COMPTE RENDU DU PROJET DE SYSTEMES

SUJET : RECHERCHE DE NOMBRES PREMIERS

Groupe n°1 :
BOUHENAF Irfan
MOLINA Romain

M3101 – Principe des systèmes d'exploitation.
Professeur : Romain RAFFIN

DESCRIPTION DU SUJET

On cherche à calculer l'ensemble des nombres premiers. Le test de primarité se fera par décomposition des diviseurs. On élimine les nombres pairs (sauf 2). Les nombres trouvés doivent être écris dans un fichier.

RESOLUTION DU PROBLEME

DESCRIPTION

- On cherche à trouver les nombres premiers sur un l'intervalle [2,n]
- Pour ce faire, on doit trouver les diviseurs (sauf 1 et lui-même) d'un nombre, s'il en possède un sauf 1 et lui même, ce n'est pas un nombre premier, sinon, c'en est un. Puis, on répète ça pour les nombres de 2 à n.
- Ensuite, on doit écrire les nombres premiers dans un fichier texte.

RESOLUTION DU PROBLEME

DESCRIPTION DE LA MACHINE DE TEST

Modèle	Mémoire vive	Processeur	Carte graphique	RAM disponible avant les calculs	RAM disponible après les calculs	Système d'exploitation
Acer AN515-52	8 Go DDR4	Intel Core i7-8750h 6 cœurs 12 threads cadencés à 2,2 GHz et jusqu'à 4 GHz	NVIDIA GTX 1050 4go de mémoire vidéo GDDR5	2,6 Go	2,6 Go	Windows 10

RESOLUTION DU PROBLEME

LIENS VERS LES DOCUMENTATIONS UTILISEES

- https://docs.microsoft.com/fr-fr/cpp/parallel/openmp/reference/openmp-clauses?view=vs-2019
- http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-122/Decouverte-de-la-programmation-parallele-avec-OpenMP
- https://fr.cppreference.com/w/cpp/chrono

ALGORITHME EN SEQUENTIEL

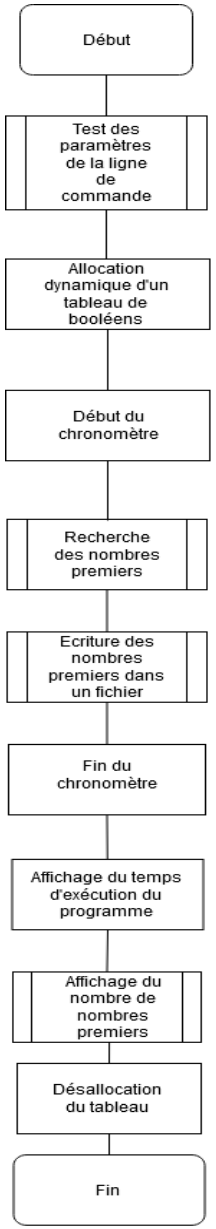
SCHEMA NON DETAILLE + LEGENDE

Début/Fin d'un algorithme

Appel d'un sous-programme

Instruction(s)

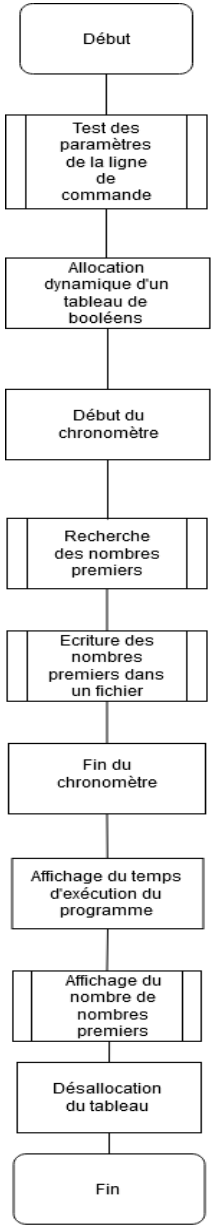
Condition



ALGORITHME EN SEQUENTIEL

EXPLICATIONS

- Pour la résolution du problème, on utilise un tableau dynamique de booléens car il possède plus de cases qu'un tableau statique et c'est un tableau de booléens car c'est plus simple de manipuler les nombres, on le désalloue à la fin du programme sinon il y aura une fuite de mémoire.
- Pour le chronomètre, nous avons utilisé la classe std::chrono qui permet de gérer le temps
- Dans les schémas, on appellera le tableau « tab »



