Programmation parallèle et distribuée
GIF-4104

TP 2
OpenMP

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

Le travail consiste à créer avec OpenMP 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. Les intervalles triés sont ensuite traités afin d'en retirer les intersections. Finalement, une mémoire pour les premiers trouvés est préallouée et chaque intervalle s'y voit assigner une section.

Les fils sont lancés à l'aide d'une section parallèle OpenMP dans laquelle ils partagent l'instance du problème (le tableau d'intervalles) et ont chacun un itérateur d'intervalle et un gros entier qu'ils initialisent à leur lancement.

Le travaille parallèle est effectué par une boucle *for* OpenMP avec un ordonnancement dynamique de blocs de quatre intervalles. Pour chaque intervalle, un fil ne fait que le parcourir en ordre croissant pour en trouver les nombres premiers.

En quittant la section parallèle, la fonction principale arrête le chronomètre. L'affichage des résultats a lieu et les ressources de l'instance sont libérées.

Résultats expérimentaux[1]:

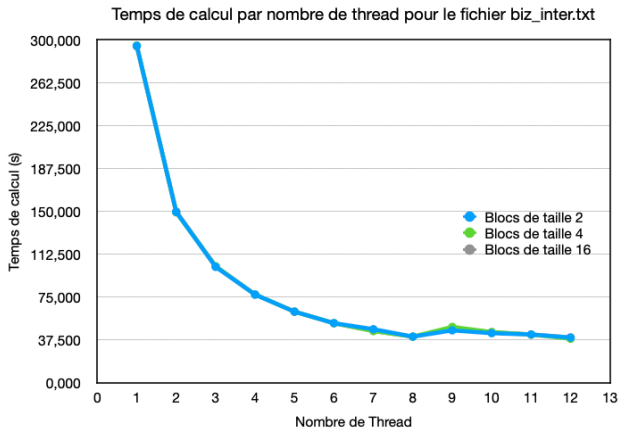
Le fichier biz_inter*.txt* est dans le dossier de remise.

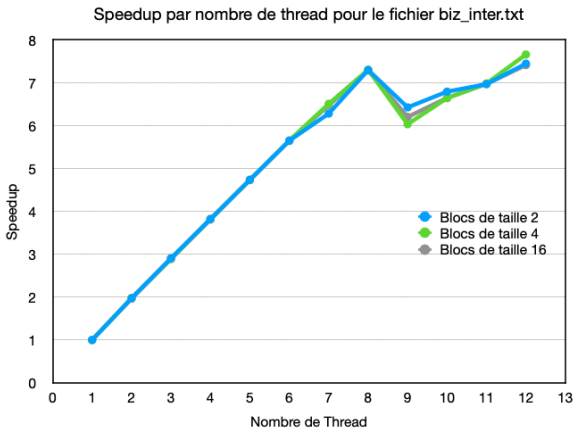
Le fichier *petits_inter.txt* est dans le dossier de remise.

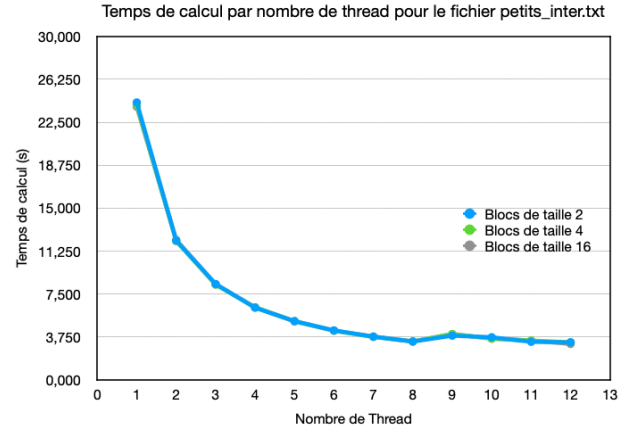
Le fichier gros*_inter.txt* est dans le dossier de remise.

