Projet Système - Fréquences dans des dictionnaires

Sujet :

À partir de plusieurs dictionnaires utilisant l'alphabet latin, déterminer pour chacun les fréquences des lettres "a, b, c, ..., z", des digrammes "aa, ab, ac, ..., ba, bb, bc, ..., zz", et la même chose pour les trigrammes "aaa, aab, aac, ... aba, abc, ... baa, bab, …, zzz". Les résultats sont stockés dans un fichier unique.

Documentations utilisées :

http://www.cplusplus.com
https://en.cppreference.com

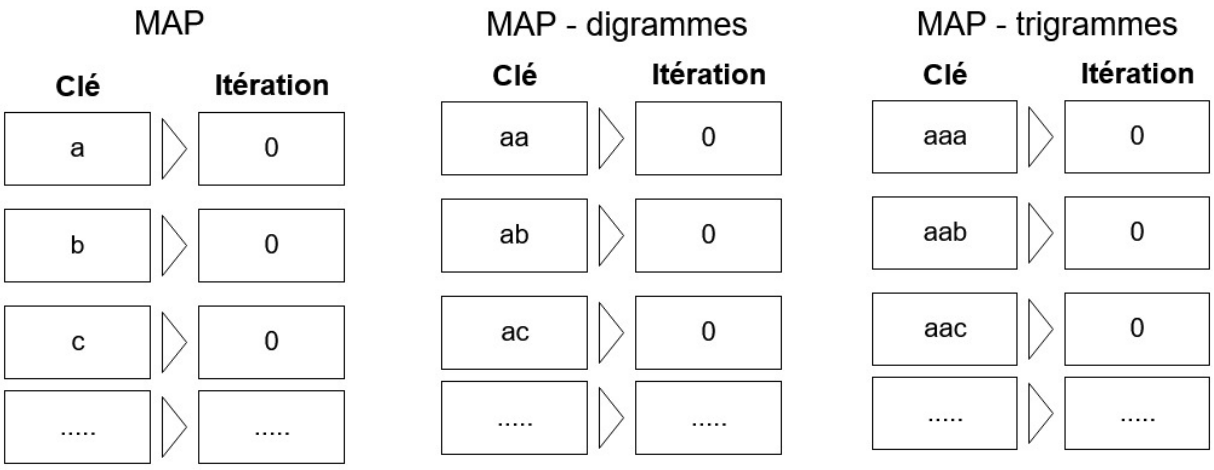
Explication des algorithmes

Algorithme 1

FreqAnalysis-sequentiel.cpp déroulement du programme :

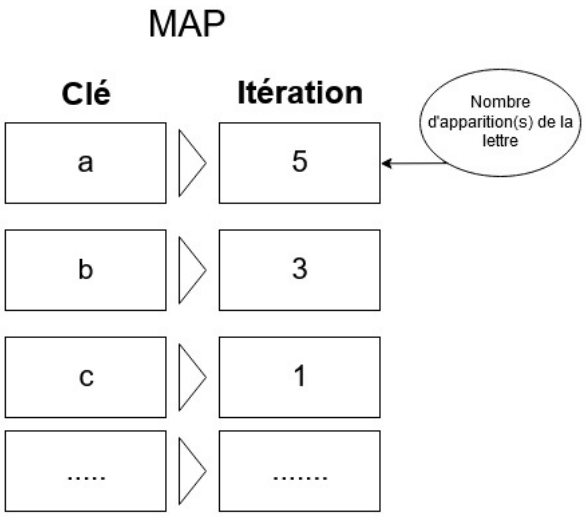
Initialisation :

Pour analyser la fréquence d'apparition des lettres on utilise une map contenant un string pour la lettre et un float pour la fréquence d'apparition de celle-ci. On initialise d'abord la map avec toutes les lettres de l'alphabet et leurs fréquences à 0. On fait de même pour les digrammes et les trigrammes : on initialise avec toutes les possibilités à 0.



Analyse :

Pour chaque objet Analyse on ouvre le fichier concerné. Puis on lit caractère par caractère tant qu'il y en a. À chaque graphène trouvé, on incrémente son nombre d'apparitions à l'aide de la fonction incGraphene(). Cette fonction de la classe Analyse va incrémenter la partie itération de la map correspondant au graphène lu. Et on incrémente le nombre de graphènes total trouvés. Donc on a d'abord l'analyse des lettres, puis des digrammes et enfin des trigrammes. Ce qui veut dire trois lectures du fichier

