UNIVERSITÉ
LAVAL

TP 2 - OpenMP

Réalisé par **l'équipe 1** constitué de :

Rémi Huguet - 536 795 651
Gabriel Phillipon - 536 795 815

Dans le cadre du cours :
GIF-7104 – Programmation parallèle et distribuée

Travail présenté à :
Marc Parizeau

Remis le :
22 Février 2021

Objectif

L'objectif de ce TP est de trouver tous les nombres premiers dans une série d'intervalles, grâce à un programme multifilaire utilisant la librairie OpenMP. Le but est également de mesurer le Speedup entre l'implémentation séquentielle et l'implémentation parallèle de ce programme.

Solution

Tout comme dans le TP pthread, notre solution a été de répartir le nombre d'intervalles des fichiers à travers plusieurs threads. En effet, de très gros fichier (comme le fichier `8_test.txt`) contiennent un grand nombre d'intervalles qu'il faut repartir en sous-groupe d'intervalles. Ainsi dans un premier temps on construit T (nombre de threads) sous-groupes d'intervalles contenant un nombre égal (ou presque, tout dépend du reste) d'intervalles à traiter.

Prise en compte des pistes d'amélioration du TP pthread

Dans le cas ou le nombre d'intervalles est inférieur ou égal au nombre de thread demandé, on divise les intervalles de façon à avoir au moins un intervalle par thread. Par ailleurs dans le cas du traitement de recouvrement, on peut se retrouver avec des intervalles dont la quantité de nombres premiers à vérifier est beaucoup plus grands, pour certains thread, que pour d'autres.

Ainsi, on fait sorte d'attribuer à chaque thread un nombre d'intervalles quasi-équivalent et de même taille. De ce fait, chaque thread traitera la même quantité de donnée et aura donc un temps d'exécution quasi-égale. De plus, de cette façon, on minimise l'inactivité des threads. (voir section : Temps par thread)

> Prenons par exemple les intervalles (avec recouvrements) suivants : [22,43], [30,60], [63,100], [85,100]
> Pour un nombre de thread égal à 4, ils vont être gérés comme suit :
>
> — Le thread 1 va gérer : [22,32], [33,42]
>
> — Le thread 2 va gérer : [43,51], [52,60]
>
> — Le thread 3 va gérer : [63,73], [74,82]
>
> — Le thread 4 va gérer : [83,91], [92,100]

OpenMP

```
1  /* Partie 1 */
2  vector master_primeNumbers; /// Master vector
3
4  /* Partie 2 */
5  #pragma omp parallel default(none) shared(intervals, master_primeNumbers)
6  {
7      vector slave_primeNumbers;  /// Slave vectors
8
9  /* Partie 3 */
10 #pragma omp for nowait schedule(dynamic,1)
11     for (size_t i = 0; i < intervals.size(); ++i) {
```

```
12          slave_primeNumbers.emplace_back(computePrime(intervals[i]));
13      }
14
15  /* Partie 4 */
16  #pragma omp critical
17      {
18          mergeMasterAndSlavesVectors(master_primeNumbers, slave_primeNumbers);
19      };
20  };
```

Listing 1 – Code simplifié openMP

Partie 1 : On crée un vector `master_primeNumbers` qui aura pour fonction de recueillir les résultats de chaque `slave_primeNumbers` (un par thread).

Partie 2 : Le travail parallèle s'effectue dans la directive `#pragma omp parallel` d'openMP. Chaque thread aura un `slave_primeNumber` dans lequel il stockera les résultats du calcul de nombre premier.

Partie 3 : Les threads n'ont pas besoin de s'attendre entre eux puisqu'ils sont totalement indépendant, et traite des données qui le sont aussi. De ce fait, il n'est pas non plus nécessaire d'attendre que **tous** les threads ait fini pour passer à la suite du programme. On ajoute donc l'instruction `nowait` pour supprimer la barrière de fin de boucle. Ainsi dès qu'un thread a fini son traitement sur ses intervalles de nombre, il peut directement passer à la suite.

Enfin, connaissant notre nombre d'itération on peut travailler en précisant manuellement le `block_size`, ce qui nous permet de gagner un peu de temps. Ainsi avec `schedule(dynamic,1)` on précise que chaque thread s'occupera d'une itération.