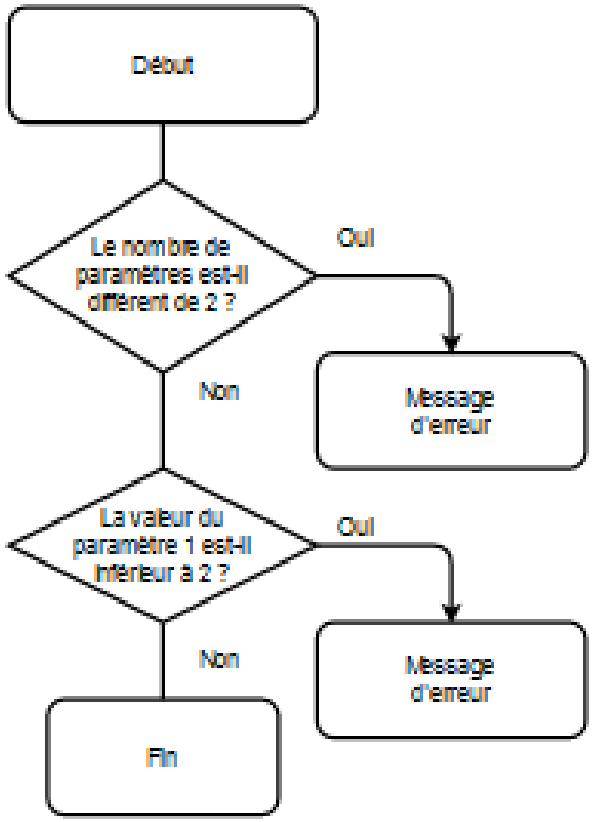
ALGORITHME EN SEQUENTIEL

SOUS–PROGRAMME 1 (TEST DES PARAMETRES)

- Le prototype de ce sous-programme est :
  `void test_param(int, char **)`

- Les arguments sont le nombre de paramètres de la ligne de commande et son contenu, c'est un tableau de pointeur de caractères et on peut convertir une variable de ce type en un entier avec la fonction `atoi`, cela va être utile par la suite

- On génère un message d'erreur avec la classe std::runtime_error qui permet de signaler les erreurs qui sont dues à des événements dépassant le cadre du programme et qui ne peuvent pas être facilement prévues.



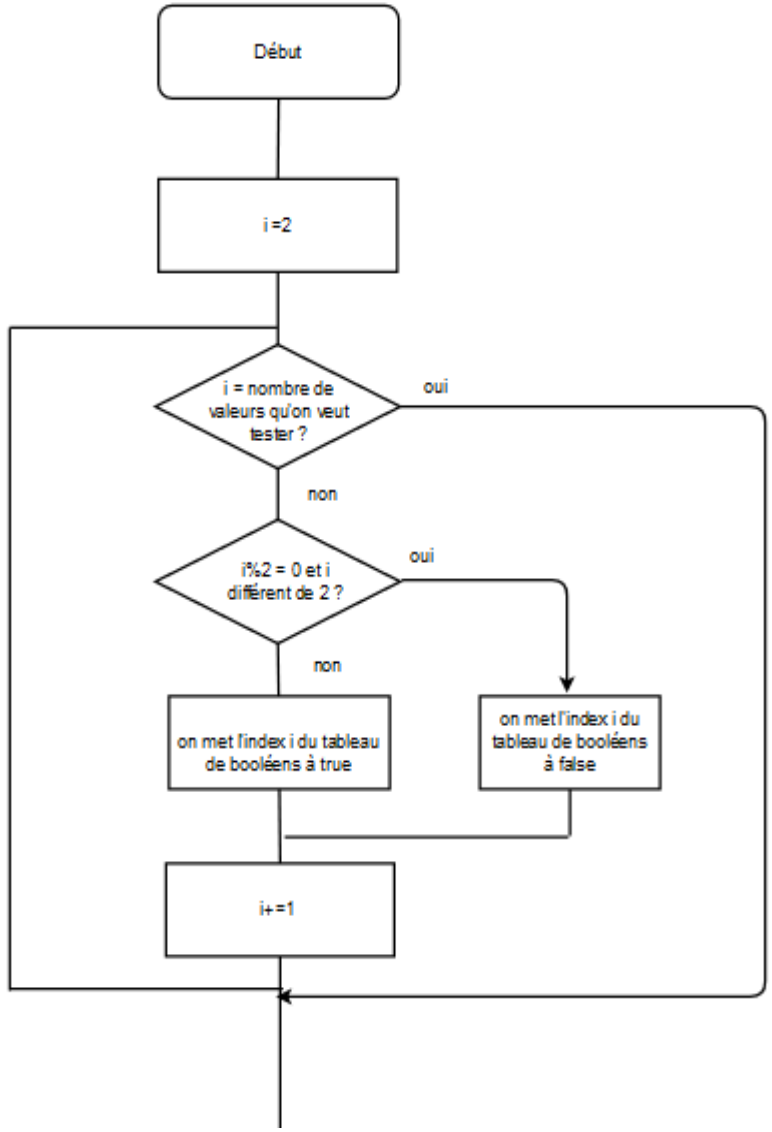
ALGORITHME EN SEQUENTIEL

SOUS–PROGRAMME 2 (PARTIE 1) (RECHERCHE DES NOMBRES PREMIERS)

- Le prototype de ce sous-programme est :
  `void calcul(int, bool *)`

- Les arguments sont le nombre de valeurs à et le tableau dynamique de booléens, il y a un passage de paramètre par adresse.

