Programmation parallèle et distribuée
GIF-4104

TP 1
Pthread

Équipe 2
Gérémy Desmanche, 111 232 013
Samuel Cloutier, 111 232 614

Le travail demandé consiste à créer un programme parallèle permettant de trouver les nombres premiers contenus à l'intérieur d'intervalles de bornes de taille arbitraire se trouvant dans un fichier reçu en entrée.

Description du programme:

Le programme commence par lire les intervalles du fichier et les place dans un vecteur. Puis, le chronomètre est lancé avant que le vecteur soit trié en ordre croissant à l'aide d'un algorithme inspiré du tri rapide, mais qui priorise le tri de la section inférieure. Les intervalles triés sont ensuite traités afin d'en retirer les intersections. Finalement, une mémoire pour les premiers trouvés est préallouée, chaque intervalle s'y voit assigner une section et on peut lancer les threads.

Les threads sont lancés sans synchronisation puisque leurs espace mémoire et paramètre sont séparés. Après les avoir lancés, le thread principal se comporte comme le premier des threads de travail.

Au sein des threads,

Une approche sans verrouillage est utilisée pour minimiser la surcharge de synchronisation. Chaque thread se voit attribuer des blocs d'intervalles en fonction de leur numéro, ce qui signifie que la distribution du travail est entièrement prédéfinie par l'instance et donc n'a pas besoin de synchronisation outre pour la vérification de disponibilité d'un intervalle à traiter, inhérente au vol de travail.

En effet, pour éviter que la disparité entre les tailles des intervalles ou le temps CPU attribué à chaque thread ne puisse trop ralentir le processus en le forçant à attendre après un unique thread surchargé, une approche par vol de travail est utilisée et les threads ayant fini de traiter leurs blocs tentent de trouver un thread inachevé et traiter ses blocs dans l'ordre inverse.

La fin de ce processus est détectée avec un champ d'intervalle *is_processed*, qui est un drapeau atomique de la STL C++, défini pour fonctionner sans verrouillage.

Les threads volent systématiquement du travail d'un thread à la fois, afin de réduire le risque de collision entre les threads volant actuellement du travail.

Le schéma de la page suivante illustre un exemple d'exécution à 3 threads, 29 intervalles séparés en blocs de 2.
On remarquera les cases grises qui dénote qu'un intervalle est visité, mais non disponible. En particulier, l'ajustement atomique du drapeau, au temps 16, a protégé l'accès à l'intervalle 22.
La séquence 28, 26, 24 est la recherche finale d'un thread duquel du travail peut être volé.

Temps	T_0	T_1	T_2
0	0		
1		2	4
2		3	
3		8	5
4		9	10
5		14	
6			
7		15	11
8		20	
9			
10		21	
11	1	26	16
12		27	
13			
14		28	17
15	6		
16		22	22
17		23	28
18	7		26
19		16	24
20			25
21		28	
22		26	18
23	12	24	19
24	13		12
25	18		28
26	28		26
27	26		24
28	24		

Intervalles

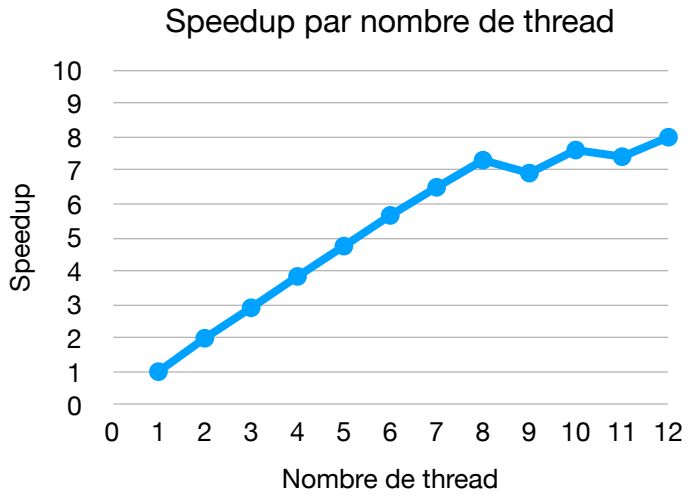
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

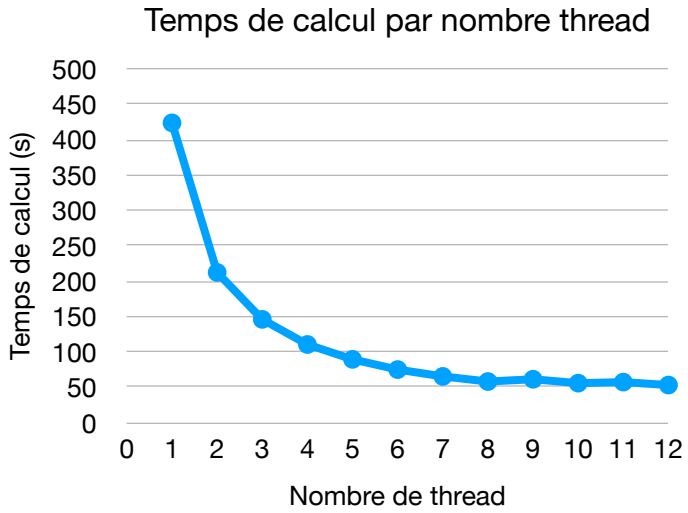
En quittant son espace de travail, le thread principal joint les autres threads puis arrête le chronomètre. L'affichage des résultats a lieu et les ressources de l'instance sont libérées.