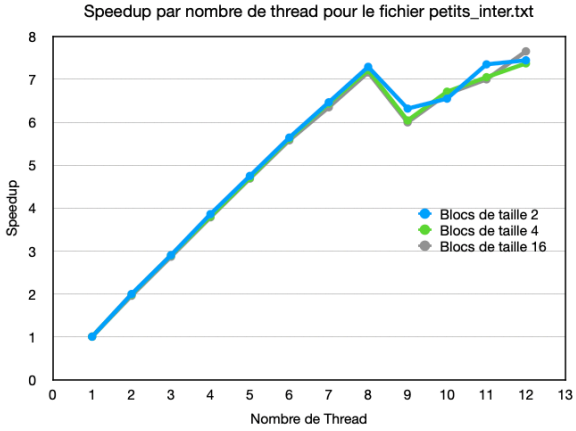
Le fichier *test_8.txt* est tel que retrouvé sur le forum.

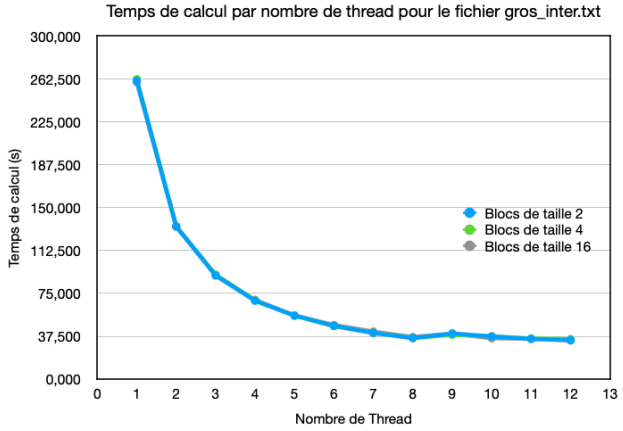
Le fichier *half_test_8.txt* consiste en les 42420 premières lignes du fichier *test_8.txt.*

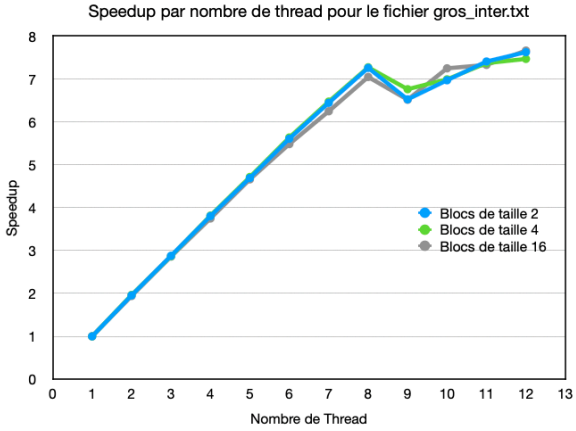












[1] Ces résultats ont été obtenus sur la machine décrite en annexe 1.

Dynamique à blocs de 4 intervalles avec half_test_8.txt

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	179,620	1	1,000
2	90,555	1,984	0,992
3	61,978	2,898	0,966
4	47,814	3,757	0,939
5	37,780	4,754	0,951
6	31,806	5,647	0,941
7	27,819	6,457	0,922
8	24,649	7,287	0,911
9	27,217	6,600	0,733
10	27,584	6,512	0,651
11	25,556	7,029	0,639
12	24,458	7,344	0,612

Résultats expérimentaux (TP1) avec test_8.txt

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

Analyse des résultats:

Pour commencer, les tableaux associés aux graphiques de la page précédente se trouvent à l'annexe 1. Ils contiennent également l'efficacité calculée à partir de chaque speedup.

Les résultats sont bons ; pour un nombre de fils d'exécution inférieur ou égal au nombre de coeurs physiques de la machine, l'efficacité est élevée (supérieure à 0.9 en général). Elle diminue cependant plus rapidement plus le nombre de fils augmente, ce qui peu être associé en partie à l'augmentation de l'impact de la synchronisation sur un plus grand nombre de fils, mais aussi possiblement à l'overclocking automatique du CPU AMD, qui est moins prononcé lorsque le nombre de thread est élevé à des fins d'autoconservation (problème de chaleur).

