

Mini Projet de Module Programmation Parallèle

L'implémentation parallèle de l'algorithme de classification K-means avec
OpenMP

Réalisation par :

Lahlimi Oussama

Encadrée par :

Mm.Azzouzi Salma

2022/2023

Table des matières

Résumé	3
I. Introduction	3
II. OpenMP et les concepts de programmation parallèle.	4
1. Points importants à dans l'implémentation d'un programme en utilisant des threads	5
III. K-means:	6
1. Objectif de K-means	7
2. Processus itératif	8
3. Convergence	8
4. Choix du nombre de clusters K	8
IV. Parallélisation de l'algorithme K-means avec OpenMP	9
V. Implémentation et explication de code	11
VI. Analyse de Résultat de notre code d'exécution	13
1. Threads et Itérations en Parallèle	14
2. Centres Finaux de Clusters	14
3. Interprétation du Résultat	15
4. Intérêt du Parallélisme	16
5. La comparaison entre l'exécution séquentielle et parallèle de l'algorithme K-means	16
Références	19

Résumé

Ce projet présente une (Implémentation parallèle (Multithreading) de l'algorithme de classification K-means afin de trouver les positions finales des centroïdes pour chaque cluster, avec OpenMP dans le langage de programmation C, en plus la comparaison entre une implémentation séquentielle et parallèle).

I. Introduction

L'objectif de ce projet est d'implémenter l'algorithme de classification K-means en utilisant OpenMP pour la parallélisation des calculs en langage C. L'algorithme K-means est une technique de regroupement de données largement utilisée pour diviser un ensemble de données en clusters homogènes. Le projet se concentre sur la parallélisation des deux étapes essentielles de l'algorithme : l'assignation des points aux clusters et la mise à jour des centroïdes.

L'assignation des points aux clusters consiste à attribuer chaque point de données au cluster le plus proche en termes de distance euclidienne. La mise à jour des centroïdes consiste à recalculer les positions des centroïdes en fonction des points attribués à chaque cluster.

L'utilisation d'OpenMP permet d'accélérer le traitement en parallèle, ce qui peut être particulièrement efficace pour des ensembles de données volumineux. Le projet vise à trouver les positions finales des centroïdes après un certain nombre d'itérations défini par l'utilisateur, ce qui est essentiel pour la classification des données.

L'implémentation parallèle de l'algorithme de classification K-means avec OpenMP est un exemple de classification "**SIMD**" (Single Instruction, Multiple Data) ou "**SIMT**" (Single Instruction, Multiple Threads) en fonction de la terminologie utilisée.

Cela permet d'exploiter efficacement les capacités de calcul parallèle des processeurs multi-cœurs pour accélérer le processus de classification K-means.

II. OpenMP et les concepts de programmation parallèle.

OpenMP (Open Multi-Processing) est une API (Application Programming Interface) qui permet la programmation parallèle en C, C++, et Fortran. Elle facilite le développement de programmes parallèles en utilisant la parallélisation de boucles, de sections de code, et d'autres structures parallèles. OpenMP est largement utilisé pour exploiter efficacement les architectures multicœurs et multiprocesseurs.



Voici quelques concepts clés de la programmation parallèle avec OpenMP :