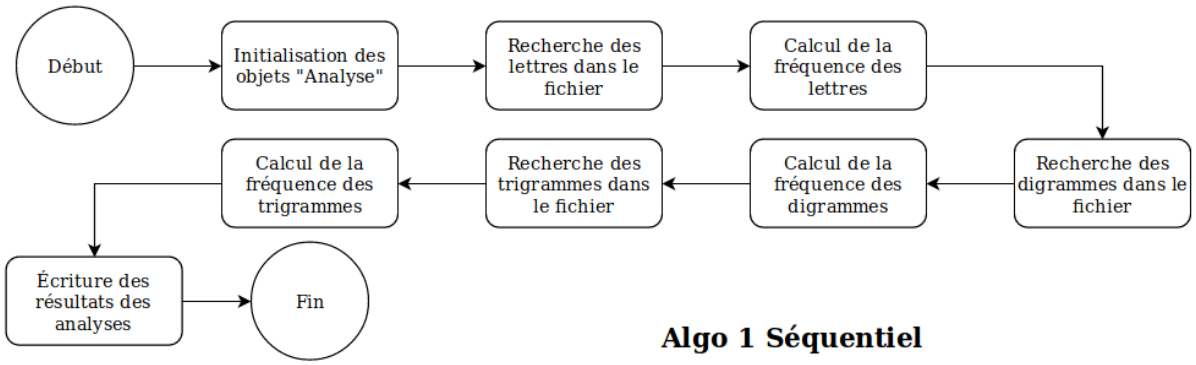


Calcul et écriture des fréquences :

Via la méthode calcFreq() de la classe Analyse on calcule la fréquence de chaque lettre en divisant le nombre d'apparitions (contenu dans la map) par le nombre de lettres total (donnée membre de la classe). Et on écrase l'itération de la lettre par sa fréquence dans la map. Et on fait de même pour les digrammes et trigrammes.

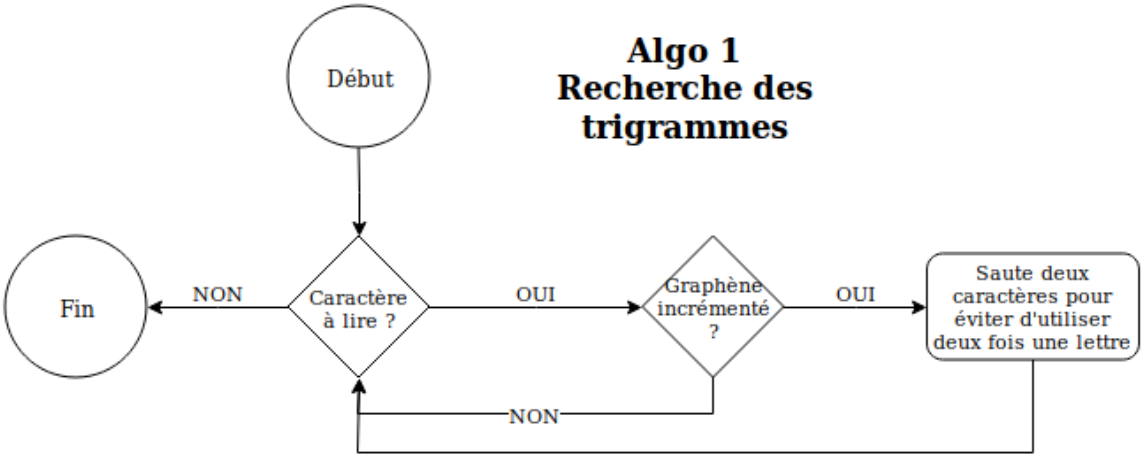
On écrit dans un fichier de sortie le résultat du calcul par la fonction printAnalyse().

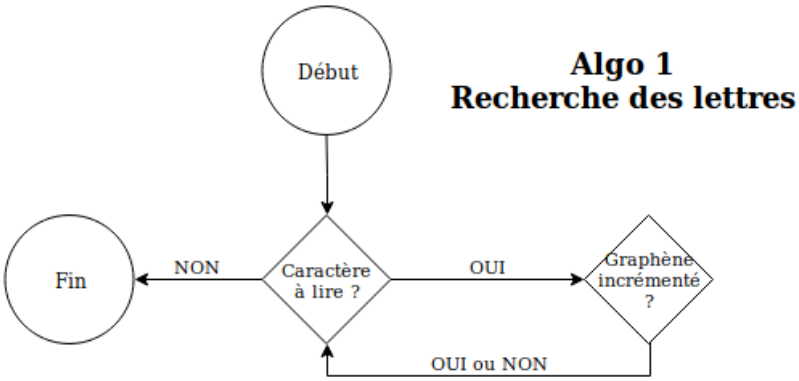
Diagramme du fonctionnement simplifié :