Partie 4 : Pour construire le `vector master_primeNumber` on insère **un par un** (instruction `omp critical`) les résultats de chaque slave_primeNumber dans le vector `master_primeNumber`. De cette façon, on évite les overwrites et les courses critiques. Ici, `pragma omp critical` agit donc comme un mutex.

Spécification Matériel

— OS : Ubuntu 20.10

— Processeur : Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8

— RAM : 8 Go

Résultats

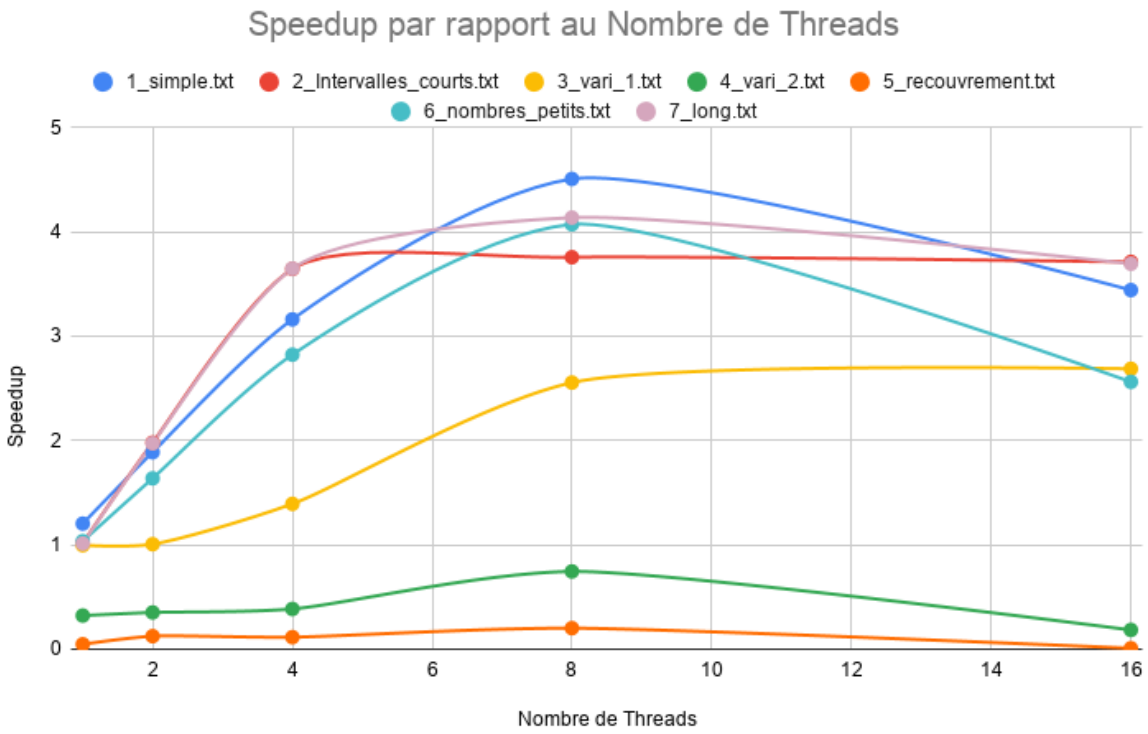

FIGURE 1 – Speedup pour les fichiers tests de 1 à 7

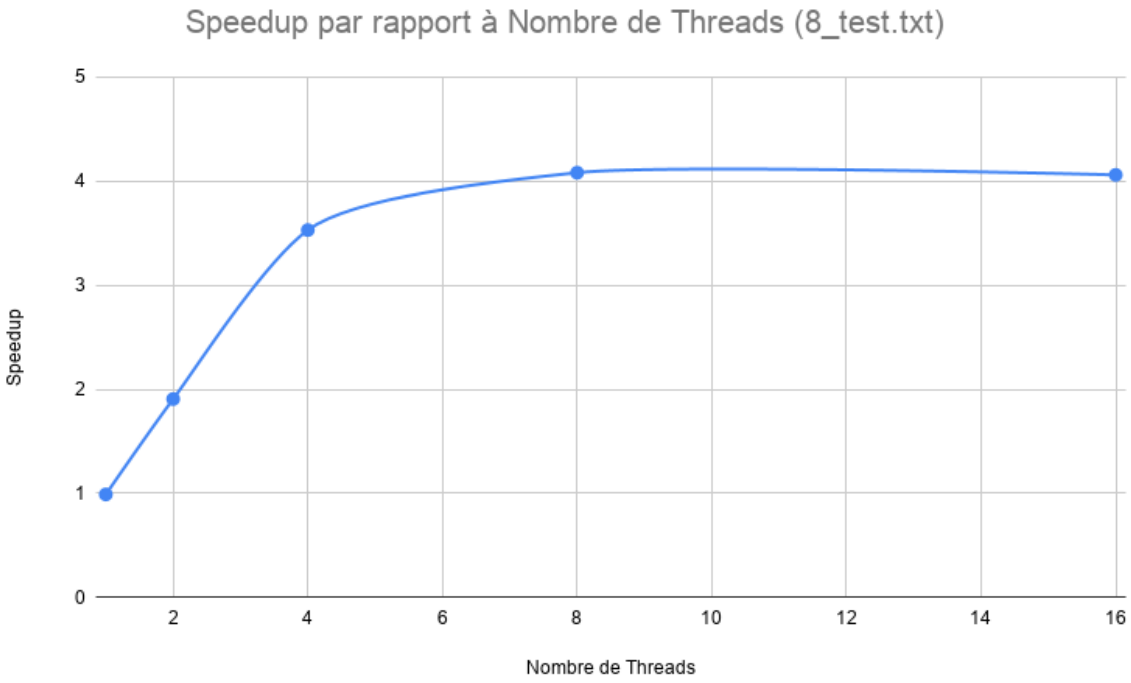

FIGURE 2 – Speedup pour le fichier 8_test.txt

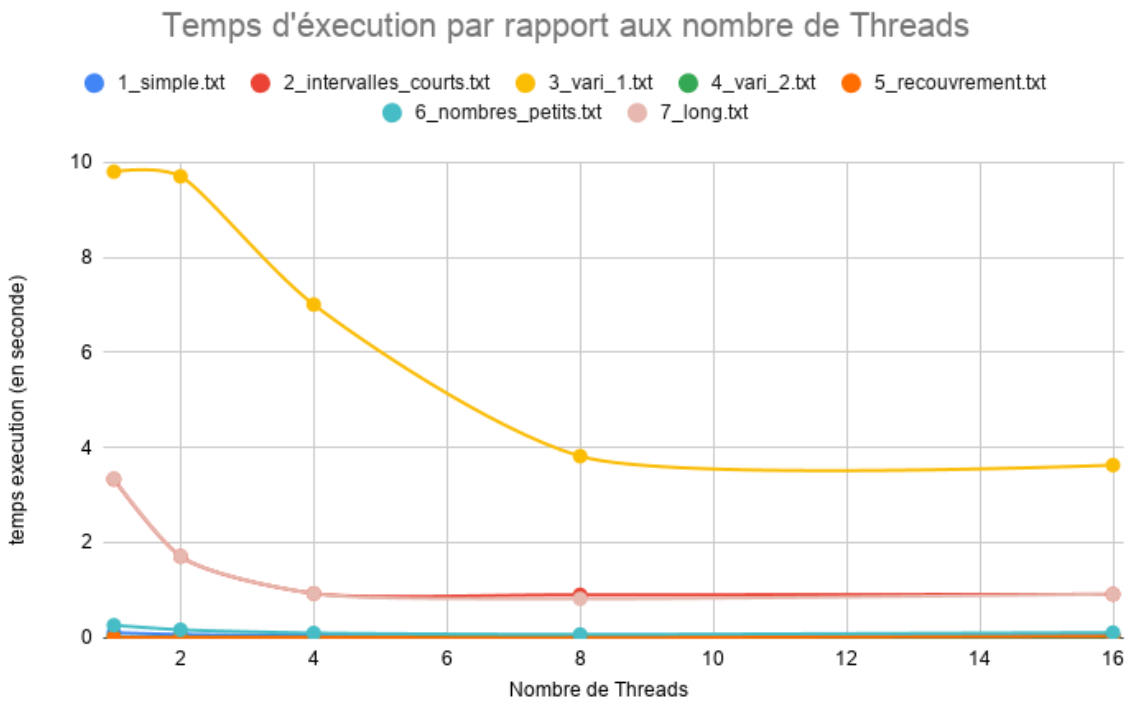

FIGURE 3 – Temps d'exécution pour les fichiers tests de 1 à 7

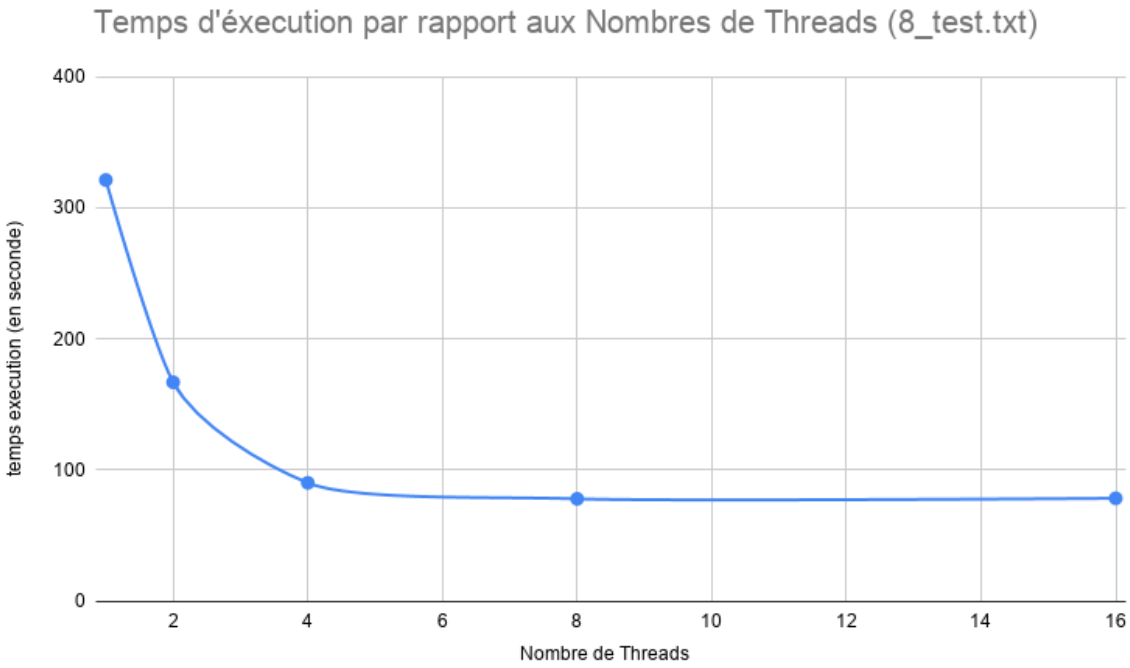

FIGURE 4 – Temps d'exécution pour le fichier 8_test.txt

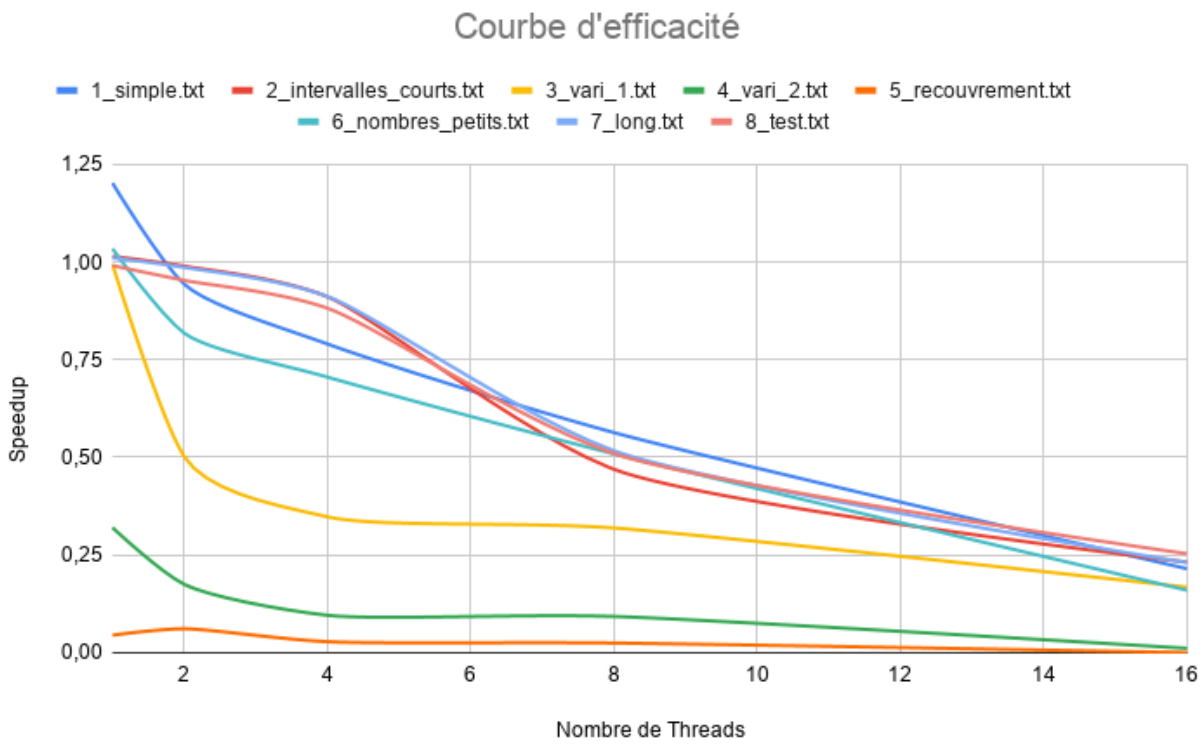

FIGURE 5 – Courbe d'efficacité pour les 8 fichiers tests

Voir les données brutes en Annexe.

Analyse