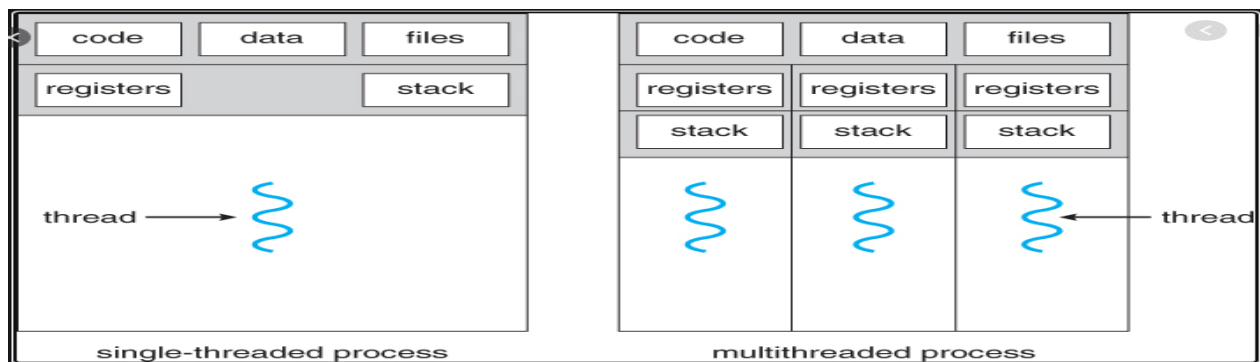
- **Threads**: OpenMP repose sur la création de threads pour exécuter des parties de code en parallèle. Un thread est une unité de traitement légère qui peut s'exécuter en parallèle avec d'autres threads. Ces threads sont souvent créés automatiquement par OpenMP.
- **Directives OpenMP** : Les directives OpenMP sont des commentaires spéciaux qui sont interprétés par le compilateur. Elles indiquent les parties du code qui peuvent être exécutées en parallèle. Par exemple, la directive `#pragma omp parallel` crée un bloc de code parallèle.
- **Régions parallèles** : Une région parallèle est une partie du code qui est exécutée en parallèle. Les threads sont créés pour exécuter cette région et le nombre de threads peut être contrôlé via des directives OpenMP.
- **Partage de données** : Lorsque vous travaillez en parallèle, il est souvent nécessaire de partager des données entre les threads. OpenMP fournit des mécanismes pour gérer cela, notamment la clause `shared` pour partager des données entre les threads et la clause `private` pour créer des copies privées de données pour chaque thread.
- **Synchronisation** : Les threads parallèles ont besoin de se synchroniser à certains moments pour éviter des problèmes comme les conditions de course. OpenMP propose des mécanismes de synchronisation tels que la directive `#pragma omp barrier`.
- **Itérations parallèles** : OpenMP facilite la parallélisation de boucles en utilisant des directives telles que `#pragma omp for`. Cela permet d'itérer sur un ensemble de données en parallèle.
- **Environnement d'exécution** : OpenMP gère l'environnement d'exécution parallèle, y compris la création et la gestion des threads, l'ordonnancement des tâches, etc.

- **Compilateur OpenMP** : Pour utiliser OpenMP, vous avez besoin d'un compilateur compatible OpenMP. La plupart des compilateurs modernes prennent en charge OpenMP, et vous devrez peut-être activer la prise en charge OpenMP avec des indicateurs de compilation appropriés.

En utilisant OpenMP, les développeurs peuvent créer des programmes parallèles qui exploitent efficacement les ressources des systèmes multicœurs, ce qui permet d'améliorer les performances de manière significative pour des tâches intensives en calcul et en traitement de données. Cependant, la programmation parallèle comporte des défis liés à la gestion de la concurrence, à la synchronisation et à la gestion de la mémoire partagée, qui nécessitent une compréhension approfondie des concepts d'OpenMP pour écrire du code robuste et efficace.

1. Points importants à dans l'implémentation d'un programme en utilisant des threads :

L'implémentation parallèle multithread consiste à diviser un programme en plusieurs threads (ou fils d'exécution) qui s'exécutent en parallèle sur un processeur multicœur ou sur plusieurs processeurs. Cette approche est utilisée pour améliorer les performances des applications en exploitant la capacité de traitement simultané offerte par le matériel informatique moderne.



Cette figure représentée l'Architecture de processus Multithread et single-thread

Voici quelques points importants à considérer lors de l'implémentation d'un programme en utilisant des threads :

- **Synchronisation** : La synchronisation est cruciale pour éviter les conflits entre les threads qui accèdent aux mêmes ressources partagées. Des mécanismes de synchronisation tels que les mutex (verrous), les sémaphores, et les variables de condition sont utilisés pour coordonner l'accès aux données partagées.

