PC-2022/23 k-means

Jacopo Manetti
E-mail address
jacopo.manetti1@stud.unifi.it

Abstract

In questo progetto abbiamo confrontato le prestazioni dell'algoritmo k-means in versione parallela con quella sequenziale, tutti i risultati sono stati ottenuti su un computer dotato di un processore Intel i7 11800h a 8 core (16 logici). Utilizzando C++ e la libreria OpenMP, abbiamo sviluppato entrambe le implementazioni e abbiamo utilizzato tecniche di profilazione per valutare le prestazioni. I risultati hanno dimostrato che l'utilizzo della versione parallela dell'algoritmo ha permesso di ottenere uno speed-up medio di circa 6,5 rispetto alla versione sequenziale, dimostrando l'efficienza dell'implementazione parallela su questo specifico hardware. Inoltre, questo progetto ha fornito una panoramica delle tecniche di programmazione parallela e dei loro vantaggi nell'ottimizzare i tempi di esecuzione di un algoritmo.

1. L'algoritmo

L'algoritmo k-means è un metodo di clustering dei dati che permette di suddividere un insieme di punti in k gruppi (dove k è specificato dall'utente) in modo tale da minimizzare la varianza all'interno di ogni gruppo.

Il funzionamento dell'algoritmo k-means può essere riassunto nei seguenti passaggi:

1. Inizializzare k centroidi (punti rappresentativi dei gruppi) in modo casuale all'interno del dataset.

2. Assegnare ogni punto del dataset al centroide più vicino, calcolato utilizzando la distanza euclidea.

3. Aggiornare le posizioni dei centroidi come la media delle posizioni dei punti assegnati a quel centroide.

4. Ripetere i passaggi 2 e 3 finché i centroidi non cambiano posizione.

1.1. Il codice

Per l'implementazione dell'algoritmo k-means sono state create 2 classi **Point** e **Cluster**.

La classe **Point** rappresenta un punto con coordinate x e y e un identificativo di cluster a cui appartiene. La classe include due costruttori: uno che prende come argomenti le coordinate x e y del punto e imposta l'id del cluster a 0, l'altro che imposta le coordinate x e y del punto a 0 e l'id del cluster a 0.

La classe include anche quattro metodi pubblici:

- **get_x_coord()**: restituisce la coordinata x del punto.
- **get_y_coord()**: restituisce la coordinata y del punto.
- **get_id_cluster()**: restituisce l'id del cluster a cui appartiene il punto.
- **set_id_cluster(int id_cluster)**: imposta l'id del cluster a cui appartiene il punto.

La classe **Cluster** rappresenta un cluster con un centroide, cioè un punto con coordinate x e y che rappresenta il centro del cluster. La classe include due costruttori: uno che prende come argomenti le coordinate x e y del centroide e imposta il numero di punti nel cluster a 0, l'altro che imposta le coordinate x e y del centroide a 0 e il numero di punti nel cluster a 0.

La classe include anche cinque metodi pubblici:

- **add_point(Point point)**: aggiunge il punto specificato al cluster e aggiorna le coordinate del centroide del cluster.
- **free_point()**: rimuove tutti i punti dal cluster.
- **get_x_coord()**: restituisce la coordinata x del centroide del cluster.
- **get_y_coord()**: restituisce la coordinata y del centroide del cluster.
- **update_coords()**: aggiorna le coordinate del centroide del cluster se necessario. Restituisce true se le coordinate del centroide sono state aggiornate, false altrimenti.

La funzione principale è **main()**, che chiede all'utente di inserire il numero di punti da generare e il numero di cluster desiderati, quindi genera i punti e i cluster casualmente utilizzando le funzioni **initializePoint()** e **initializeCluster()**.

Successivamente, il ciclo while continuerà a iterare fino a quando il valore di conv è true. All'inizio di ogni iterazione, la funzione **calculateDistance()** viene chiamata per calcolare la distanza di ogni punto dai centroidi dei cluster e assegnare il punto al cluster con centroide più vicino. Dopo di che, la funzione **updateClusters()** viene chiamata per aggiornare le coordinate dei centroidi dei cluster in base ai punti assegnati a ciascun cluster. Se almeno uno dei centroidi viene aggiornato, conv viene impostato su true, altrimenti viene impostato su false. Quindi, se conv è true, il ciclo while continuerà con un'altra iterazione, altrimenti il ciclo terminerà e il programma visualizzerà il plot dei punti.