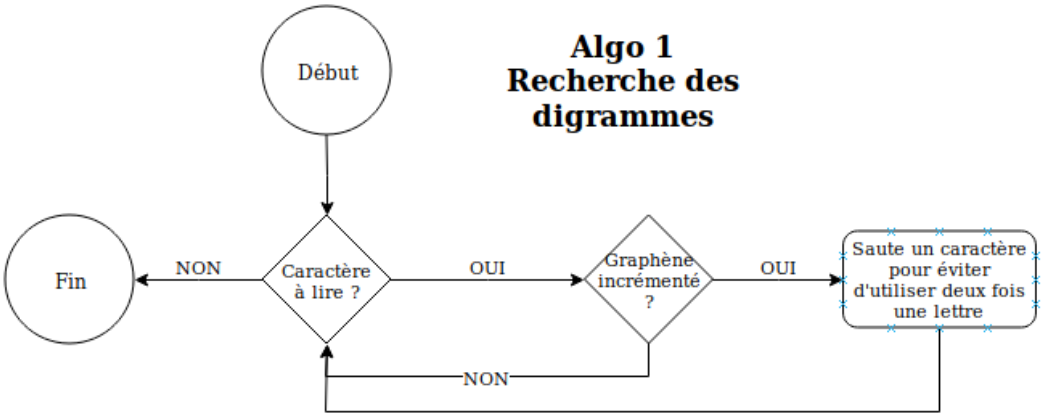


Algo 1 Séquentiel

Fonction de recherche :







FreqAnalysis-pthread.cpp déroulement du programme :

Initialisation :
Même chose que **FreqAnalysis-sequentiel.cpp**

Analyse :
On utilise trois threads, un pour la fréquence des lettres, un pour la fréquence des digrammes, un pour la fréquence des trigrammes.
Les trois threads sont lancés en parallèle

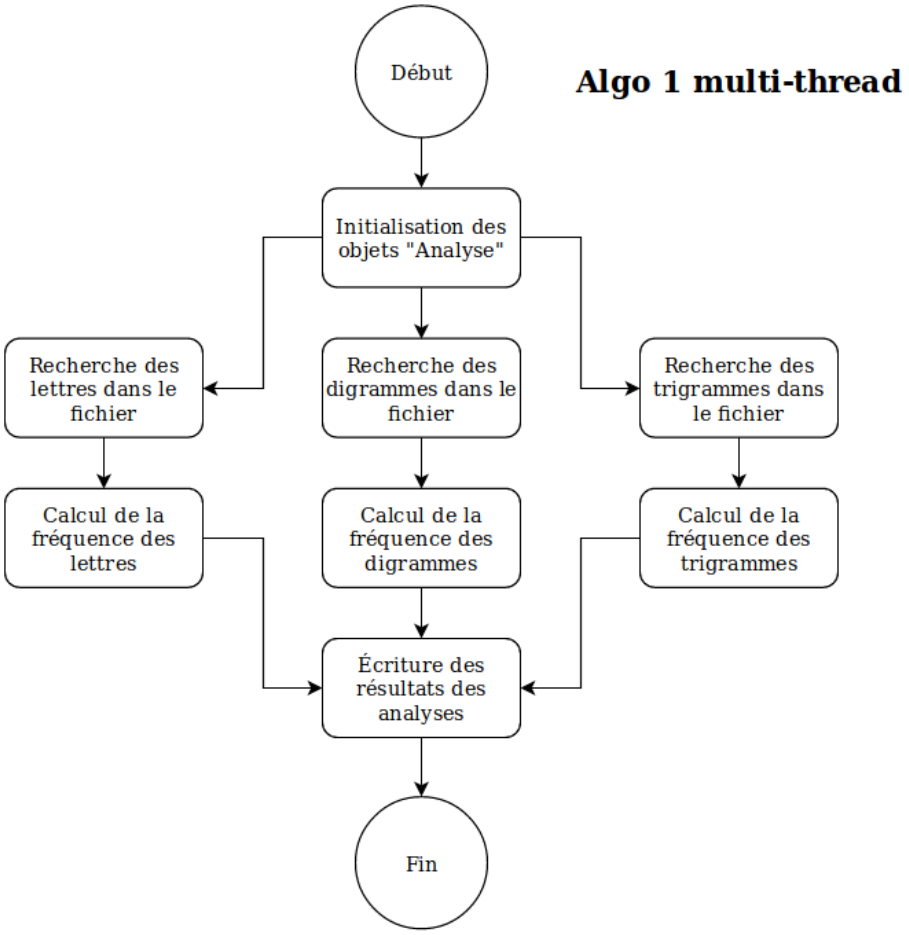
Calcul et écriture des fréquences :

La méthode calcFreq() de la classe Analyse calcule la fréquence de chaque graphène en divisant le nombre d'apparitions (contenu dans la map) par le nombre de graphènes total (donnée membre de la classe). Et on écrase l'itération du graphène par sa fréquence dans la map.

Lorsque le premier thread (analyse des lettres) se termine, on lance la fonction calcFreq() qui lance le calcul de fréquence de chaque lettre. Lorsque le calcul est fini on écrit les résultats dans le fichier de sortie pour pas perdre de temps. De même lorsque l'analyse des digrammes et de trigrammes finissent.