Notre solution est conforme à ce que l'on attendait. Elle est efficace sur les fichiers comportant, un grand nombre d'intervalles ET minimisant les temps d'inactivité ; C'est à dire des threads dont la charge de travail est quasi-égale.

En effet, les 5 premiers fichiers du tableau (les plus performants) traitent des intervalles de même taille, et dont les nombres à traiter sont quasiment de même grandeur. (voir section : Piste d'amélioration)

On retrouve les effets de l'application des pistes d'amélioration lié au TP pthread. En effet, les fichiers `1_simple` et `6_nombre_petits` conservent des speedups décents.

Fichier	1_simple	7_long	8_test	6_nombre_petit	2_intervalles_courts
Nombres Intervalles (après optimisation)	8	7944	800224	8	8200
Speedup (8 threads)	Bon (4,5)	Bon (4.1)	Bon (4.1)	Bon (4.0)	Bon (3.75)

Fichier	3_vari_1	4_vari_2	5_recouvrements
Nombres Intervalles (après optimisation)	48	64	16
Speedup (8 threads)	Moyen (2.5)	Mauvais (0.75)	Mauvais (0.2)

Comparaison avec Pthread

Dans l'ensemble les speedups sont légèrement moins bon qu'avec l'implémentation pthread (voir annexe).

Cela peut s'expliquer par l'utilisation d'un mutex (simulé par la directive `#pragma omp critical`) dans l'implémentation openMP, alors qu'il n'y en avait pas dans l'implémentation pthread.

A partir de 8 threads, le speedup stagne. En effet, c'est le comportement attendu, puisque sur la machine de test nous avons un processeur 8 core ; donc à partir de plus de 8 threads, le programme commence à travailler en concurrence et non plus en parallèle et donc ralentit.

De plus, on remarque que la gestion de la concurrence (au delà de 8 thread) est moins bonne qu'avec pthread pour les fichiers `1_simple.txt` et `6_nombre_petits.txt`.

En effet, dans notre implémentation pthread nous conservions des speedup à 8 et 16 threads quasi-égaux ; Ici le speedup à 16 threads chute de manière assez significative. Cela est du à la découpe d'intervalles qui devient non optimale.

Temps par thread

Du coté du temps de travail sur les threads nous avons atteint ce que nous voulions. En effet, chaque thread possède un temps de travail quasi-équivalent. Prenons par exemple le fichier `7_long.txt` :

```
--- Threads Time Detail ---
Total accumulated time : 6.07166
Thread 0 worked for 0.731270 (s) - 12.0440%
Thread 1 worked for 0.738217 (s) - 12.1584%
Thread 2 worked for 0.754448 (s) - 12.4257%
Thread 3 worked for 0.756010 (s) - 12.4515%
Thread 4 worked for 0.764933 (s) - 12.5984%
Thread 5 worked for 0.771329 (s) - 12.7038%
Thread 6 worked for 0.772391 (s) - 12.7212%
Thread 7 worked for 0.783062 (s) - 12.8970%
```

Optimisations

Pour tirer un maximum partie de la machine sur lequel le programme a été exécuter et afin d'optimiser les temps d'exécution, la compilation est faite en `"-O3"` (voir `CMakeLists.txt`). Pour ce qui est des structures de données, nous avons préférés les types `Tuples` (std) au `struct`, puisqu'elles sont légèrement plus rapide.