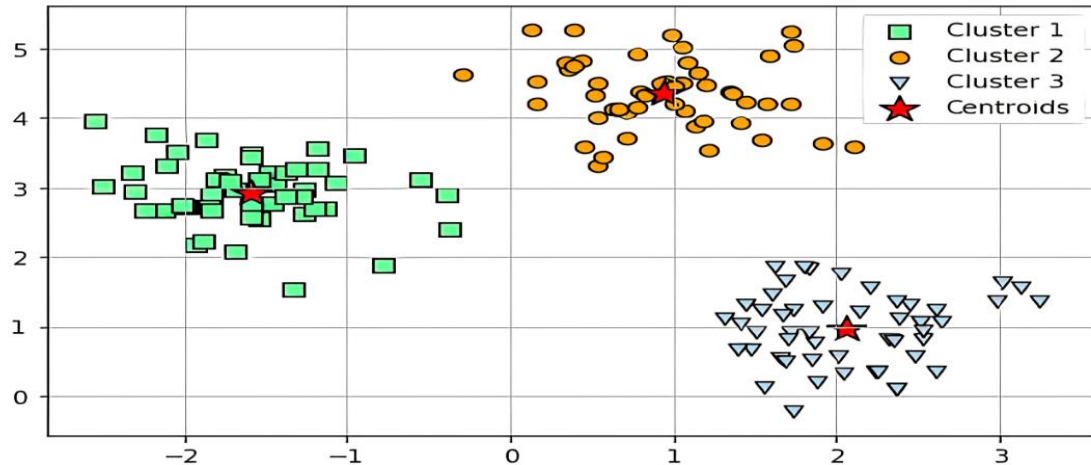
- Concurrence : La gestion de la concurrence est un aspect critique de la programmation multithread. Les développeurs doivent faire attention aux conditions de course (race conditions) qui peuvent survenir lorsque plusieurs threads tentent d'accéder ou de modifier la même donnée en même temps.
- Parallélisme : Le parallélisme peut améliorer les performances en répartissant les tâches sur plusieurs threads. Cependant, toutes les applications ne peuvent pas tirer parti du parallélisme de manière significative. Il est important de choisir judicieusement quelles parties de l'application doivent être parallélisées.
- Gestion des ressources : La gestion des ressources, telles que la mémoire, est essentielle. Les fuites de mémoire et les blocages (deadlocks) sont des problèmes courants dans les applications multithread. Un soin particulier doit être pris pour allouer et libérer correctement les ressources.
- Plateforme : Les détails de l'implémentation multithread peuvent varier en fonction de la plateforme (système d'exploitation) et du langage de programmation utilisé. Par exemple, Java a ses propres mécanismes de gestion de threads, tout comme C++.
- Débogage : Le débogage des applications multithread peut être plus complexe que le débogage des applications à un seul thread. Des outils de débogage spécifiques aux threads sont souvent nécessaires pour détecter et résoudre les problèmes.

L'implémentation multithread est couramment utilisée pour accélérer les opérations intensives en calcul, améliorer la réactivité des applications utilisateur, et gérer de multiples tâches simultanément. Cependant, elle peut également introduire des complexités supplémentaires dans le code, nécessitant une planification soignée et une attention particulière à la gestion des threads pour garantir la stabilité et la performance de l'application.

III. K-means:

- ❖ Comprenez en profondeur l'algorithme K-means. Assurez-vous de comprendre comment il fonctionne, y compris l'initialisation des centroïdes, les étapes d'attribution et de mise à jour des clusters

L'algorithme de classification K-means est un algorithme de clustering non supervisé utilisé pour regrouper un ensemble de données en clusters distincts en fonction de la similarité des éléments de données. Voici une description de la théorie mathématique sous-jacente à l'algorithme de classification K-means :



Cette figure représentée une implémentation de l'algorithme de K-means avec 3 centroïdes

1. Objectif de K-means :

L'objectif de l'algorithme K-means est de partitionner un ensemble de données en K clusters de telle sorte que la somme des carrés des distances entre les points de données et leurs centroïdes attribués soit minimisée. En d'autres termes, l'algorithme cherche à minimiser la fonction de coût, généralement appelée "inertie" ou "somme des carrés intra-cluster" :

$$J = \sum_{i=1}^K \sum_{j=1}^n ||x_j^{(i)} - \mu_i||^2$$

où :

- K est le nombre de clusters.
- n est le nombre de points de données.
- $x_j^{(i)}$ est le j-ème point de données du cluster i.
- μ_i est le centroïde du cluster i.

2. Processus itératif :

L'algorithme K-means fonctionne de manière itérative. Il commence par initialiser les centroïdes de manière aléatoire ou en utilisant d'autres méthodes, puis il effectue alternativement les deux étapes suivantes jusqu'à la convergence :

- ❖ Étape d'assignation : Chaque point de données est assigné au cluster dont le centroïde est le plus proche en termes de distance euclidienne. Cela signifie que chaque point de données est attribué au cluster dont le centroïde a la distance minimale par rapport à ce point.
- ❖ b. Étape de mise à jour des centroïdes : Une fois que tous les points de données ont été assignés à des clusters, les centroïdes de chaque cluster sont mis à jour en recalculant la moyenne des points de données appartenant à ce cluster.

3. Convergence :

L'algorithme continue d'itérer jusqu'à ce que la convergence soit atteinte, c'est-à-dire que les centroïdes ne changent plus de manière significative ou que la fonction de coût (inertie) atteigne un seuil prédéfini.

4. Choix du nombre de clusters K :

Le choix du nombre de clusters K est un aspect important de l'algorithme K-means. Il peut être déterminé de manière empirique en utilisant des techniques telles que la méthode du coude (Elbow method) ou en fonction des besoins spécifiques de l'application.

