K-means - OpenMP

Alessandro Marinai, Eleonora Ristori

Aprile 2023

1 Introduzione

L'elaborato ha l'obiettivo di realizzare tre implementazioni dell'algoritmo di clustering K-means: la prima in maniera sequenziale, la seconda e la terza in modo parallelo sfruttando la libreria openMP.

2 Struttura del codice

Il programma si articola in 3 file:

- s_kmeans.h che contiene l'implementazione sequenziale dell'algoritmo K-means;
- p_kmeans.h che contiene le implementazioni parallele dell'algoritmo;
- utils.h che contiene i metodi relativi alla generazione dei punti e al salvataggio dei risultati.

2.1 Generazione dei punti

Al fine di testare l'algoritmo è stato realizzato un metodo che genera punti in cluster di forma globulare dato che questo è un requisito fondamentale per il corretto funzionamento di K-means.
Viene quindi generato il primo centroide randomicamente, mentre i successivi vengono selezionati solo se sufficientemente distanti dagli altri.
Una volta selezionati i centroidi si procede alla generazione dei punti che avviene utilizzando una distribuzione gaussiana per ciascuna dimensione, centrata nel centroide e con varianza passata come parametro. I punti sono inseriti in un array linearizzato per offrire performance migliori evitando l'uso di doppi puntatori.

```
void generateCluster(int num_points,int data_point_dim, int k, float sigma, float*data_points, float*centroids){
    //Assigning centroids
    for(int dim=0; dim<data_point_dim; dim++){
        centroids[dim] = float(std::rand()) / RAND_MAX *100*k;
    }
    int num_assigned_centroids = 1;

    while(num_assigned_centroids != k){

        for(int dim=0; dim<data_point_dim; dim++){
            centroids[num_assigned_centroids*data_point_dim+dim] =  float(std::rand()) / RAND_MAX *100*k;
        }
        int centroid_id = 0;
        float sum_distances;
        do {
            sum_distances = 0;
            for(int dim=0; dim<data_point_dim; dim++) {
                float diff = centroids[centroid_id*data_point_dim+dim] - centroids[num_assigned_centroids*data_point_dim+dim];
                sum_distances += diff*diff;
            }
            centroid_id++;
        }while(centroid_id < num_assigned_centroids && sum_distances > sigma*sigma*100);
        if(centroid_id == num_assigned_centroids){
            num_assigned_centroids++;
        }
    }


    //Generate random points
    std::default_random_engine generator;
    for(int point_id=0; point_id<num_points; point_id++){
        int cluster = int(std::rand()) % k;

        for(int dim=0; dim < data_point_dim; dim++){
            std::normal_distribution<double> distribution( Mean0: centroids[cluster*data_point_dim+dim], Sigma0: sigma);
            data_points[point_id*data_point_dim + dim] = float(distribution( &: generator));
        }
    }
}
```

Figure 1: Funzione per la generazione randomica dei punti

2.2 Versione sequenziale

Nella versione sequenziale dell'algoritmo, in primo luogo si procede all'inizializzazione dei centroidi. Come nel caso della loro generazione, il primo centroide è scelto randomicamente tra i punti del dataset mentre gli altri sono selezionati solamente se la loro distanza dai centroidi già assegnati è maggiore di una soglia

prefissata.

```
void initialize_centroids(int data_point_dim, float* centroids, const float *data_points, int k, float threshold){
    for(int dim=0; dim<data_point_dim; dim++){
        centroids[dim] = data_points[dim];
    }
    int num_assigned_centroids = 1;
    int point_id = 1;
    while(num_assigned_centroids != k){

        for(int dim=0; dim<data_point_dim; dim++){
            centroids[num_assigned_centroids*data_point_dim+dim] =  data_points[point_id*data_point_dim+dim];
        }
        int centroid_id = 0;
        float sum_distances;
        do {
            sum_distances = 0;
            for(int dim=0; dim<data_point_dim; dim++) {
                float diff = centroids[centroid_id*data_point_dim+dim] - centroids[num_assigned_centroids*data_point_dim+dim];
                sum_distances += diff*diff;
            }
            centroid_id++;
        }while(centroid_id < num_assigned_centroids && sum_distances > threshold);
        if(centroid_id == num_assigned_centroids){
            num_assigned_centroids++;
            point_id++;
        }
        else {
            point_id++;
        }
    }
}
```