Par ailleurs, étant donné qu'aucun nombre premier n'est pair (hormis 2), on ne traite que les nombres impairs.

Piste d'amélioration

Dans certain cas particulier, on peut avoir quelques threads qui calcul des nombres très petits, puis d'autres qui calcul des nombres très grand. Cela crée un déséquilibre au niveau du traitement de chaque thread, que la segmentation des intervalles ne peut résoudre. En effet, plus un nombre est grand, plus il va demander de temps de traitement. On peut bien voir ce phénomène avec le fichier `4_vari_2.txt` (voir Annexe)

```
--- Threads Time Detail ---
Total accumulated time : 0.00589299
Thread 0 worked for 0.000133038 (s) - 2.25756%
Thread 1 worked for 0.000171900 (s) - 2.91702%
Thread 2 worked for 0.000170946 (s) - 2.90084%

Thread 3 worked for 0.000236988 (s) - 4.02152%
Thread 4 worked for 0.000293016 (s) - 4.97229%

Thread 5 worked for 0.000841856 (s) - 14.2857%
Thread 6 worked for 0.001382110 (s) - 23.4535%
Thread 7 worked for 0.002663140 (s) - 45.1916%
```

Les threads 0,1,2 gèrent des nombres de grandeur 10^9.
Les threads 3,4 gèrent des nombres de grandeur 10^{49}.
Les threads 5,6,7 gèrent des nombre de grandeur 10^{99}.
Ainsi, on voit bien que plus les nombres sont grand, plus le temps de traitement augmente. On pourrait donc penser à une méthode qui repartirait uniformément la complexité des données à travers les threads.