L'algorithme K-means est largement utilisé dans des domaines tels que la segmentation d'image, la classification de documents, la recommandation de produits, et bien d'autres applications où la classification ou la segmentation de données est nécessaire. Il repose sur des concepts mathématiques liés à la minimisation de la variance intra-cluster et à la recherche du meilleur partitionnement des données en clusters.

Étape de mise à jour des centroïdes (M):

Dans cette étape, les centroïdes des clusters sont mis à jour en recalculant la moyenne des points de données appartenant à chaque cluster. L'équation pour mettre à jour un centroïde est la suivante :

$$\mu_k = \frac{1}{|C_k|} \sum_{x_j \in C_k} x_j$$

- μ_k est le nouveau centroïde du cluster k.
- $|C_k|$ est le nombre de points de données dans le cluster k.
- $\sum_{x_j \in C_k} x_j$ représente la somme de tous les points de données x_j appartenant au cluster k.

L'algorithme K-means itère entre ces deux étapes (assignation et mise à jour des centroïdes) jusqu'à ce que la convergence soit atteinte, c'est-à-dire que les centroïdes ne changent plus de manière significative ou que la fonction de coût atteigne un seuil prédéfini.

Ces équations résument les principales opérations mathématiques effectuées par l'algorithme K-means pour regrouper des données en clusters en minimisant la somme des carrés des distances entre les points de données et les centroïdes de leurs clusters respectifs.

IV. Parallélisation de l'algorithme K-means avec OpenMP

L'algorithme K-means est principalement composé de deux étapes itératives : la mise en cluster des points de données et le recalcul des centroïdes. Chacune de ces étapes peut être parallélisée pour accélérer l'algorithme.

Division du travail en plusieurs threads

- Mise en cluster des points de données :

La première étape consiste à attribuer chaque point de données au cluster le plus proche (généralement en fonction de la distance euclidienne). Vous pouvez diviser les points de données en lots (mini-batch) et affecter chaque lot à un thread. Chaque thread traitera son lot de données de manière indépendante.

Assurez-vous que les threads n'accèdent pas simultanément aux mêmes variables partagées (par exemple, un tableau de clusters). Vous pouvez utiliser des mécanismes de synchronisation tels que `#pragma omp critical` ou `#pragma omp atomic` pour éviter les conflits.

- Recalcul des centroïdes :

La deuxième étape consiste à recalculer les centroïdes de chaque cluster en fonction des points de données assignés à chaque cluster. Pour paralléliser cette étape, vous pouvez diviser les clusters en sous-groupes et affecter chaque sous-groupe à un thread.

Assurez-vous que les threads ne modifient pas simultanément les mêmes variables partagées (par exemple, les coordonnées des centroïdes). Vous pouvez utiliser des mécanismes de synchronisation pour éviter les conflits, mais soyez attentif aux performances, car des mécanismes de synchronisation excessifs peuvent réduire l'efficacité de la parallélisation.

Coordination entre les threads

La coordination entre les threads est essentielle pour garantir que l'algorithme fonctionne correctement. Voici quelques considérations :

Utilisez `#pragma omp parallel` pour créer une région parallèle où les threads sont générés.

Utilisez `#pragma omp for` pour paralléliser les boucles for dans chaque étape de l'algorithme.

Utilisez des directives OpenMP telles que `#pragma omp critical` ou `#pragma omp atomic` lorsque plusieurs threads accèdent aux mêmes variables partagées pour éviter les conflits.