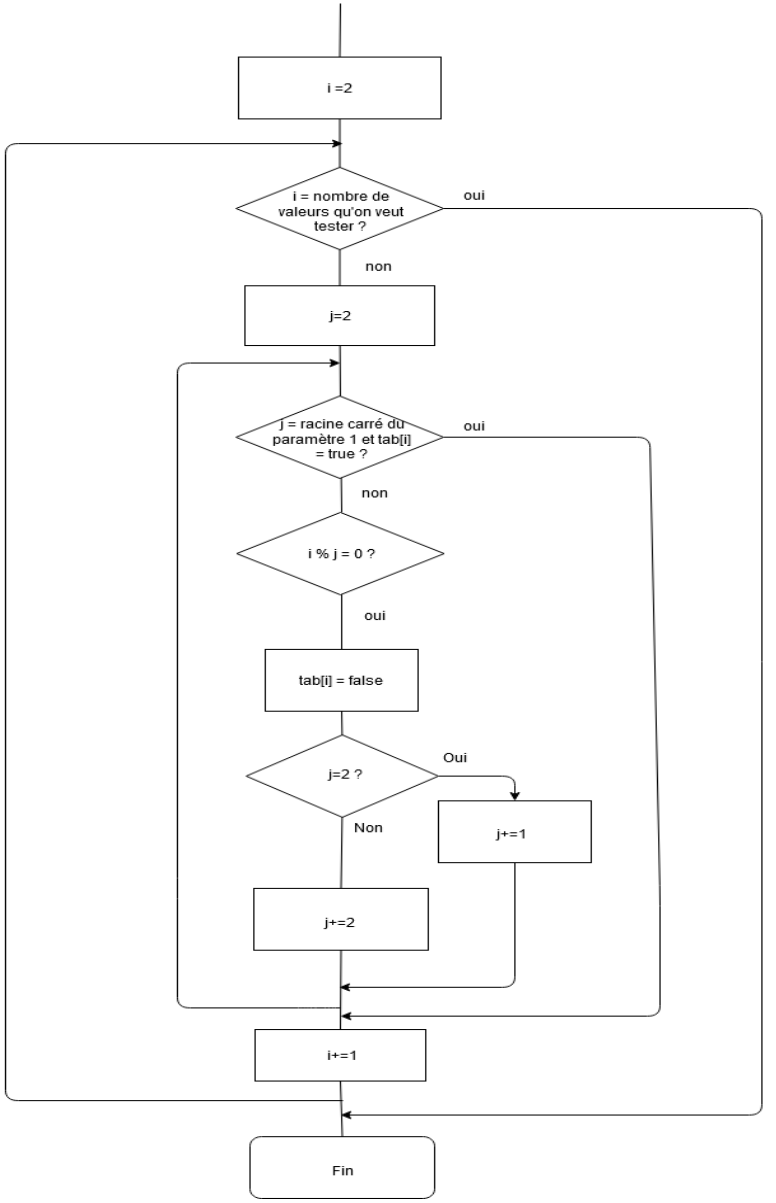
- Dans cette partie, on remplit le tableau avec une boucle for. En effet, si un nombre est pair (nombre%2 = 0) et différent de 2, ce n'est pas un nombre premier, sinon c'en est un. On traduit cela par `false` pour un nombre non premier et `true` pour un nombre premier



ALGORITHME EN SEQUENTIEL

SOUS–PROGRAMME 2 (PARTIE 2) (RECHERCHE DES NOMBRES PREMIERS)