Annexe

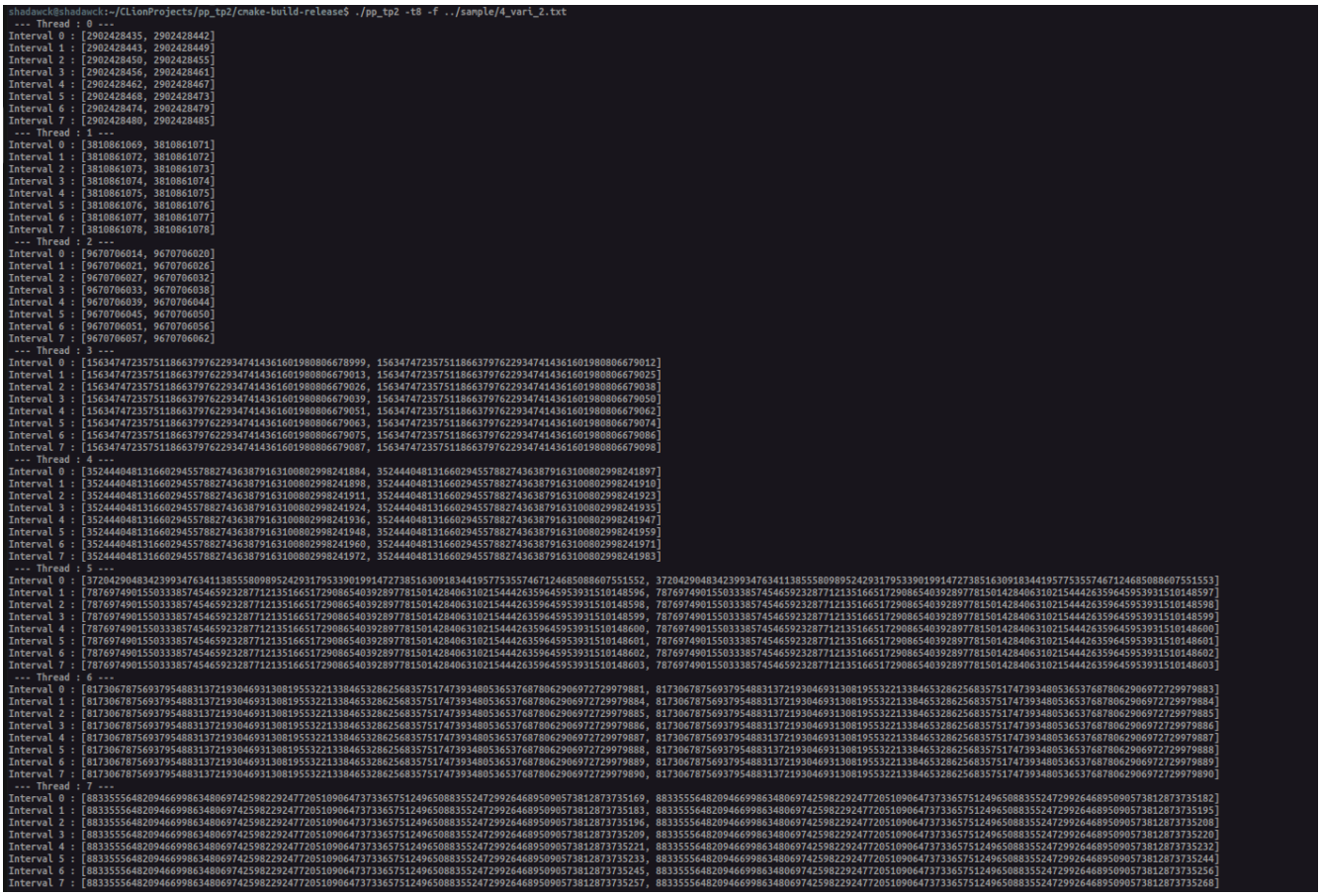

FIGURE 6 – Grandeur des nombres constituant les intervalles pour chaque threads

Données brutes pour chaque fichier de test

Consultable sur un Google Sheet

FILE 1 : simple.txt				
Nombre de Threads	Speedup	Temps execution sequentiel (s)	Temps execution parallèle (s)	Efficacité
1	1,202571622	0,122613	0,101959	1,202571622
2	1,88777694		0,064951	0,9438884698
4	3,162166344		0,038775	0,7905415861
8	4,509488783		0,02719	0,5636860978
16	3,443024823		0,035612	0,2151890514

FILE 2 : intervalles_courts.txt				
Nombre de Threads	Speedup	Temps execution sequentiel (s)	Temps execution parallèle (s)	Efficacité
1	1,014825772	3,381504	3,332103	1,014825772
2	1,979548302		1,70822	0,9897741509
4	3,646231334		0,927397	0,9115578334
8	3,758266454		0,899751	0,4697833067
16	3,713351357		0,910634	0,2320844598

FILE 3 : vari_1.txt				
Nombre de Threads	Speedup	Temps execution sequentiel (s)	Temps execution parallèle (s)	Efficacité
1	0,9949951393	9,752019	9,801072	0,9949951393
2	1,005042206		9,703094	0,5025211031
4	1,391957834		7,005973	0,3479894584
8	2,555609423		3,815927	0,3194511779
16	2,687777938		3,628283	0,1679861211

FILE 4 : vari_2.txt				
Nombre de Threads	Speedup	Temps execution sequentiel (s)	Temps execution parallèle (s)	Efficacité
1	0,3204671581	0,003869	0,012073	0,3204671581
2	0,3503894222		0,011042	0,1751947111
4	0,383943634		0,010077	0,0959859085
8	0,7438954047		0,005201	0,09298692559
16	0,1823709639		0,021215	0,01139818525

FILE 5 : recouvrement.txt				
Nombre de Threads	Speedup	Temps execution sequentiel (s)	Temps execution parallèle (s)	Efficacité
1	0,04499484	0,000218	0,004845	0,04499484004
2	0,122886133		0,001774	0,06144306652
4	0,1134235172		0,001922	0,02835587929
8	0,1996336996		0,001092	0,02495421245
16	0,0068544837		0,031804	0,000428405232

FILE 6 : nombres_petits.txt				
Nombre de Threads	Speedup	Temps exécution sequentiel (s)	Temps exécution parallèle (s)	Efficacité
1	1,034571151	0,268794	0,259812	1,034571151
2	1,637021383		0,164197	0,8185106914
4	2,823051232		0,095214	0,705762808
8	4,07387087		0,06598	0,5092338587
16	2,562554222		0,104893	0,1601596389

FILE 7 : long.txt				
Nombre de threads	Speedup	Temps exécution séquentiel (s)	Temps exécution parallèle (s)	Efficacité
1	1,010897679	3,378416	3,341996	1,010897679
2	1,973144477		1,712199	0,9865722384
4	3,649051558		0,925834	0,9122628895
8	4,138177026		0,816402	0,5172721282
16	3,695687036		0,914151	0,2309804398