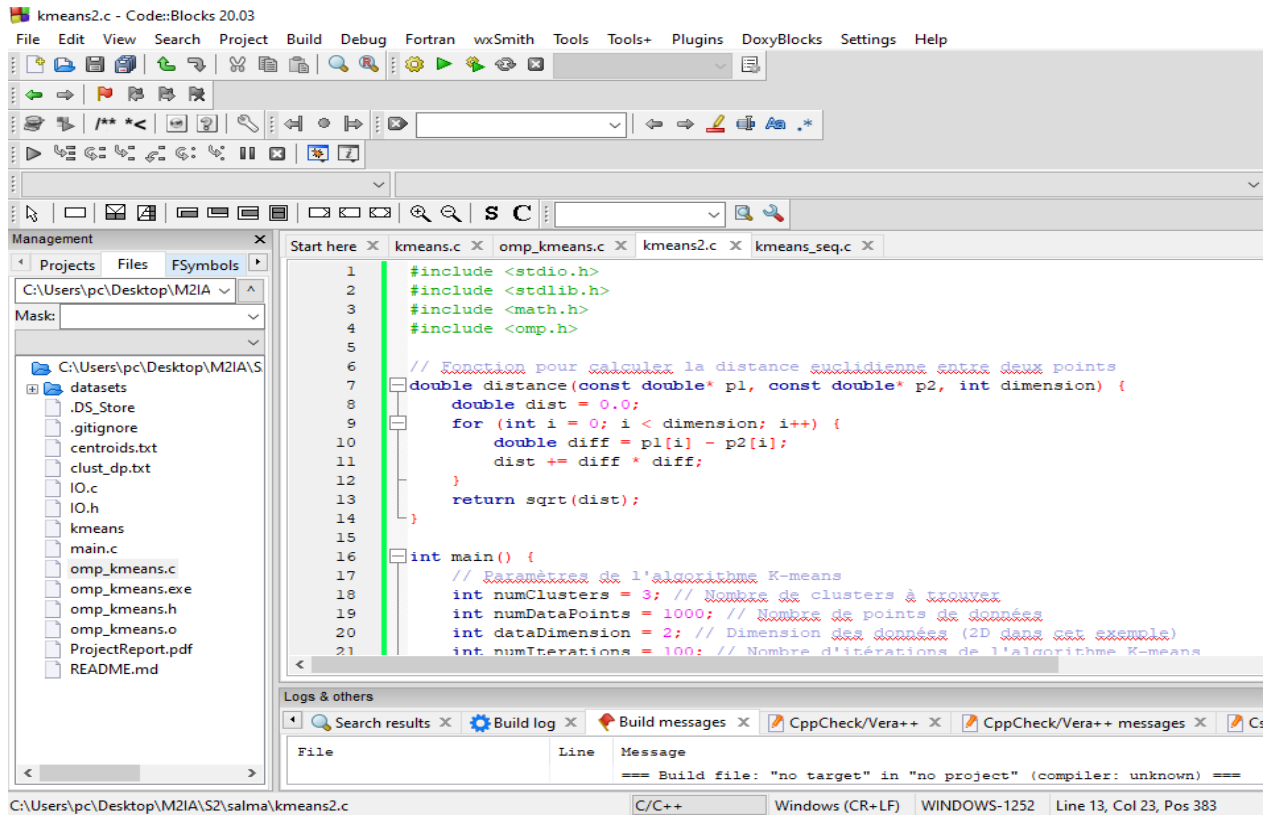
Utilisez des structures de données partagées pour stocker les résultats partiels et fusionnez-les à la fin de chaque étape.

Il est important de noter que la parallélisation introduit des surtêtes liées à la gestion des threads. Vous devez évaluer les performances de votre implémentation par rapport à l'implémentation séquentielle pour vous assurer que la parallélisation apporte des avantages significatifs.

Une fois que vous avez mis en œuvre cette approche, vous pouvez effectuer des expériences pour mesurer les gains de performances et discuter des résultats dans la section "Expérimentation et Résultats" de votre rapport de projet.

V. Implémentation et explication de code

Vous pouvez trouver le code source à mon Github :



Cette figure représente l'environnement de travail (code blocks)

Ce code implémente l'algorithme K-means avec une parallélisation multithread à l'aide de la bibliothèque OpenMP. L'algorithme K-means est un algorithme de classification non supervisée qui vise à regrouper un ensemble de données en clusters en fonction de la similarité entre les points de données. Voici une explication détaillée du code pour votre projet d'implémentation parallèle de l'algorithme K-means avec OpenMP :

- ✓ Inclusion de bibliothèques :

Le code commence par inclure les bibliothèques nécessaires, notamment `'stdio.h'`, `'stdlib.h'`, `'math.h'`, et `'omp.h'`, cette dernière étant utilisée pour gérer le parallélisme avec OpenMP.

- ✓ Fonction de calcul de distance :

La fonction `'distance'` calcule la distance euclidienne entre deux points, représentés par des tableaux de doubles. La distance euclidienne mesure la similarité entre les points.

- ✓ Paramètres de l'algorithme K-means :
Vous définissez ici les paramètres de l'algorithme, tels que le nombre de clusters (``numClusters``), le nombre de points de données (``numDataPoints``), la dimension des données (``dataDimension``), et le nombre d'itérations (``numIterations``) que l'algorithme va effectuer.
- ✓ Allocation de mémoire pour les données et les centroïdes :
Vous allouez de l'espace mémoire pour les données (points de données) et les centroïdes.