Pour écrire on utilise la fonction printAnalyse().

Diagramme du fonctionnement simplifié :



Fonction de recherche :

Identique au programme séquentiel

Algorithme 2

FreqAnalysis-sequentiel.cpp déroulement du programme :

Initialisation :

Pour analyser la fréquence d'apparition des lettres on utilise une map contenant un string pour la lettre et un float pour la fréquence d'apparition de celle-ci. On initialise d'abord la map avec toutes les lettres de l'alphabet et leurs fréquences à 0. On fait de même pour les digrammes et les trigrammes : on initialise avec toutes les possibilités à 0. Pour le fichier on l'ouvre et on récupère le buffer.

Analyse :

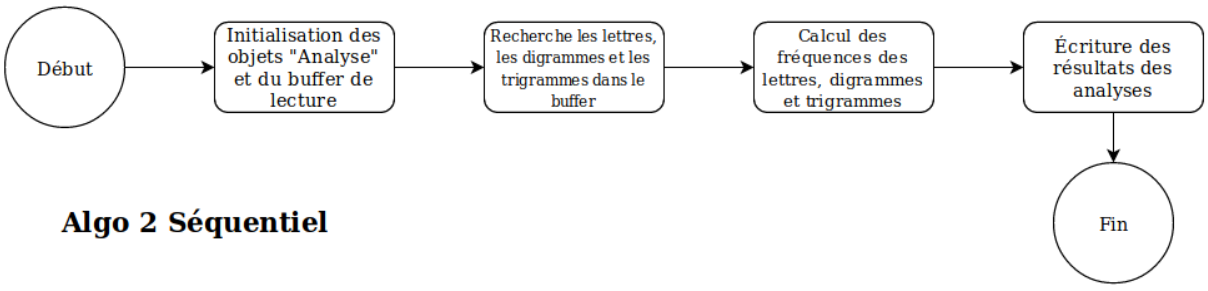
Pour faire l'analyse on récupère le buffer et on lit caractère par caractère tant qu'il y en a. Envoie du caractère dans la fonction incGraphène de l'analyse des lettres. Ajout du caractère dans une chaine de caractères temporaire du digramme et du trigrammes. Si la chaine de caractères digramme contient deux caractères alors on l'envoie dans la fonction incGraphène() de l'analyse de digrammes. De même avec les trigrammes pour 3 caractères.
La fonction incGraphène() renvoie vrai s'il a réussi à incrémenter un graphène (ce qui veux dire que l'on veut la fréquence de ce graphène), faux sinon.
Si incGraphène() de l'analyse des digrammes renvoie faux alors on supprime le premier caractère de la chaine de caractère de digrammes. De même pour les trigrammes.

Calcul et écriture des fréquences :

La méthode calcFreq() de la classe Analyse calcule la fréquence de chaque graphène en divisant le nombre d'apparitions (contenu dans la map) par le nombre de graphènes total (donnée membre de la classe). Et on écrase l'itération du graphène par sa fréquence dans la map.