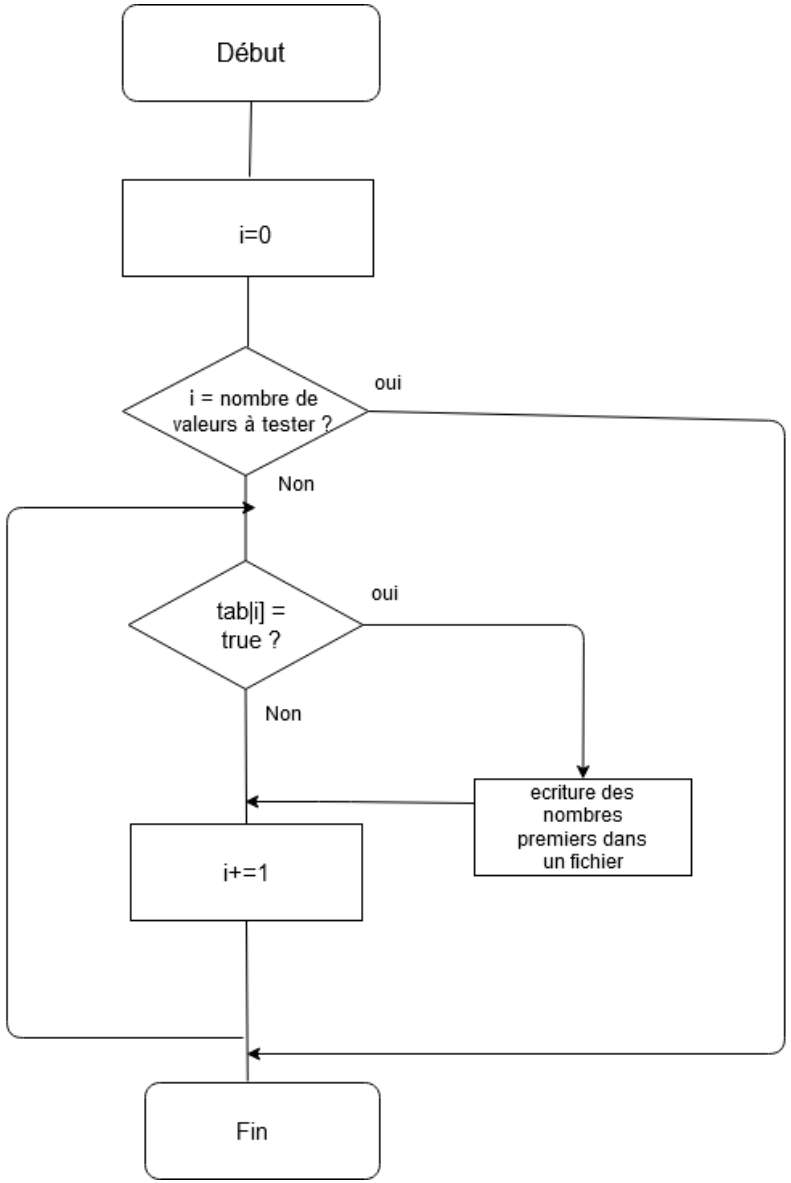
- Dans cette partie, on veut savoir si les nombres sont premiers pour ceux qui ne sont pas encore testés, pour cela, on fait une boucle for imbriqué dans une boucle for.
- Dans la première boucle, la variable itérative est i qui représente un nombre, dans la boucle imbriquée, la variable itérative est j qui représente les potentiels diviseurs de i
- Dans la première boucle, on teste les valeurs de 2 jusqu'à i càd le nombre de valeurs qu'on veut tester, dans la boucle imbriquée, on teste les valeurs de 2 jusqu'à la racine carré de i et on reste dans cette boucle si la case n°i du tableau est toujours à `true` et si elle est à `false`, on sort de cette boucle afin de gagner beaucoup de temps.
- Dans la boucle imbriquée, si j=2 alors j=j+1 sinon j=j+2, c'est aussi pour gagner du temps car un nombre pair est toujours divisible par 2



ALGORITHME EN SEQUENTIEL

SOUS–PROGRAMME 3 (ECRITURE DES NOMBRES PREMIERS DANS UN FICHIER)

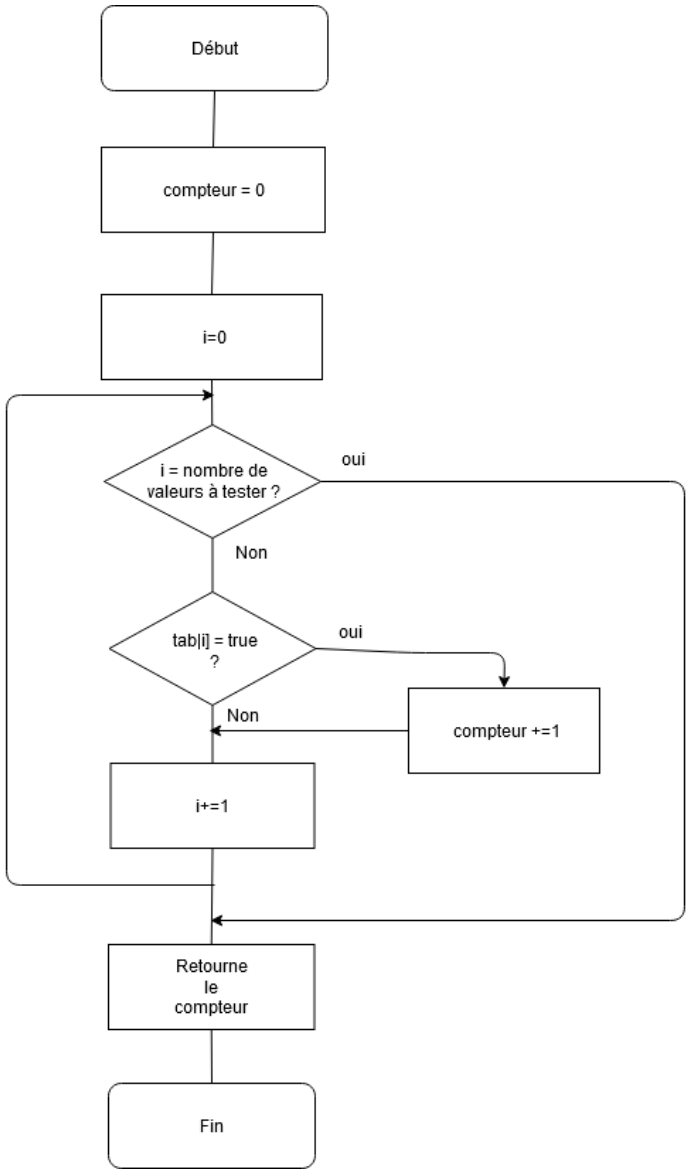
- Le prototype de ce sous-programme est :
  ```
  void ecriture_fichier(int, bool *)
  ```
- Les arguments sont le nombre de valeurs à et le tableau dynamique de booléens, il y a un passage de paramètre par adresse.
- Pour écrire les nombres premiers dans un fichier on utilise la classe std::ofstream qui permet d'écrire dans un fichier, on crée une boucle for qui va de 2 jusqu'au nombre de valeurs qu'on veut tester, dans cette boucle si un nombre est premier càd tab[i] = true, on écrit ce nombre dans un fichier.