I metodi principali nel main sono:

- **calculateDistance()**: viene utilizzata per calcolare la distanza di ogni punto dai centroidi dei cluster e assegnare il punto al cluster con centroide più vicino. Per ogni punto, viene calcolata la distanza dal centroide del primo cluster e viene memorizzato l'indice del cluster con il centroide più vicino. Quindi, il ciclo interno itera su ogni cluster e calcola la distanza del punto dal centroide di ogni cluster. Se la distanza è minore della distanza minima precedentemente calcolata, viene aggiornata la distanza minima e l'indice del cluster con il centroide più vicino.

  Una volta che la distanza minima e l'indice del cluster con il centroide più vicino sono stati calcolati per ogni punto, il punto viene assegnato al cluster con il centroide più vicino utilizzando il metodo set_id_cluster() della classe Point.

- **updateCluster()**: viene utilizzata per aggiornare la posizione dei centroidi dei cluster una volta che tutti i punti sono stati assegnati a un cluster. Per ogni cluster, viene chiamato il metodo free_point() della classe Cluster, che azzera il numero di punti assegnati al cluster, la somma delle coordinate x e y dei punti assegnati al cluster e la dimensione del cluster. Quindi, viene eseguito un altro ciclo for per ogni punto e, se il punto appartiene al cluster corrente (determinato dall'id del cluster del punto), viene aggiunto il punto al cluster utilizzando il metodo add_point() della classe Cluster. Infine, viene chiamato il metodo update_coords() della classe Cluster per ogni cluster per aggiornare le coordinate del centroide del cluster in base ai punti assegnati al cluster. Il metodo restituisce true se le coordinate del centroide sono cambiate e false altrimenti. Il risultato di update_coords() per ogni cluster viene utilizzato per determinare se il processo di aggiornamento dei cluster deve continuare o meno. Se almeno uno dei cluster ha cambiato le sue coordinate, il processo continua altrimenti termina e la funzione restituisce false.

1.2. La versione parallela

La parte parallela del codice è gestita dall'utilizzo della libreria **OpenMP**, che permette di eseguire in parallelo le operazioni di calcolo della distanza tra i punti e i cluster, l'aggiornamento dei cluster e il riassetto dei punti in base al cluster più vicino.

Nella funzione **calculateDistance()** si trova l'intera regione parallela viene utilizzata la direttiva "parallel for" per indicare che il ciclo "for" che segue deve essere eseguito in parallelo dai diversi thread. In questo modo, ogni iterazione del ciclo viene eseguita da un thread differente, permettendo di ottenere una maggiore velocità di esecuzione.

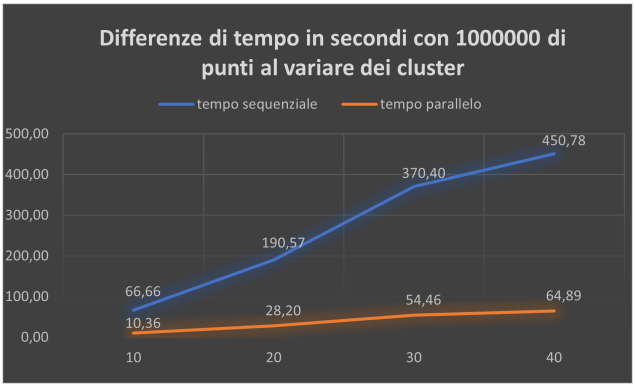
sempre nella funzione calculateDistance() viene utilizzato **add_point()** in cui la direttiva "atomic" viene utilizzata per garantire la correttezza dei risultati nelle operazioni di aggiornamento dei cluster. In particolare, questa direttiva indica che l'operazione che segue deve essere eseguita in modo atomico, cioè deve essere eseguita in modo esclusivo da un solo thread alla volta. In questo modo si evitano problemi di race condition, cioè situazioni in cui più thread cercano di modificare lo stesso valore contemporaneamente causando risultati non corretti.

2. Analisi delle prestazioni

Di seguito vengono riportati i risultati degli speed-up ottenuti dall'esecuzione del codice in versione sequenziale e parallela in varie condizioni, notare che per il controllo dello speed-up l'inizializzazione casuale dei punti e dei cluster è stata eseguita con lo stesso seed, quindi il set di dati era sempre il solito.