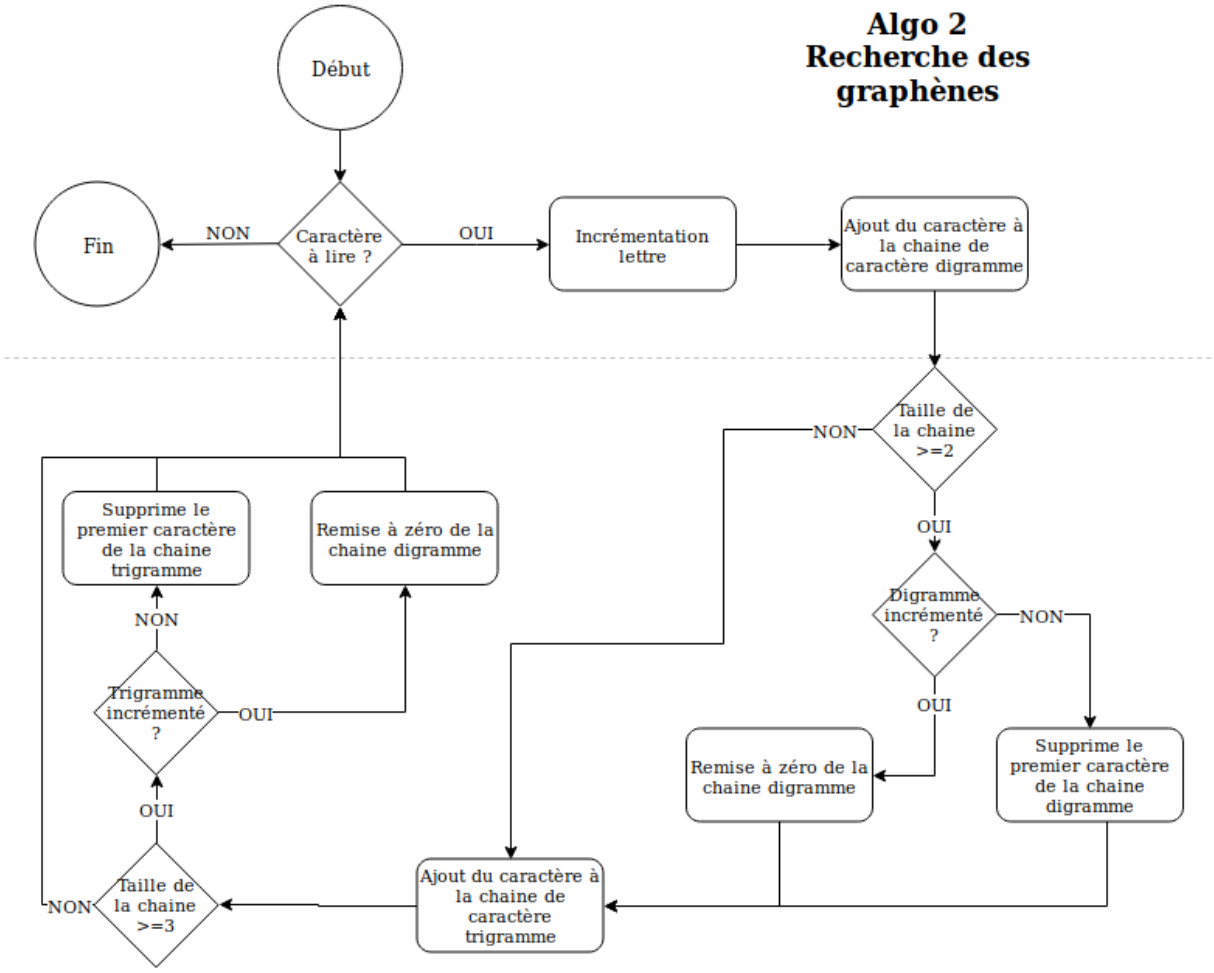
Donc lorsque l'analyse est terminée on lance le calcul des fréquences de toutes les analyses et on les écrit dans le fichier de sortie.

Diagramme du fonctionnement simplifié :



Algo 2 Séquentiel

Fonction de recherche :



FreqAnalysis-pthread.cpp déroulement du programme :

Initialisation :

Pour analyser la fréquence d'apparition des lettres on utilise une map contenant un string pour la lettre et un float pour la fréquence d'apparition de celle-ci. On initialise d'abord la map avec toutes les lettres de l'alphabet et leurs fréquences à 0. On fait de même pour les digrammes et les trigrammes : on initialise avec toutes les possibilités à 0. Pour le fichier on l'ouvre et on récupère le buffer. Ce buffer on va ensuite le divisé par le nombre de threads N (4 dans le projet). On aura donc N buffers avec chacun une partie du fichier de base.

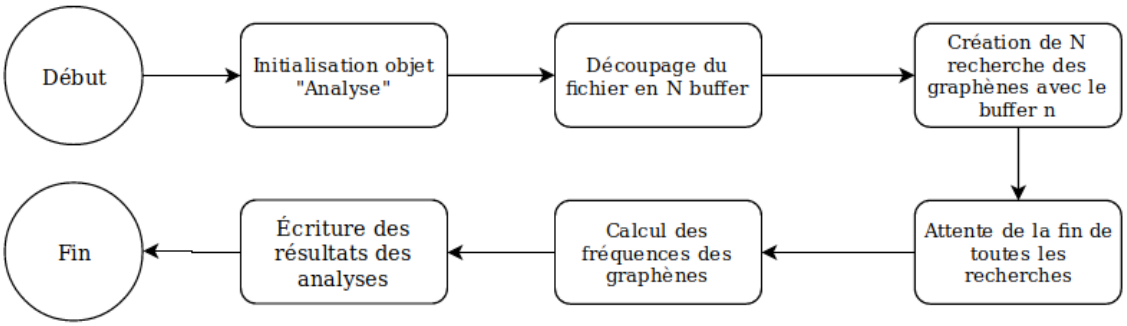
Analyse :

On utilise N threads, avec chacun un bout du fichier de base sous forme de buffer. Sachant que chaque thread partage les analyses il nous faut un sémaphore pour bloquer l'accès en écriture pour l'incrémentation des graphènes. Pour chaque thread le fonctionnement d'analyse reste le même que l'algo2 du programme séquentiel.

Calcul et écriture des fréquences :

Lorsque tous les threads sont terminés, calcul des fréquences de toutes les analyses. Puis on écrit les résultats dans le fichier de sortie.