ALGORITHME EN SEQUENTIEL

SOUS–PROGRAMME 4 (COMPTEUR DE NOMBRES PREMIERS)

- Le prototype de ce sous-programme est :
  `int nombre_premiers(int, bool *)`
- Les arguments sont le nombre de valeurs à et le tableau dynamique de booléens, il y a un passage de paramètre par adresse.
- Ici, on crée une même boucle for que le sous programme précédent. En effet, au lieu d'écrire dans un fichier, on compte les nombre premiers à l'aide d'un compteur et à la fin, on retourne ce nombre.



PARALLELISATION DE L'ALGORITHME

EXPLICATIONS

- Pour la parallélisation de notre Algorithme, nous avons utilisé OpenMP, c'est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est prise en charge par de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran.
- En effet, pour ce sujet, l'utilisation de OpenMP nous semblait le plus approprié car l'utilisation est simple.

PARALLELISATION DE L'ALGORITHME

UTILISATION

- Pour la parallélisation de notre algorithme, nous avons crée une section du programme parallélisable dans la fonction `void calcul(int, bool *)` à l'aide de `#pragma omp parallel` et peut avoir plusieurs paramètres, ici avec `num_threads`, on peut fixer le nombre maximum de threads lancé par le programme

```
#pragma omp parallel num_threads(atoi(nbthread[2]))
{
```

- Dans cette section, nous parallélisons les boucles for à l'aide de `#pragma omp for` qui fait en sorte que le travail effectué dans une boucle for à l'intérieur d'une zone parallèle soit divisé entre tous les threads

PARALLELISATION DE L'ALGORITHME

UTILISATION

- Cette boucle for est parallélisée grâce à `#pragma omp for`
- Ici le parcours de la boucle est divisé entre les threads c'est-à-dire qu'un thread s'occupe d'une itération, par exemple le thread 3 s'occupe de la 2 ème itération.
- En effet, on peut savoir si on est dans un thread en particulier grâce à :

```
const int thread_id = omp_get_thread_num();
```

- Et on peut l'afficher dans la boucle avec ça par exemple :

```
std::cout << "valeur de i : " << i << " " << "thread n° " << thread_id << std::endl;
```

- Après exécution du programme, avec i <= 5, voici la sortie standard:

```
valeur de i : 5 thread n 3
valeur de i : 3 thread n 1
valeur de i : 2 thread n 0
valeur de i : 4 thread n 2
```

- On peut voir qu'on est dans le thread 3 quand i = 5 .

```
#pragma omp for
for (int i = 2; i<=max; ++i)
{
    if ((i%2 == 0) && (i!=2))
        tab[i] = false;
    else
        tab[i] = true;
}
```

PARALLELISATION DE L'ALGORITHME

UTILISATION

- La boucle for ci-dessous est aussi parallélisée sauf qu'une clause est ajoutée, c'est la clause « schedule », ça permet de configurer la répartition les tâches entre les différents threads
- Avec la répartition dynamique : chaque thread traite une quantité de données spécifiée par la taille passée après dans la déclaration dynamic. Dès qu'un thread a terminé son lot de données, il peut reprendre un lot de données à traiter.

```
#pragma omp for schedule(dynamic)
for (int i = 2; i <= max; i++)
{
    for (int j = 2; j <= sqrt(i) && tab[i]; j += ( j == 2) ? 1:2)
    {
        if (i%j == 0)
            tab[i] = false;
    }

}
```

PARALLELISATION DE L'ALGORITHME

UTILISATION

- A la fin de la section parallèle du programme, on ajoute une barrière de synchronisation avec `#pragma omp barrier`
- Cela permet de faire en sorte que les threads d'une section parallèle ne s'exécuteront pas au-delà de la barrière jusqu'à ce que tous les autres threads aient terminé toutes les tâches dans la section.