Les données sont générées aléatoirement pour la démonstration.
- ✓ Boucle d'itérations K-means :
Vous entrez dans une boucle d'itérations K-means qui se répète ``numIterations`` fois pour raffiner les clusters.
À chaque itération, vous allouez de l'espace pour stocker les nouvelles positions des centroïdes et le nombre de points dans chaque cluster.
- ✓ Parallélisation de la boucle principale :
Vous utilisez ``#pragma omp parallel`` pour créer un bloc de code parallélisé. Cela permet à plusieurs threads de travailler en parallèle pour traiter les points de données.
- ✓ Affichage des informations de thread :
Vous affichez des informations sur le démarrage et la fin de chaque itération pour chaque thread. Cela vous permettra de suivre l'exécution parallèle des threads.
- ✓ Parallélisation de la boucle de calcul des clusters :
Vous utilisez ``#pragma omp for`` pour paralléliser la boucle qui traite les points de données et les attribue aux clusters. Chaque thread travaille sur une partie des données.
- ✓ Calcul du cluster le plus proche :
Pour chaque point de données, chaque thread calcule le cluster le plus proche en mesurant la distance euclidienne entre le point de données et chaque centroïde.
- ✓ Mise à jour des nouvelles positions des centroïdes :
Vous mettez à jour les nouvelles positions des centroïdes en additionnant les données de chaque point attribué à un cluster.
- ✓ Utilisation de l'opération atomique :
Vous utilisez ``#pragma omp atomic`` pour effectuer une mise à jour atomique du nombre de points dans chaque cluster.
- ✓ Mise à jour des positions des centroïdes :

Une fois que tous les threads ont traité leurs données, vous mettez à jour les positions des centroïdes en divisant la somme des points du cluster par le nombre de points.

✓ Libération de la mémoire allouée :

Vous libérez la mémoire allouée pour les nouvelles positions des centroïdes et les tailles des clusters à la fin de chaque itération.

✓ Affichage des résultats :

Après toutes les itérations, vous affichez les positions finales des centroïdes pour chaque cluster.

✓ Libération de la mémoire :

Enfin, vous libérez la mémoire allouée pour les données et les centroïdes.

Ce code illustre comment utiliser OpenMP pour paralléliser l'algorithme K-means, ce qui peut accélérer considérablement le processus de clustering sur des systèmes multicœurs. Assurez-vous d'avoir configuré OpenMP correctement pour que la parallélisation fonctionne, et assurez-vous également de comprendre les spécificités de l'algorithme K-means pour l'adapter à vos besoins.

VI. Analyse de Résultat de notre code d'exécution :

Le résultat que vous montrez est le suivi de l'exécution de deux threads qui effectuent des itérations en parallèle, ainsi que les coordonnées des centres finaux de trois clusters. Permettez-moi de décomposer cela en détail et d'expliquer l'intérêt du parallélisme dans ce contexte.

```
Thread 0 / 2 : Starting iteration 0
Thread 1 / 2 : Starting iteration 0
Thread 0 / 2 : Completed iteration 0
Thread 1 / 2 : Completed iteration 0
Thread 1 / 2 : Starting iteration 1
Thread 0 / 2 : Starting iteration 1
Thread 1 / 2 : Completed iteration 1
Thread 0 / 2 : Completed iteration 1
Thread 1 / 2 : Starting iteration 2
Thread 0 / 2 : Starting iteration 2
```

Cette figure représente Le résultat qui fournit notre code

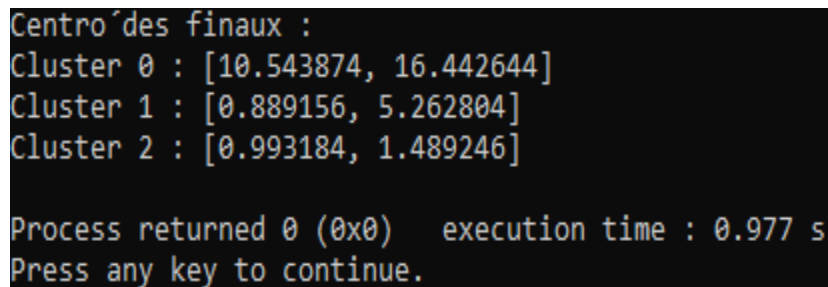
1. Threads et Itérations en Parallèle:

- Vous avez deux threads, numérotés 0 et 1, qui travaillent en parallèle. Cela signifie qu'ils exécutent leur travail simultanément pour accélérer le traitement.

- Chaque thread effectue un certain nombre d'itérations. Par exemple, le thread 0 effectue les itérations de 0 à 99. Pendant ce temps, le thread 1 effectue les itérations de 0 à 99 également. Cette parallélisation est bénéfique pour les tâches gourmandes en calcul, car elle permet d'exploiter efficacement les processeurs multi-cœurs.