FILE 8 : test.txt				
Nombre de Threads	Speedup	Temps execution sequentiel (s)	Temps execution parallèle (s)	Efficacité
1	0,9911901235	318,460914	321,291452	0,9911901235
2	1,906839568		167,00981	0,9534197841
4	3,528693841		90,24895	0,8821734602
8	4,079585822		78,06207	0,5099482277
16	4,058846374		78,460943	0,2536778984

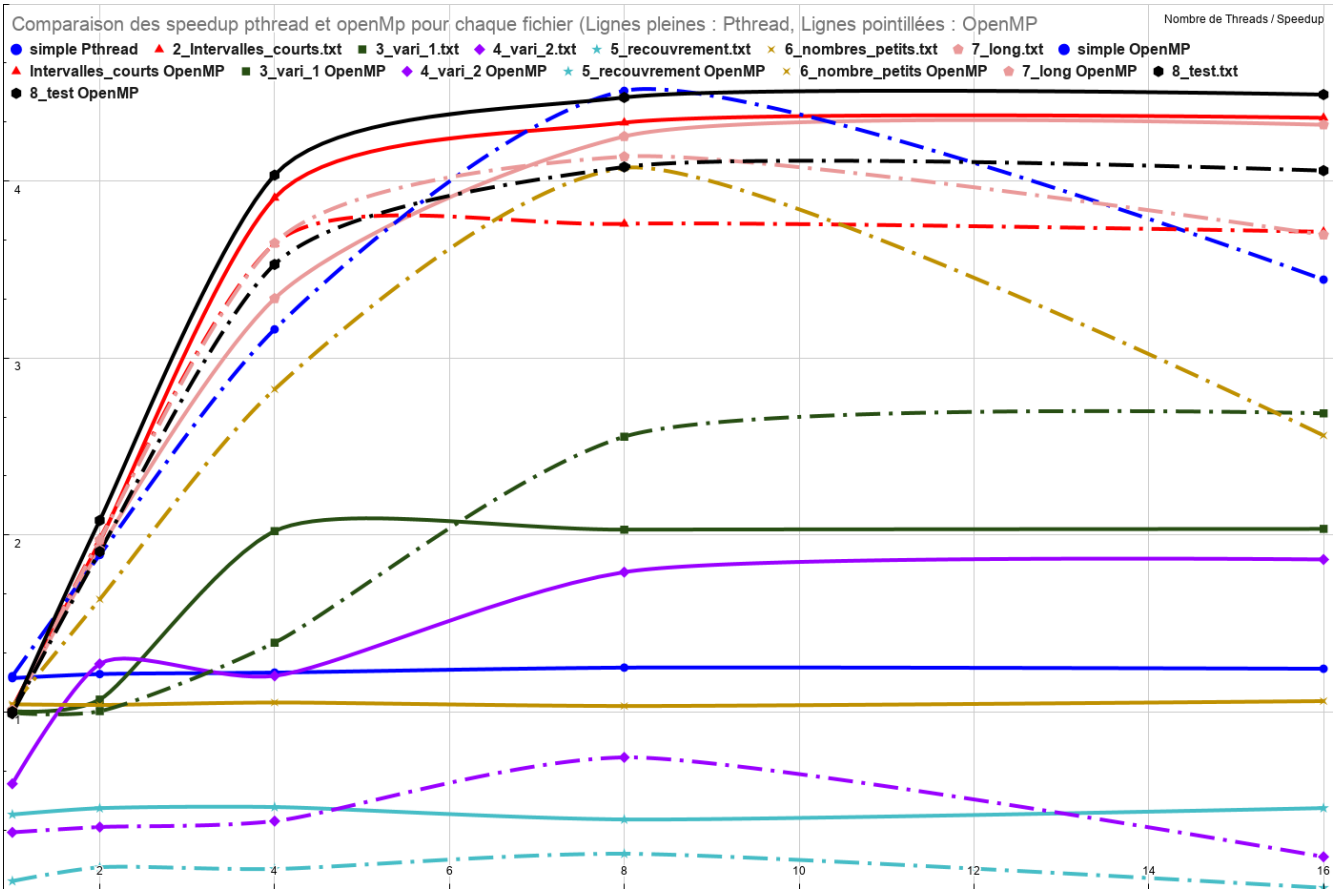

FIGURE 7 – Comparaison des speedup pthread et openMP pour chaque fichier (Lignes pleines pthread, Lignes pointillées OpenMP)