On remarquera que l'utilisation de OpenMP semble avoir légèrement impacté à la baisse l'efficacité du programme par rapport à la version sans synchronisation fournie au TP1, à l'exception du déroulement à cinq threads.

Le but de tester avec les fichiers biz_inter.txt, petits_inter.txt et gros_inter.txt était d'étudier le comportement du programme lorsqu'il doit traiter des fichiers d'intervalles de tailles variables, ainsi que l'influence de la taille des bloc sur celui-ci. Par exemple, le fichier biz_inter.txt contient des intervalles de taille entre 500 et 2000, petits_inter.txt des intervalles de taille entre 1 et 200 et gros_inter.txt des intervalles de taille entre 10000 et 100000. Les résultats obtenus montrent que ces variations ont peu d'impacte sur le speedup du programme.

Discussion:

Tout d'abords, le programme fonctionne correctement. En dehors de l'utilisation de OpenMP, le travail des threads est complètement indépendant.

Un avantage à l'utilisation de OpenMP est qu'il prend en charge l'ordonnancement des threads: en mode dynamique, il n'est plus nécessaire d'implémenter notre propre algorithme de répartition et de vol de travail comme dans le TP1, ce qui résulte en un code beaucoup plus simple.

Par rapport à cet ordonnancement, nous avons effectué plusieurs tests afin d'évaluer les modes dynamique, guidé et même statique avec différentes tailles (minimales) de blocs. En général, vu la taille du travail effectué à chaque itération, les différents paramètres d'ordonnancement donnent des résultats semblables. Pour réduire les collisions de cache, nous avons opté pour une taille de quatre intervalles. (Voir annexe 4) Pour limiter l'impact négatif d'un fichier d'entrée pour lequel la répartition des gros intervalles serait « pathologique » (au début du fichier pour guidé, à chaque nb_thread intervalle pour statique), nous avons opté pour le mode dynamique.

Certaines données expérimentales se trouvent en annexe 1 et 2.

Le programme est compilé avec l'optimisation maximale par défaut de g++.

Améliorations possibles:

Nous sommes satisfaits des résultats obtenus. La simplicité du code valant de loin la possible amélioration du speedup qu'on tirerait d'une complexification significative du code.

On pourrait tout-de-même envisager de paralléliser le tri des intervalles dans un thread distinct, mais il faudrait alors trouver un moyen de malgré tout minimiser la synchronisation nécessaire.

Il pourrait être pertinent de découper les intervalles afin de mieux traités les fichiers contenant quelques intervalles très gros, mais cela demanderait une modification importante du modèle du programme.

On pourrait finalement explorer les paramètres de compilation g++ à la recherche d'une configuration idéale.

Annexe 1 (Calculs avec l'ordinateur Linux[1]):

Les temps de calculs sont obtenus sur la moyenne de trois essais.

Résultats pour le fichier biz_inter.txt avec des blocs de taille 2

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	294,659	1	1
2	149,425	1,972	0,986
3	101,381	2,906	0,969
4	77,167	3,818	0,955
5	62,239	4,734	0,947
6	52,187	5,646	0,941
7	46,935	6,278	0,897
8	40,399	7,294	0,912
9	45,877	6,423	0,714
10	43,399	6,789	0,679
11	42,309	6,965	0,633
12	39,591	7,443	0,620

Résultats pour le fichier biz_inter.txt avec des blocs de taille 4

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	294,914	1	1
2	149,063	1,978	0,989
3	101,922	2,894	0,965
4	77,175	3,821	0,955
5	62,282	4,735	0,947
6	52,189	5,651	0,942
7	45,330	6,506	0,929
8	40,392	7,301	0,913
9	48,906	6,030	0,670
10	44,422	6,639	0,664
11	42,252	6,980	0,635
12	38,522	7,656	0,638

[1] 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

Résultats pour le fichier biz_inter.txt avec des blocs de taille 16

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	294,248	1	1
2	149,823	1,964	0,982
3	101,909	2,887	0,962
4	77,259	3,809	0,952
5	62,246	4,727	0,945
6	52,163	5,641	0,940
7	45,456	6,473	0,925
8	40,401	7,283	0,910
9	47,468	6,199	0,689
10	44,302	6,642	0,664
11	42,222	6,969	0,634
12	39,707	7,410	0,618