2. Centres Finaux de Clusters:

- Vous montrez les centres finaux de trois clusters. Ces centres sont des coordonnées qui définissent les points centraux de ces groupes de données.



```
Centro' des finaux :  
Cluster 0 : [10.543874, 16.442644]  
Cluster 1 : [0.889156, 5.262804]  
Cluster 2 : [0.993184, 1.489246]  
  
Process returned 0 (0x0)   execution time : 0.977 s  
Press any key to continue.
```

Cette figure représente l'affichage de notre code d'exécution de notre code

Voici une explication de ces résultats :

- Les résultats indiquent les centres finaux de trois clusters.
- Chaque cluster est représenté par un ensemble de coordonnées [x, y]. Cela signifie que les centres finaux des clusters sont les points moyens ou centraux de chaque groupe de données. Ces coordonnées représentent les emplacements spatiaux autour desquels les points du cluster sont regroupés.

- Cluster 0, Cluster 1, Cluster 2:

- Vous avez trois clusters numérotés 0, 1 et 2, chacun ayant son propre centre. Chaque cluster est défini par des données similaires dans l'espace, et son centre est l'endroit vers lequel convergent ces données lors du processus de clustering.

- Process Returned 0 (0x0) :

- La ligne "Process returned 0 (0x0)" est généralement un message standard indiquant que le processus s'est terminé sans erreurs. Il s'agit d'un code de sortie (0) qui indique que le programme s'est achevé correctement.

- Execution Time (Temps d'exécution) :

- L'exécution a pris un certain temps, indiqué ici comme **0.977** secondes. Cela mesure la durée totale nécessaire pour exécuter l'algorithme de clustering et trouver les centres finaux des clusters.

- Explication :

- ❖ Les centres finaux des clusters obtenus à partir de l'algorithme de clustering sont essentiels pour identifier des groupes de données similaires. Ces clusters sont formés en minimisant la distance entre les points de données et les centres des clusters, ce qui permet de regrouper des données similaires ensemble.
- ❖ Par exemple, dans le Cluster 0, le centre est [10.543874, 16.442644], ce qui signifie que les données dans ce cluster ont tendance à se rassembler autour de ces coordonnées dans l'espace. De même, pour les autres clusters.
- ❖ Ces résultats sont utiles pour l'analyse de données et la segmentation. Ils vous permettent de comprendre comment vos données sont regroupées, d'identifier des tendances ou des groupes de données similaires, et de prendre des décisions éclairées basées sur ces informations.

- Interprétation :

- Ces résultats peuvent être utiles dans divers domaines. Par exemple, dans le domaine de la vente au détail, vous pourriez utiliser ces clusters pour identifier différents types de clients en fonction de leurs comportements d'achat.

- Dans la recherche médicale, vous pourriez regrouper des patients en fonction de leurs caractéristiques médicales pour personnaliser les traitements.

- Dans l'analyse de données géospatiales, vous pourriez segmenter des régions en fonction de données telles que la densité de population, la criminalité, etc.

- L'algorithme de clustering est un outil puissant pour découvrir des structures dans vos données et prendre des décisions éclairées en fonction de ces structures.

3. Interprétation du Résultat:

- Le calcul parallèle est souvent utilisé dans le contexte de l'apprentissage automatique et de l'analyse de données, notamment pour des algorithmes comme le clustering (groupement). L'algorithme de clustering le plus courant est K-means, qui vise à regrouper les données en clusters de manière à minimiser la distance entre les points d'un cluster et son centre.

- L'exécution parallèle de threads permet de réduire le temps nécessaire pour effectuer de nombreuses itérations de calcul, ce qui peut être très bénéfique pour des tâches intensives en calcul telles que le calcul des clusters.

- Dans votre résultat, on peut voir comment le parallélisme accélère l'exécution des itérations. Chaque thread commence des itérations à des indices différents, mais ils travaillent simultanément pour compléter leur travail.

- Les centres finaux des clusters sont importants dans l'analyse des données, car ils représentent les caractéristiques centrales de chaque groupe de données. Les coordonnées fournies indiquent où ces centres se trouvent dans l'espace.

4. Intérêt du Parallélisme:

- L'intérêt du parallélisme réside dans l'amélioration des performances. En divisant une tâche en plusieurs sous-tâches exécutées en parallèle, on peut tirer parti de processeurs multi-cœurs pour accélérer le traitement.

- Cela est particulièrement utile pour des applications qui exigent beaucoup de calculs, comme l'apprentissage automatique, la simulation, l'analyse de données, etc. Le parallélisme permet de réduire considérablement le temps d'exécution de ces tâches.