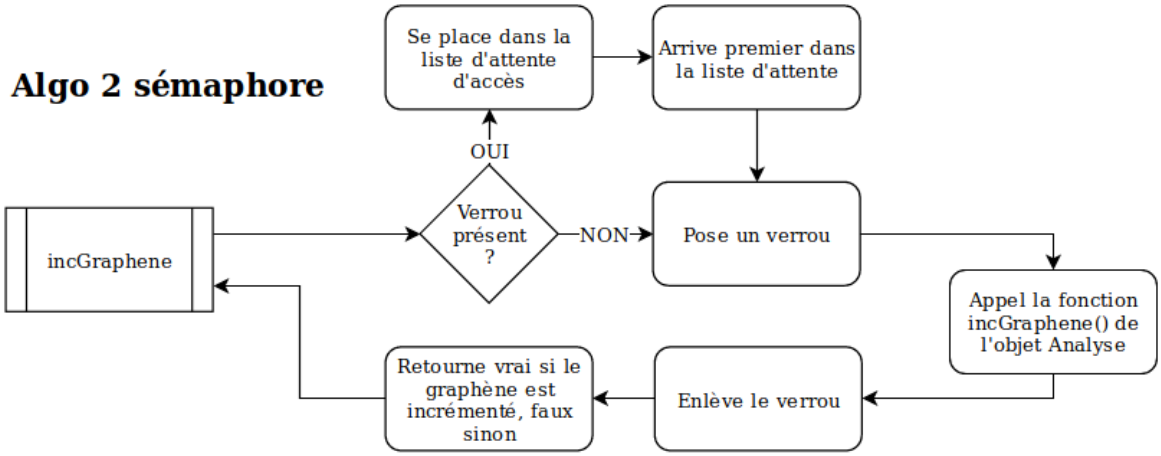
Diagramme du fonctionnement simplifié :



Algo 2 multi-thread

Fonction de recherche :

C'est la même chose que le séquentiel à l'exception que chaque incrémentation d'un graphène, un verrou est posé sur l'objet « Analyse » concerné.