Résultats expérimentaux* avec 8_test.txt (tel que publié sur le forum):

Résultats expérimentaux

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	423,51	1	1,000
2	212,14	1,996	0,998
3	145,82	2,904	0,968
4	110,35	3,838	0,960
5	89,326	4,741	0,948
6	74,883	5,656	0,943
7	65,227	6,493	0,928
8	58,004	7,301	0,913
9	61,227	6,917	0,769
10	55,700	7,603	0,760
11	57,190	7,405	0,673
12	53,032	7,986	0,666





*Les résultats ont été obtenus sur un ordinateur ayant les spécificités suivantes:

OS: Arch Linux x86_64
Host: MS-7B79 1.0
Kernel: 5.10.11-arch1-1
Uptime: 3 hours, 47 mins
Packages: 447 (pacman)
Shell: bash 5.1.4
Resolution: 2560x1440
Theme: Adwaita [GTK3]
Icons: Adwaita [GTK3]
Terminal: urxvt
CPU: AMD Ryzen 7 2700X (16) @ 3.700GHz
GPU: AMD ATI Radeon Pro WX 5100
Memory: 3656MiB / 32121MiB

getconf -a | grep CACHE

LEVEL1_ICACHE_SIZE 65536
LEVEL1_ICACHE_ASSOC 4
LEVEL1_ICACHE_LINESIZE 64
LEVEL1_DCACHE_SIZE 32768
LEVEL1_DCACHE_ASSOC 8
LEVEL1_DCACHE_LINESIZE 64
LEVEL2_CACHE_SIZE 524288
LEVEL2_CACHE_ASSOC 8
LEVEL2_CACHE_LINESIZE 64
LEVEL3_CACHE_SIZE 16777216
LEVEL3_CACHE_ASSOC 16
LEVEL3_CACHE_LINESIZE 64
LEVEL4_CACHE_SIZE 0
LEVEL4_CACHE_ASSOC 0
LEVEL4_CACHE_LINESIZE 0

Analyse des résultats:

On observe que pour un nombre de fils d'exécution inférieur ou égal au nombre de coeurs physiques de la machine, l'efficacité est supérieure à 0.913 ce qui nous semble être de bons résultats. En particulier, le gain obtenu en ajoutant le second fil d'exécution était presque parfait, mais décroit pour les fils subséquents. Cependant, il est possible que cette diminution de l'efficacité puisse être associée à l'augmentation de la température liée au plus grand nombre de threads travaillant, ce qui limité la technologie d'overclocking automatique des CPU AMD. Passé ce nombre de thread, l'efficacité diminue plus rapidement, de façon moins régulière alors que le speedup plafonne, ce qui nous semble parfaitement normal.

Discussion:

Tout d'abords, le programme fonctionne correctement. Les fils d'exécutions sont surtout indépendants, mais on réalise en rétrospective que notre approche ne permet qu'à un maximum de deux fils de travailler sur la donnée associée à un thread, ce qui pose problème si un thread est exceptionnellement lent et que le thread lui volant son travail devient lui aussi lent après avoir commencé le vol.

L'usage de mutex a été minimisé : il n'y en a pas.
Le groupement des intervalles pour qu'ils soient traités par un même thread ainsi que la préallocation des listes de premiers en sortie assurent une réduction des invalidations et des défauts de cache. Cet aspect, bien que supporté, n'a cependant pas été optimisé.
Le programme est compilé avec l'optimisation maximale par défaut de g++.

Améliorations possibles:

Après la lecture du fichier à traiter, les intervalles sont ordonnés et filtrés de manière complètement séquentielle. Paralléliser cette étape du code pourrait améliorer l'efficacité totale du programme en réduisant la latence. La séparation par blocs, telle que déjà implémentée, de même que le choix d'algorithme de tri, faciliterait déjà l'implémentation de cette amélioration.

On pourrait explorer les paramètres de compilation g++ à la recherche d'une configuration idéale. Il serait aussi pertinent de configurer de manière optimale la taille des blocs, idéalement en fonction de l'instance à traiter.

Finalement, le vol de travail pourrait être amélioré. Les blocs ne sont jamais subdivisés en unités plus petites (intervalle), mais nous pourrions envisager de diviser le dernier bloc pour minimiser davantage le temps passé à attendre après le dernier fil. Il serait également de bon ton de trouver une manière de permettre à plusieurs fils de voler le travail d'un même fil, et ce sans trop causer de collisions en regard à la cache.