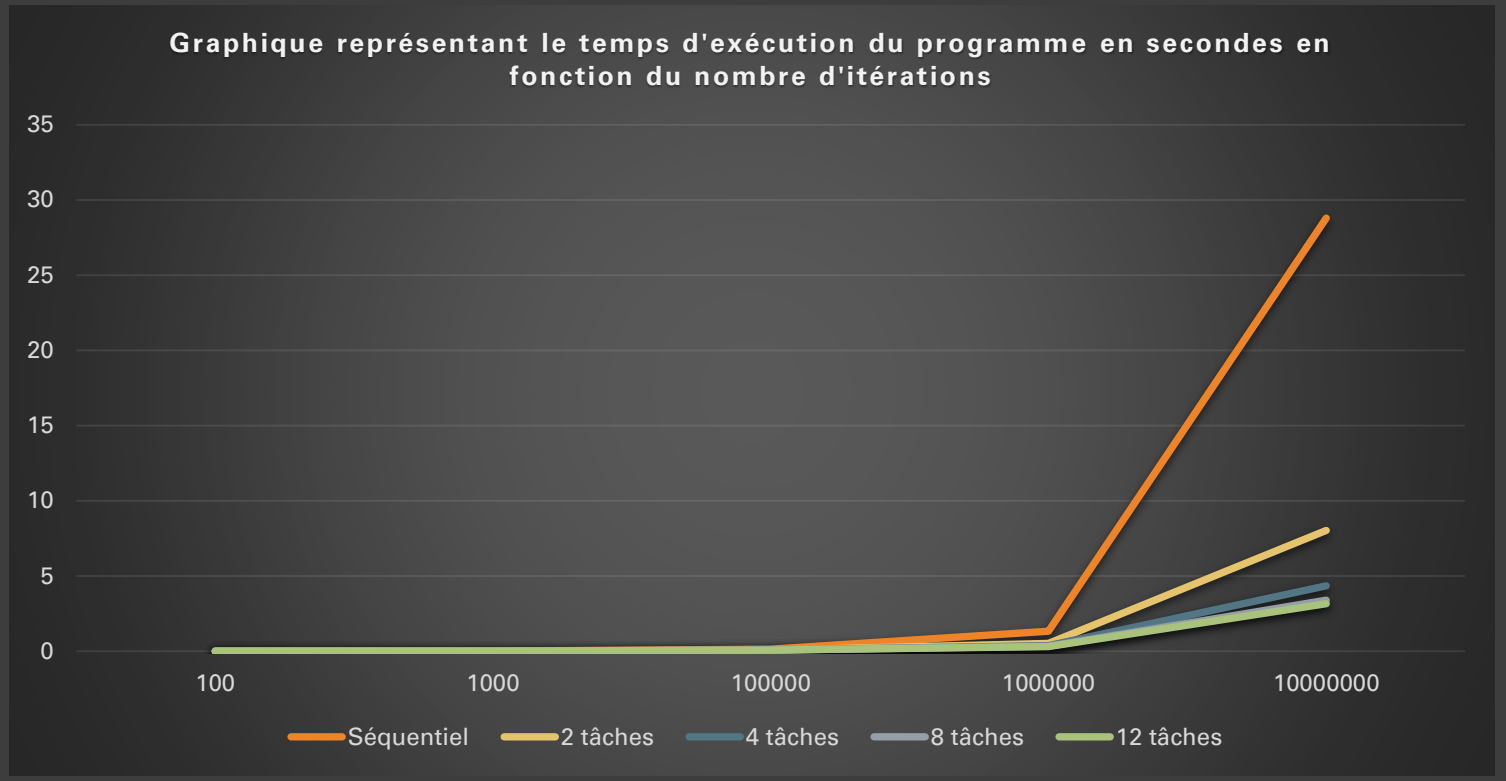
TEMPS D'EXECUTION DE L'ALGORITHME (EN SECONDES)

TABLEAU

itérations / séquentiel /multithread	100	1000	100000	1000000	10000000
Séquentiel	0,003951	0,007974	0,153581	1,34122	28,7837
2 tâches	0,004889	0,002991	0,08944	0,527455	8,03748
4 tâches	0,00802	0,007944	0,081813	0,381761	4,36492
8 tâches	0,011388	0,009964	0,080395	0,335951	3,40018
12 tâches	0,011966	0,009937	0,080782	0,320135	3,17486