Résultats pour le fichier petits_inter.txt avec des blocs de taille 2

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	24,229	1	1
2	12,151	1,994	0,997
3	8,352	2,901	0,967
4	6,281	3,858	0,964
5	5,103	4,748	0,950
6	4,295	5,642	0,940
7	3,748	6,465	0,924
8	3,323	7,292	0,911
9	3,835	6,317	0,702
10	3,700	6,549	0,655
11	3,297	7,349	0,668
12	3,255	7,443	0,620

Résultats pour le fichier petits_inter.txt avec des blocs de taille 4

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	24,006	1	1
2	12,106	1,983	0,992
3	8,296	2,894	0,965
4	6,315	3,802	0,950
5	5,105	4,703	0,941
6	4,275	5,615	0,936
7	3,730	6,437	0,920
8	3,328	7,213	0,902
9	3,975	6,040	0,671
10	3,577	6,711	0,671
11	3,406	7,049	0,641
12	3,257	7,371	0,614

Résultats pour le fichier petits_inter.txt avec des blocs de taille 16

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	23,853	1	1
2	12,205	1,954	0,977
3	8,327	2,864	0,955
4	6,306	3,783	0,946
5	5,091	4,686	0,937
6	4,276	5,578	0,930
7	3,757	6,348	0,907
8	3,330	7,162	0,895
9	3,978	5,997	0,666
10	3,581	6,661	0,666
11	3,408	6,999	0,636
12	3,117	7,652	0,638

Résultats pour le fichier gros_inter.txt avec des blocs de taille 2

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	260,936	1	1
2	133,476	1,955	0,977
3	90,677	2,878	0,959
4	68,612	3,803	0,951
5	55,619	4,691	0,938
6	46,535	5,607	0,935
7	40,496	6,444	0,921
8	35,949	7,259	0,907
9	39,991	6,525	0,725
10	37,438	6,970	0,697
11	35,220	7,409	0,674
12	34,219	7,625	0,635

Résultats pour le fichier gros_inter.txt avec des blocs de taille 4

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	262,299	1	1
2	133,488	1,965	0,982
3	91,430	2,869	0,956
4	68,751	3,815	0,954
5	55,629	4,715	0,943
6	46,542	5,636	0,939
7	40,520	6,473	0,925
8	36,066	7,273	0,909
9	38,790	6,762	0,751
10	37,514	6,992	0,699
11	35,593	7,369	0,670
12	35,121	7,469	0,622

Résultats pour le fichier gros_inter.txt avec des blocs de taille 16

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	259,836	1	1
2	133,887	1,941	0,970
3	90,847	2,860	0,953
4	69,312	3,749	0,937
5	55,756	4,660	0,932
6	47,394	5,482	0,914
7	41,588	6,248	0,893
8	36,870	7,047	0,881
9	39,870	6,517	0,724
10	35,846	7,249	0,725
11	35,452	7,329	0,666
12	33,913	7,662	0,638

Résultats pour le fichier half_test_8.txt en mode guided à 8 threads

Taille minimale des blocs	Temps de calcul (s)
1	24,68
2	24,66
4	24,65
8	24,67
16	24,66
42	24,67

Résultats pour le fichier half_test_8.txt en mode dynamique à 8 threads

Taille des blocs	Temps de calcul (s)
1	24,65
2	24,63
4	24,64
8	24,65
16	24,68
42	24,71

Annexe 2 (Calculs sur test_8.txt avec une machine virtuelle Ubuntu sur Mac[1]):

Les temps de calculs sont obtenus sur la moyenne de deux essais.