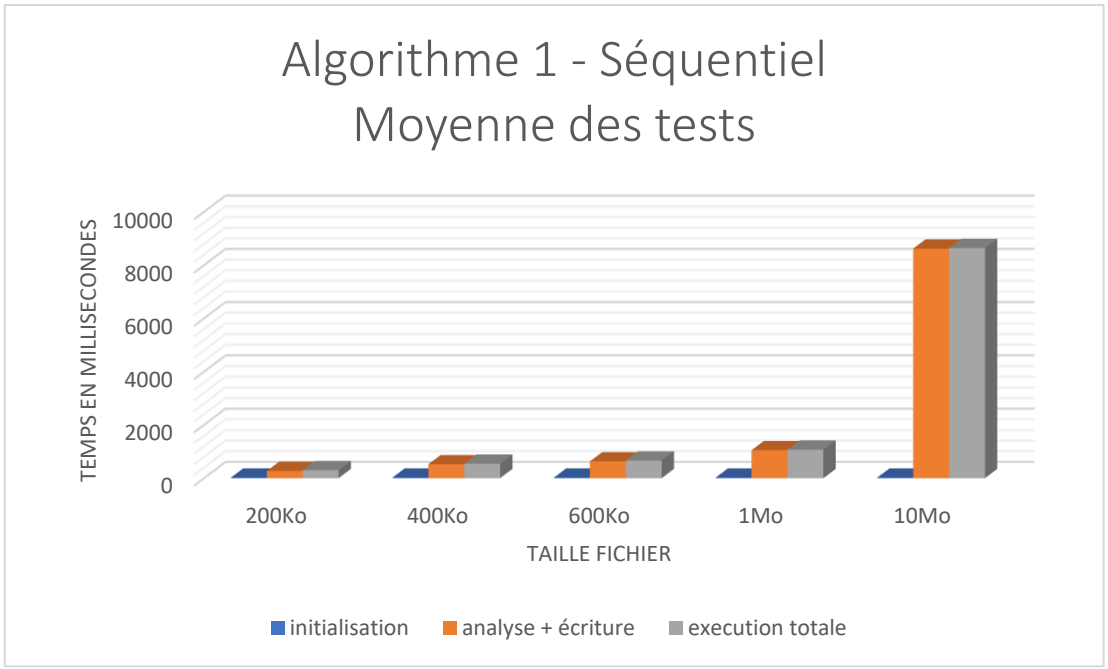
Test des programmes

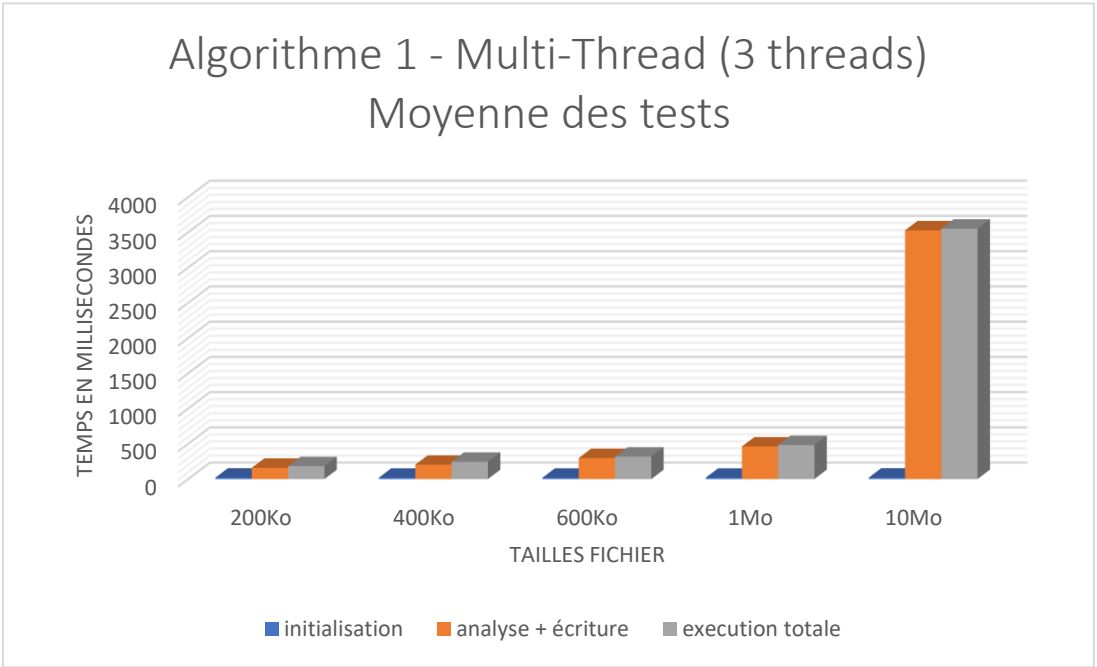
Machines de test :

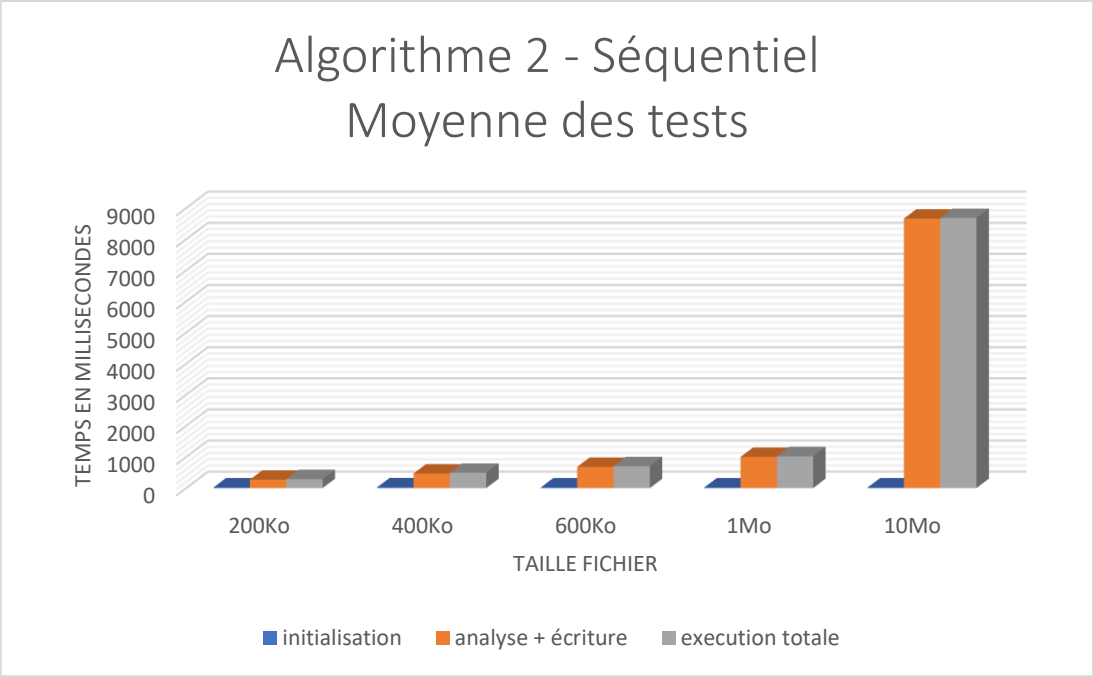
Machine 1 :
- Type machine :Laptop
- Système d'exploitation : Ubuntu 19.04
- Processeur : Intel(R) Core(TM) i7-8750H
- Fréquence processeur : Base : 2.20Ghz ; Turbo : 4.10 GHz
- Nombre Cœurs : 6
- Nombre de Threads : 12
- Cache L1/L2/L3 : 384KiB / 1536KiB / 9MiB
- RAM : 8034292 kB
- Stockage : SSD NvME M.2