Figure 2: Funzione per l'inizializzazione dei centroidi

Una volta assegnati i centroidi inizia l'algoritmo vero e proprio che si articola in un ciclo do-while che ha come condizione di uscita il fatto che i centroidi non si siano spostati più di una certa tolleranza o che si sia raggiunto un numero massimo di iterazioni.
Ad ogni iterazione si eseguono due metodi:

- assign_cluster_s() che esegue l'assegnazione dei punti al cluster più vicino;
- calculate_centroid_s() che ricalcola i nuovi centroidi come media dei punti assegnati al cluster.

```
// Main loop
int iteration = 0;
do {

    // Assign each data point to the nearest centroid
    assign_cluster_s(cluster_assignment, data_points, centroids, num_data_points, num_centroids, data_point_dim, updated_centroids, assigned_points);


    calculate_centroid_s(updated_centroids, assigned_points, centroids, old_centroids, num_centroids, data_point_dim);


    iteration++;

}while (iteration < max_iterations && centroid_distance_s( o_centroids: old_centroids,  n_centroids: centroids,  k: num_centroids, data_point_dim) > tolerance);
```

Figure 3: Main loop K-means sequenziale

2.2.1 assign_cluster_s()

L'assegnazione dei punti al cluster avviene iterando sull'intero dataset, calcolando le distanze di ciascun punto da ogni centroide e determinando quindi quello più vicino.
Per ogni punto l'id del centroide più vicino viene salvato nella struttura dati cluster_assignment. Vengono inoltre sommate le coordinate dei punti via via che vengono assegnati ai cluster, mantenendo anche un contatore sul numero totale di punti per cluster.

```
void assign_cluster_s(int *cluster_assignment, const float* data_points, const float* centroids, int num_data_points,
                      int num_centroids, int data_point_dim, float* updated_centroids, int* assigned_points) {
    for(int data_point_id=0; data_point_id<num_data_points; data_point_id++){
        float min_dist = INFINITY;
        int min_centroid_id = -1;
        for (int centroid_id = 0; centroid_id < num_centroids; centroid_id++) {
            float dist = 0;
            for (int dim = 0; dim < data_point_dim; dim++) {
                float diff = data_points[data_point_id * data_point_dim + dim] - centroids[centroid_id * data_point_dim + dim];
                dist += diff * diff;
            }
            if(dist < min_dist){
                min_dist = dist;
                min_centroid_id = centroid_id;
            }
        }
        cluster_assignment[data_point_id] = min_centroid_id;
        for (int dim = 0; dim < data_point_dim; dim++)
            updated_centroids[min_centroid_id*data_point_dim+dim] += data_points[data_point_id*data_point_dim+dim];
        assigned_points[min_centroid_id]+= 1;
    }
}
```

Figure 4: Funzione per l'assegnazione dei punti ai cluster

2.2.2 calculate_centroid_s()

Questa funzione salva i vecchi centroidi che saranno utilizzati al fine di determinare lo spostamento rispetto a quelli nuovi e ricalcola i nuovi centroidi come media dei punti appartenenti al cluster dividendo le somme accumulate nel metodo precedente per il totale dei punti per cluster.

```
void calculate_centroid_s(float* updated_centroids, int* assigned_points, float* centroids, float* old_centroids,
                          int num_centroids, int data_point_dim) {
    for(int centroid_id=0; centroid_id<num_centroids; centroid_id++){
        for (int dim = 0; dim < data_point_dim; dim++) {
            if (assigned_points[centroid_id] > 0) {
                old_centroids[centroid_id * data_point_dim + dim] = centroids[centroid_id * data_point_dim + dim];
                centroids[centroid_id * data_point_dim + dim] = updated_centroids[centroid_id*data_point_dim+dim] / assigned_points[centroid_id];
            }
        }
    }
}
```

Figure 5: Funzione per il calcolo dei nuovi centroidi

2.3 Versioni parallele

Al fine di offrire di fornire una completa analisi delle prestazioni, sono state implementate due diverse versioni di K-means parallelo:

- la prima accumula la somma dei punti assegnati ad un cluster in modo atomico per poi dividere il totale per il numero di punti assegnati al cluster;
- la seconda scorre tutti i punti sfruttando l'assignment già avvenuto per il ricalcolo del centroide.

