du code. Cette option permet une légère amélioration du temps d'exécution.

5.b Améliorations possibles

Il serait préférable de vérifier en amont la taille du fichier d'entrée, afin d'éviter une saturation de la mémoire. Ici nous avons supposé que tout l'ensemble des intervalles (représenté par une structure) tient en mémoire.
Durant nos tests il nous est arrivé de saturer la RAM et donc d'être obligé de redémarrer le pc. Cela était du à un trop grand nombre de résultats. Il faut s'assurer de ne pas avoir un trop grand nombre de nombre premier parmis nos intervalles (donc des intervalles étroits).

Nous avons aussi vu que lors de certains tests, l'os gère lui même certaines affectation et que même avec 8 thread, certains coeurs ne sont pas toujours utilisés (même avec les 8 logiques sur la machine). Nous sommes donc à la merci de l'os pour les tâches bas niveau et celui n'est pas toujours au mieux de la performance car il doit aussi gèrer d'autres processus.

5.c Difficultés

La première difficulté à été de nous remettre à un langage plus bas niveau. Après quelques jours en revanche, les mécanismes nous sont revenus.

La deuxième à été de construire des blocs dont nous avions besoin pour un algorithme clair. En effet, la librairie GMP n'est pas extrêmement compatible avec le C++ et nous avons du redéfinir des structures et des objets afin d'utiliser de façon simple les variables d'entier de GMP [1] avec les fonction basiques (comme sort) du C++.
Enfin, puisque c'est une nouveauté, trouver la meilleure approche parallèle a été un défi. Ici nous avons comparé deux approches qui nous semblaient correctes mais, en comparant avec d'autres méthodes, il est possible de non pas diviser le fichier en thread mais par exemple l'intervalle lui même.

1. cf la classe Custom_mpz_t dans notre code qui redéfini les opérateurs utiles.