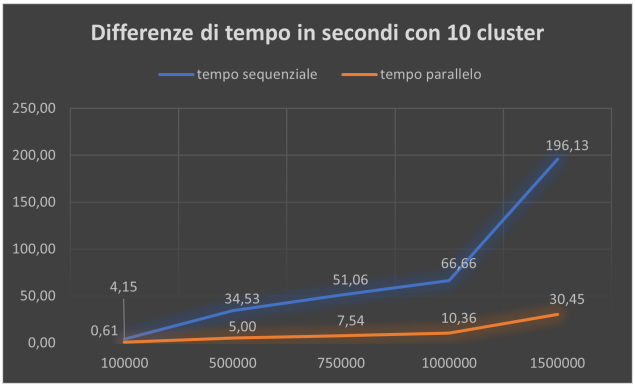
2.1. Esecuzione con 1000000 punti al variare dei Cluster

Nel primo test sono stati presi in considerazione 1000000 di punti ed è stato calcolato il tempo di esecuzione (in secondi) al variare del numero di cluster.



2.2. Esecuzione con 10 Cluster al variare del numero di punti

Nel secondo test invece sono stati fissati 10 cluster ed è stato calcolato il tempo di esecuzione al variare dei punti.

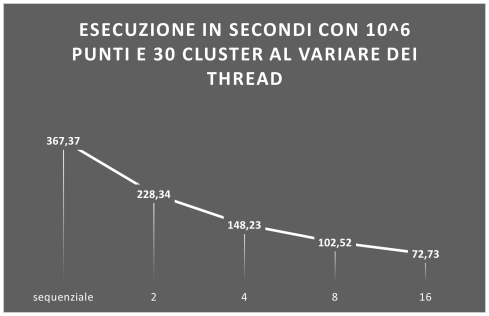


2.3. Esecuzione al variare dei thread

Gli ultimi 2 test riguardano l'esecuzione del codice parallelo limitando il numero massimo di thread disponibili usando il comando num_thread(), partendo da 1 (esecuzione sequenziale) arrivando fino a 16 (numero massimo di core disponibili sulla macchina).

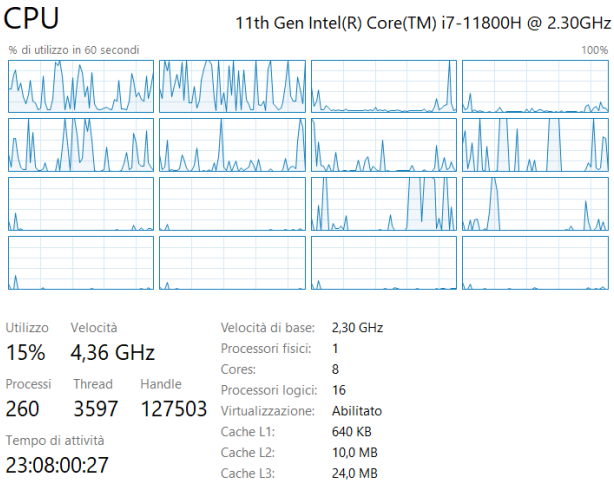
2.3.1 Esecuzione in secondi con 1000000 punti e 10 cluster al variare dei thread

2.3.2 Esecuzione in secondi con 1000000 punti e 30 cluster al variare dei thread

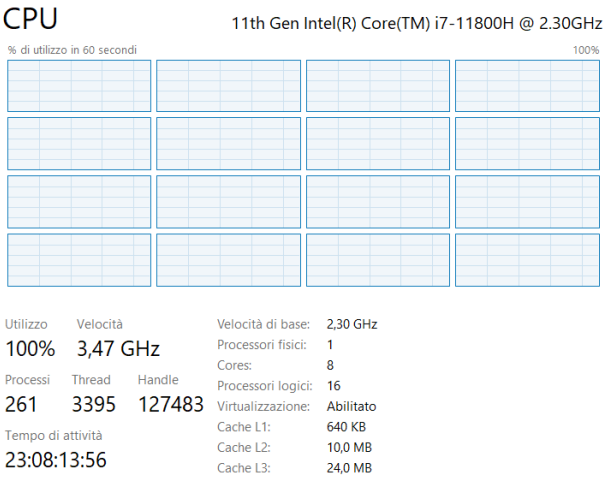


3. Conclusione

Come dimostrato nelle sezioni precedenti, l'implementazione parallela presenta un netto vantaggio in termini di tempo di esecuzione rispetto alla versione sequenziale, con una media di aumento della velocità di circa 6,5 volte. Questo è stato possibile grazie all'utilizzo di OpenMP, che ha permesso di distribuire il carico di lavoro su più core del processore. Di seguito vengono riportate delle immagini dell'uso del processore durante l'esecuzione sequenziale e parallela.



Utilizzo dell CPU durante esecuzione sequenziale



Utilizzo dell CPU durante esecuzione parallela