Come nel caso sequenziale, in entrambe le versioni si procede inizializzando i centroidi in modo tale che siano sufficientemente distanti.

2.3.1 Versione parallela 1

Nella prima versione, dopo l'inizializzazione dei centroidi si avvia il ciclo do-while che presenta le stesse condizioni di terminazione del caso sequenziale, tuttavia in questo caso al suo interno si riconoscono due cicli for marcati mediante la direttiva #pragma omp parallel for che consente lo smistamento delle iterazioni tra i diversi thread.
Con il primo ciclo si provvede alla assegnazione dei punti del dataset al cluster più vicino, mentre con il secondo si calcolano i nuovi centroidi in base all'assegnazione ottenuta. Essendo questa seconda operazione conseguente alla precedente, è necessario mantenere la barriera implicita introdotta dal ciclo for parallelo.

```
void kmeans(float* data_points, float* centroids, int* cluster_assignment, int num_data_points, int data_point_dim,
            int num_centroids, int max_iterations, float tolerance, int num_threads) {
    initialize_centroids_p(data_point_dim, centroids, data_points,  k: num_centroids,  threshold: 100);
    float* old_centroids = new float[num_centroids * data_point_dim];
    int iteration = 0;
    float distance = 0;
    do {
        // Assegnazione dei punti ai centroidi
#pragma omp parallel for num_threads(num_threads) default(none) \
firstprivate(num_data_points, data_point_dim, num_centroids, data_points, centroids) shared(cluster_assignment)
        for(int data_point_id=0; data_point_id<num_data_points; data_point_id++){
            float min_dist = INFINITY;
            int min_centroid_id = -1;
            for (int centroid_id = 0; centroid_id < num_centroids; centroid_id++) {
                float dist = 0;
                for (int dim = 0; dim < data_point_dim; dim++) {
                    float diff = data_points[data_point_id * data_point_dim + dim] - centroids[centroid_id * data_point_dim + dim];
                    dist += diff * diff;
                }
                if(dist < min_dist){
                    min_dist = dist;
                    min_centroid_id = centroid_id;
                }

            }
            cluster_assignment[data_point_id] = min_centroid_id;
        }
```

```
        // Ricalcolo dei centroidi
#pragma omp parallel for num_threads(num_threads) default(none) firstprivate(num_data_points, data_point_dim, \
num_centroids, data_points, cluster_assignment) shared(centroids, old_centroids)
        for(int centroid_id=0; centroid_id<num_centroids; centroid_id++){
            for (int dim = 0; dim < data_point_dim; dim++) {
                int num_points_assigned = 0;
                float sum = 0;
                for (int data_point_id = 0; data_point_id < num_data_points; data_point_id++) {
                    if (cluster_assignment[data_point_id] == centroid_id) {
                        sum += data_points[data_point_id * data_point_dim + dim];
                        num_points_assigned++;
                    }
                }
                if (num_points_assigned > 0) {
                    old_centroids[centroid_id * data_point_dim + dim] = centroids[centroid_id * data_point_dim + dim];
                    centroids[centroid_id * data_point_dim + dim] = sum / float(num_points_assigned);
                }
            }
        }

        iteration++;

    }while (iteration < max_iterations && centroid_distance( o_centroids: old_centroids,  n_centroids: centroids,  k: num_centroids, data_point_dim) > tolerance);
    delete[] old_centroids;
}
```