- Dans le contexte de l'analyse de données, le parallélisme est souvent utilisé pour accélérer des algorithmes itératifs, comme K-means, où les calculs répétés sont effectués pour converger vers une solution.

En résumé, le parallélisme permet d'optimiser le temps de calcul en effectuant des tâches en simultané sur des processeurs multi-cœurs. Cela est crucial pour accélérer des tâches intensives en calcul, notamment dans des domaines comme l'analyse de données et l'apprentissage automatique, où de nombreuses itérations sont nécessaires pour atteindre une solution optimale.

5. La comparaison entre l'exécution séquentielle et parallèle de l'algorithme K-means

La comparaison entre l'exécution séquentielle et parallèle de l'algorithme K-means avec OpenMP permet de déterminer si l'ajout de parallélisme améliore les performances de l'algorithme en termes de vitesse d'exécution. Voici les principaux avantages et inconvénients de chaque approche :

❖ Exécution Séquentielle :

```
Centro des finaux :  
Cluster 0 : [4.114767, 5.761138]  
Cluster 1 : [2.839936, 3.427393]  
Cluster 2 : [0.504467, 0.503389]  
  
Process returned 0 (0x0)   execution time : 1.378 s  
Press any key to continue.
```

Cette figure représente notre exécution de code en séquentielle

Avantages :

1. Facilité de mise en œuvre : L'implémentation séquentielle est plus simple à coder et à comprendre, ce qui facilite le débogage et la maintenance.
2. Compatibilité : L'algorithme séquentiel peut s'exécuter sur n'importe quelle machine sans se soucier du support d'OpenMP ou d'autres bibliothèques de parallélisme.

Inconvénients :

1. Temps d'exécution plus long : L'algorithme séquentiel traite les données un point à la fois, ce qui peut être lent pour des volumes de données importants.
2. Utilisation CPU limitée : Les processeurs multicœurs modernes ne sont pas utilisés efficacement, ce qui laisse des ressources inexploitées.

❖ Exécution Parallèle (OpenMP) :

```
Centro des finaux :  
Cluster 0 : [10.543874, 16.442644]  
Cluster 1 : [0.889156, 5.262804]  
Cluster 2 : [0.993184, 1.489246]  
  
Process returned 0 (0x0)   execution time : 0.977 s  
Press any key to continue.
```

Cette figure représente notre exécution de code en Parallèle

Avantages :

1. Temps d'exécution réduit : L'exécution parallèle répartit la charge de travail entre plusieurs cœurs de CPU, ce qui permet de traiter les données plus rapidement.

2. Utilisation efficace des ressources : OpenMP permet d'exploiter pleinement la puissance de calcul des processeurs multicœurs, ce qui peut considérablement accélérer le traitement des données.

3. Adaptabilité : Les performances s'améliorent à mesure que le nombre de cœurs disponibles augmente, ce qui permet de tirer parti des systèmes informatiques modernes.

Inconvénients :

1. Complexité : L'ajout de parallélisme augmente la complexité du code et peut nécessiter des ajustements pour assurer une exécution correcte.

2. Débogage plus difficile : Le débogage de code parallèle peut être plus complexe en raison de problèmes potentiels tels que les conditions de course.

On remarque que le temps de exécution est prendre 1.378 secondes par contre de le code d'exécution en parallèle qui prendre seulement 0.977.

En résumé, l'exécution parallèle avec OpenMP offre généralement des avantages significatifs en termes de performances en répartissant la charge de travail sur plusieurs cœurs de CPU, ce qui accélère le traitement des données. Cependant, cela nécessite une certaine complexité supplémentaire dans le code et peut être plus difficile à déboguer en raison des problèmes de concurrence potentiels. L'efficacité du parallélisme dépend de plusieurs facteurs, notamment la taille des données, le nombre de cœurs de CPU disponibles et la qualité de l'implémentation. Il est donc recommandé de mesurer les performances sur votre système spécifique pour déterminer si le parallélisme est bénéfique pour votre application particulière.

Références

- 1) [2019PA080086 SENAGI Ran.pdf](#)
- 2) [MinGW-w64 - for 32 and 64 bit Windows download | SourceForge.net](#)
- 3) [calcul-parallele-avec-R extraits.pdf \(univ-lyon1.fr\)](#)
- 4) [Microsoft PowerPoint - SG6-03-Multithreading-OpenMP.pptx \(supelec.fr\)](#)
- 5) [ProjectReport.pdf](#)
- 6) [Chapitre 2 - Parallélisme - Classification - partie1/2 - YouTube](#)