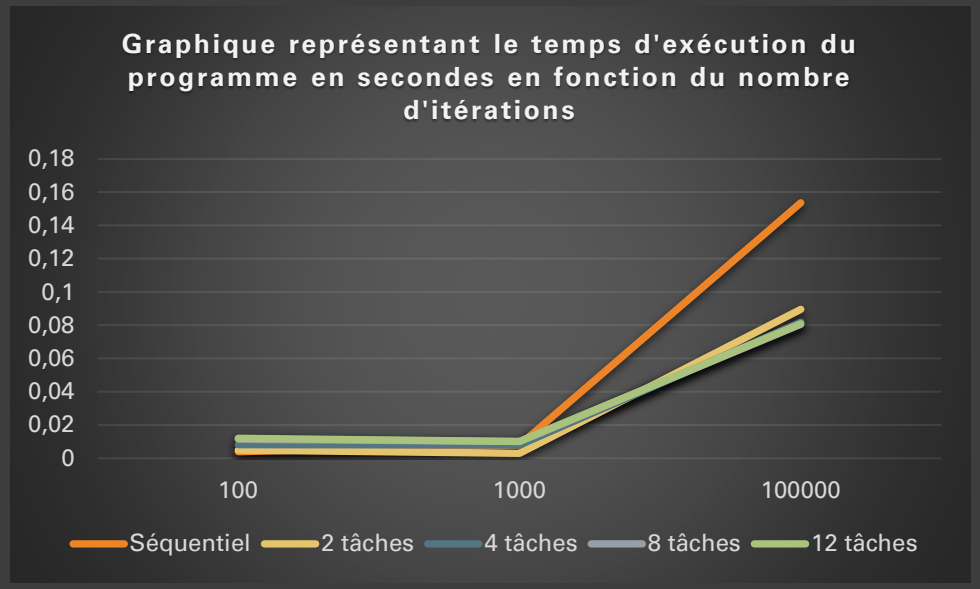
TEMPS D'EXECUTION DE L'ALGORITHME (EN SECONDES)

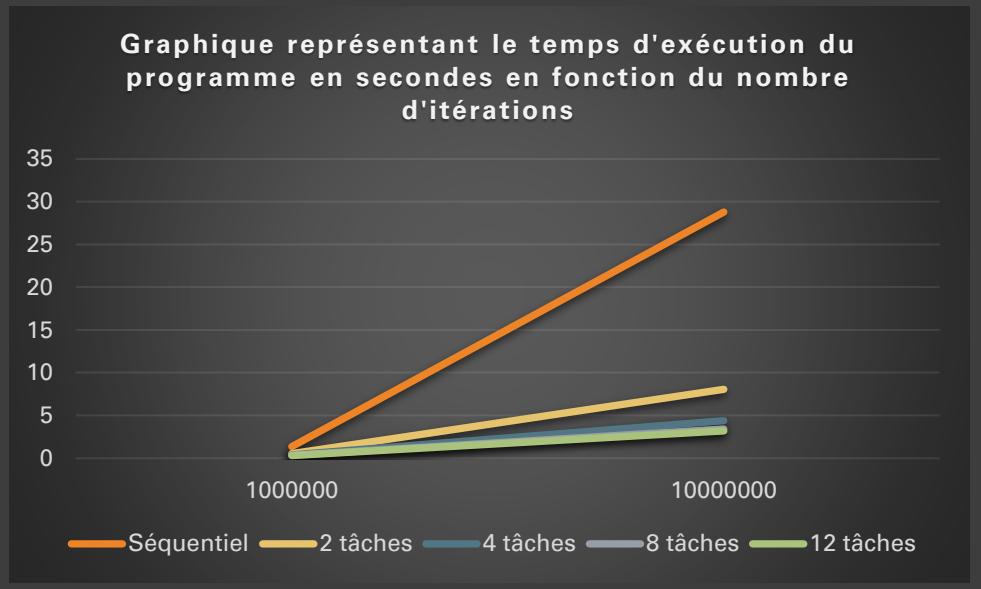
GRAPHIQUE AVEC TOUTES LES ITERATIONS



TEMPS D'EXECUTION DE L'ALGORITHME (EN SECONDES)