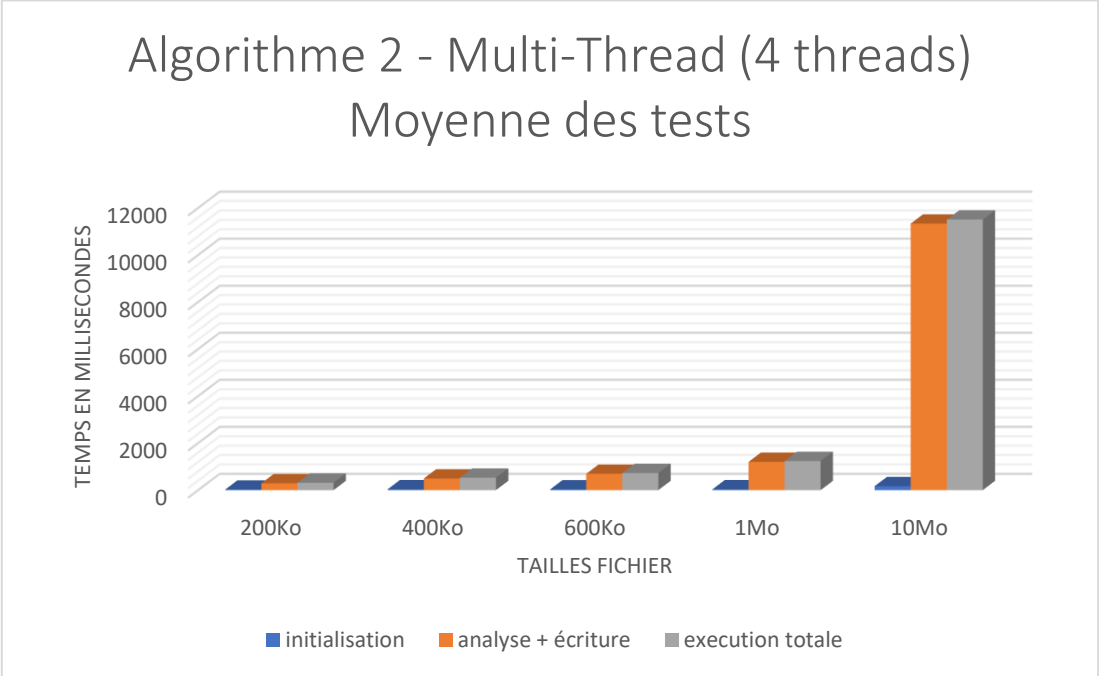
Machine 2 :
- Type machine : Laptop
- Système d'exploitation : Ubuntu 19.04
- Processeur : Intel(R) Core(TM) i7-8550U
- Fréquence processeur : Base : 1.80Ghz ; Turbo : 4.00 GHz
- Nombre Cœurs : 4
- Nombre de Threads : 8
- Cache L1/L2/L3 : 256KiB / 1MiB / 8MiB
- RAM : 8048780 kB
- Stockage : HHD 5400 tours/min

Graphiques :


Algorithme 1 - Séquentiel
Moyenne des tests
TEMPS EN MILLISECONDES
10000
8000
6000
4000
2000
0
200Ko
400Ko
600Ko
1Mo
10Mo
TAILLE FICHIER
initialisation
analyse + écriture
execution totale


Algorithme 1 - Multi-Thread (3 threads)
Moyenne des tests
TEMPS EN MILLISECONDES
4000
3500
3000
2500
2000
1500
1000
500
0
200Ko
400Ko
600Ko
1Mo
10Mo
TAILLES FICHIER
initialisation
analyse + écriture
execution totale





On remarque que pour l'algorithme 1 la version multi-thread améliore le temps de calcul, on a quasiment à chaque fois une division du temps par 2.
Tandis que pour l'algo 2 on remarque que la version multi-thread n'est pas du tout une bonne idée car il est plus long que le séquentiel à cause du verrou.
Ces histogrammes sont construits à l'aide de la moyenne des résultats des deux machines. En voyant les valeurs précises on remarque que l'ouverture est plus rapide sur la machine grâce au SSD. Évidemment la fréquence plus haute de la machine 1 améliore les résultats de temps.

Pour conclure l'algorithme 1, malgré l'obligation de lire trois fois le fichier, atteint à peu près le même niveau de performance que l'algo 2 qui ne le lit qu'une fois. C'est au niveau du multi-thread que l'algo 1 fonctionne le mieux. En effet l'algo 2 impose l'ajout d'un verrou parce que les threads partagent de la mémoire.