Dynamique à blocs de 1 intervalle

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	638,387	1	1,000
2	332,251	1,921	0,961
4	174,910	3,650	0,912

Statique à blocs de 1 intervalle

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	675,010	1	1,000
2	328,989	2,052	1,026
4	173,141	3,899	0,975

Dynamique à blocs de 16 intervalles

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	634,462	1	1,000
2	325,716	1,948	0,974
4	172,505	3,678	0,919

Statique à blocs de 16 intervalles

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	639,097	1	1,000
2	337,314	1,895	0,947
4	180,777	3,535	0,884

Dynamique à blocs de 64 intervalles

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	640,016	1	1,000
2	326,922	1,958	0,979
4	172,927	3,701	0,925

Statique à blocs de 64 intervalles

Nombre de thread	Temps de calcul (s)	Speedup (T_1/T_p)	Efficacité
1	639,010	1	1,000
2	336,401	1,900	0,950
4	176,940	3,611	0,903

[1] Les résultats ont été obtenus sur un ordinateur ayant les spécificités suivantes:

System: Host: etudiant Kernel: 4.4.0-201-generic x86_64 (64 bit gcc: 5.4.0)
Desktop: Xfce 4.12.3 (Gtk 2.24.28) Distro: Ubuntu 16.04 xenial
Machine: System: VMware product: VMware Virtual Platform
Mobo: Intel model: 440BX Desktop Reference Platform
Bios: Phoenix v: 6.00 date: 07/22/2020
CPU(s): 4 Single core[2] Intel Core i5-8259Us (-SMP-) cache: 24576 KB[
flags: (lm nx sse sse2 sse3 sse4_1 sse4_2 ssse3) bmips: 18432
clock speeds: max: 2304 MHz 1: 2304 MHz 2: 2304 MHz 3: 2304 MHz
m4: 2304 MHz
Drives: HDD Total Size: 21.5GB (66.8% used)
ID-1: /dev/sda model: VMware_Virtual_S size: 21.5GB
Info: Processes: Uptime: Memory: 572.3/7983.1MB
Init: systemd runlevel: 5 Gcc sys: 5.4.0
Client: Shell (bash 4.3.481) inxi: 2.2.35

[2] Ici, la VM voit le nombre de coeurs virtuels plutôt que de coeurs physiques.

Annexe 3 (Utilitaire de test):

Les donnés *files*, *thread_counts* et *block_sizes* sont là à titre d'exemple.

```
import subprocess

import subprocess
import sys

binaries = ['./findprimes-guided', './findprimes-dynamic']
test_count = 4
files = [ 'test_files/biz_inter.txt', 'test_files/petits_inter.txt',
'test_files/gros_inter.txt' ]
thread_counts = [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
block_sizes = [2, 4, 16]

tests = [ {'bin': b, 'file': f, 'thread_count': c, 'block_size': s,
'durations': []}
      for b in binaries
      for f in files
      for c in thread_counts
      for s in block_sizes
]
for i in range(test_count):
      for test in tests:
            test['durations'] += [float(subprocess.run(
                        [     test['bin'], str(test['thread_count']),
                              test['file'], str(test['block_size'])
                        ], stderr=subprocess.PIPE
                  ).stderr)
            ]
            with open(sys.argv[1], 'a') as f:
                  f.write(str(test) + '\n')
```

Annexe 4 (Efficacité de la mémoire cache):

Résultats obtenus avec valgrind --tool=cachegrind ./findprimes 4 8_test.txt (interrompu après plus d'une heure)

```
==2210== I   refs:      1,267,026,208,687
==2210== I1  misses:            1,872,725
==2210== LLi misses:                2,866
==2210== I1  miss rate:              0.00%
==2210== LLi miss rate:              0.00%
==2210==
==2210== D   refs:        498,476,373,648  (315,902,298,309 rd   +
182,574,075,339 wr)
==2210== D1  misses:            4,155,102  (      3,477,273 rd   +
677,829 wr)
==2210== LLd misses:              698,635  (        220,558 rd   +
478,077 wr)
==2210== D1  miss rate:               0.0% (            0.0%     +
0.0%  )
==2210== LLd miss rate:               0.0% (            0.0%     +
0.0%  )
==2210==
==2210== LL refs:               6,027,827  (      5,349,998 rd   +
677,829 wr)
==2210== LL misses:               701,501  (        223,424 rd   +
478,077 wr)
==2210== LL miss rate:                0.0% (            0.0%     +
0.0%  )
```