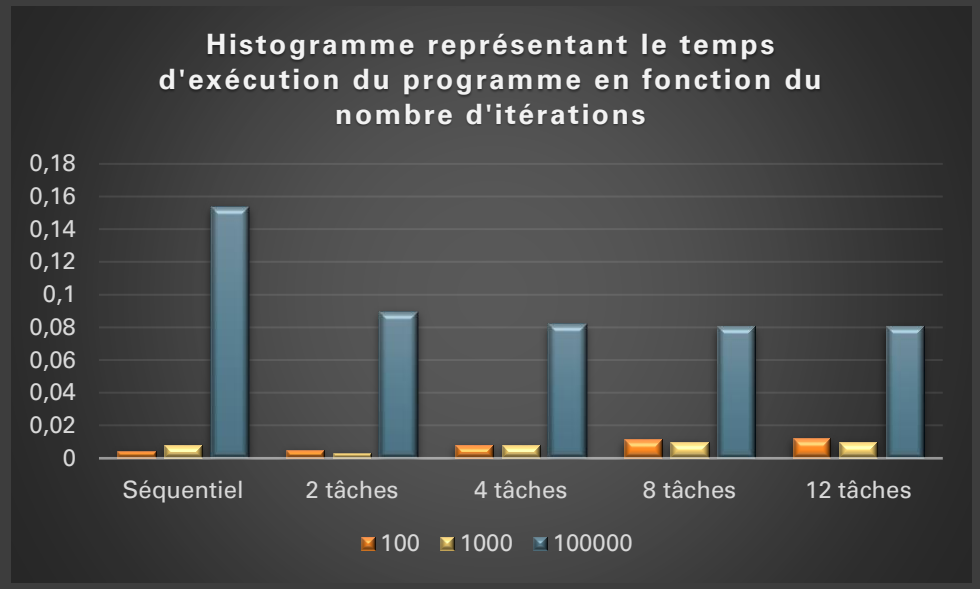
GRAPHIQUE SEPARES EN DEUX POUR VOIR MIEUX LES RESULTATS

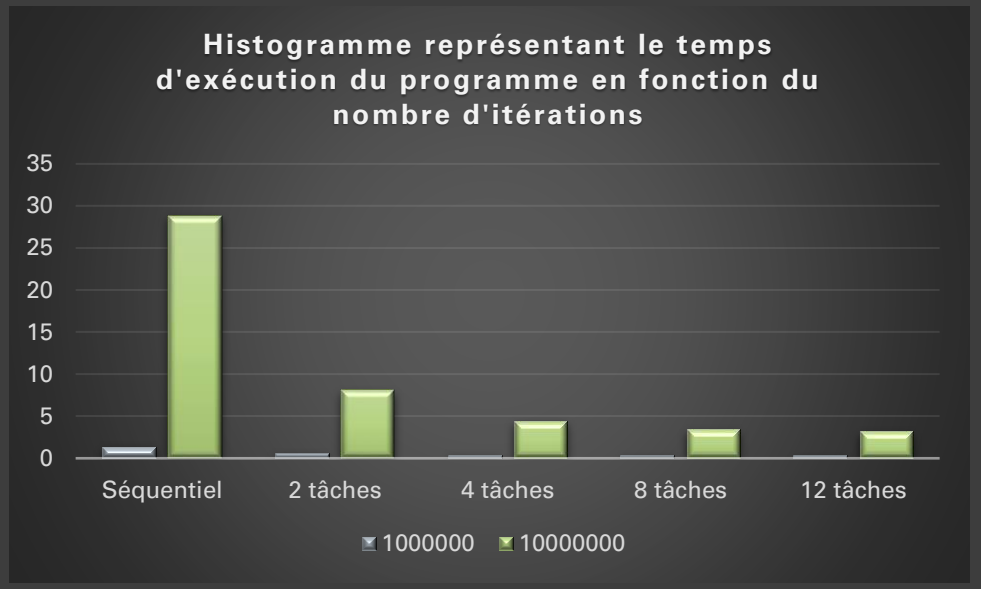




TEMPS D'EXECUTION DE L'ALGORITHME (EN SECONDES)

HISTOGRAMME SEPARES EN DEUX POUR VOIR MIEUX LES RESULTATS





TEMPS D'EXECUTION DE L'ALGORITHME (EN SECONDES)

INTERPRETATION DES RESULTATS

- Graphiquement et selon le tableau, on peut voir que le programme en séquentiel met beaucoup de temps par rapport au programme parallélisé. En effet, pour 10 millions d'itérations, le programme en séquentiel met environ 10 fois plus de temps que le programme avec 12 threads et sans surprise, ce dernier est le plus rapide de tous.
- On peut dire que plus il y a de threads et plus le programme est rapide. Cependant, au-delà de 12 threads, le temps stagne, c'est parce que l'ordinateur de test possède un processeur de 6 cœurs et 12 threads.
- Aussi, quand il faut tester avec de petites itérations, 100 par exemple, c'est le programme en séquentiel le plus rapide, cela peut s'expliquer, il faut bien stocker les valeurs dans le fichier donc ça prend du temps donc parfois avec 1 cœur c'est plus pertinent puisqu'il peut enchainer le stockage des valeurs sans attendre des threads.

CONCLUSION

- Pour conclure, on peut dire que l'utilisation de OpenMP pour la parallélisation des programmes est simple, cependant il y a des limites, en effet, les opérations pour la condition de la boucle for sont : == , >, <, >=, <=, on ne peut pas mettre d'opérateurs logique comme && mais aussi ||, on ne peut pas aussi mettre des fonctions retournant un type tels que sqrt() par exemple. C'est valable pour l'incrémentation de la boucle, on ne peut mettre que des itérations tels que i++, i+=2 ,i+=3 etc…
- Aussi, la machine de test est performante ce qui permet de faire des tests pertinents.
- Aussi, notre programme n'est pas si optimisé que ça contrairement à un programme s'inspirant de la crible d'Erastothène par exemple, le temps d'exécution est environ des centaine de fois supérieur que notre programme. Cependant, un programme de ce genre est difficilement parallélisable à cause de sa très forte optimisation.