Figure 6: K-means parallelo versione 1

2.3.2 Versione parallela 2

Nella seconda versione il ciclo do-while presenta sempre le stesse condizioni di terminazione ma ciò che cambia è la modalità in vengono ricalcolati i centroidi. Infatti essi vengono calcolati sfruttando l'accumulazione atomica delle coordinate dei punti assegnati ad un certo cluster che avviene nel primo ciclo for dove si aggiorna anche un contatore sul numero totale di punti per cluster.
Il secondo ciclo in questo caso si occupa semplicemente di eseguire la divisione dell'accumulazione eseguita nel metodo precedente per il numero totale di punti assegnati al cluster. Di seguito si riporta il codice relativo a questo metodo.

```
void kmeans_p2(float* data_points, float* centroids, int* cluster_assignment, int num_data_points, int data_point_dim,
               int num_centroids, int max_iterations, float tolerance, int num_threads) {
    initialize_centroids_p(data_point_dim, centroids, data_points,  k: num_centroids,  threshold: 100);
    float* old_centroids = new float[num_centroids * data_point_dim];
    float* updated_centroids = new float[num_centroids * data_point_dim];
    int* assigned_points = new int[num_centroids];
    int iteration = 0;
```

```
    do {
        // Assegnazione dei punti ai centroidi
#pragma omp parallel for num_threads(num_threads) default(none) firstprivate(num_data_points, data_point_dim, \
num_centroids, data_points, centroids) shared(cluster_assignment, updated_centroids, assigned_points)
        for(int data_point_id=0; data_point_id<num_data_points; data_point_id++){
            float min_dist = INFINITY;
            int min_centroid_id = -1;
            for (int centroid_id = 0; centroid_id < num_centroids; centroid_id++) {
                float dist = 0;
                for (int dim = 0; dim < data_point_dim; dim++) {
                    float diff = data_points[data_point_id * data_point_dim + dim] - centroids[centroid_id * data_point_dim + dim];
                    dist += diff * diff;
                }
                if(dist < min_dist){
                    min_dist = dist;
                    min_centroid_id = centroid_id;
                }

            }
            cluster_assignment[data_point_id] = min_centroid_id;
            for (int dim = 0; dim < data_point_dim; dim++){
#pragma omp atomic
                updated_centroids[min_centroid_id*data_point_dim+dim] += data_points[data_point_id*data_point_dim+dim];
            }
#pragma omp atomic
            assigned_points[min_centroid_id]+= 1;
        }


        // Ricalcolo dei centroidi
#pragma omp parallel for num_threads(num_threads) default(none) firstprivate(num_data_points, data_point_dim, \
num_centroids, data_points, cluster_assignment, assigned_points, updated_centroids) shared(centroids, old_centroids)
        for(int centroid_id=0; centroid_id<num_centroids; centroid_id++){
            for (int dim = 0; dim < data_point_dim; dim++) {
                if (assigned_points[centroid_id] > 0) {
                    old_centroids[centroid_id * data_point_dim + dim] = centroids[centroid_id * data_point_dim + dim];
                    centroids[centroid_id * data_point_dim + dim] = updated_centroids[centroid_id*data_point_dim+dim] / assigned_points[centroid_id];
                }
            }
        }

        iteration++;

    }while (iteration < max_iterations /*&& centroid_distance(old_centroids, centroids, num_centroids, data_point_dim) > tolerance*/);
    delete[] old_centroids;
}
```

Figure 7: K-means parallelo versione 2

3 Analisi dei risultati

Al fine di valutare il beneficio offerto dalle versioni parallele in termini di speed-up sono stati eseguiti diversi esperimenti. Lo speed-up è stato calcolato in termini di wall-clock time ovvero determinando il tempo di esecuzione dei diversi algoritmi.

3.1 Valutazione dello speed-up al variare del numero di thread

3.1.1 Versione parallela 1

In questo test si osserva lo speed-up che si ottiene al variare del numero di thread con cui viene eseguita la versione 1 del programma parallelo. Il test è stato condotto facendo variare il numero di thread da 1 a 24 con 5.000.000 di punti e 5 cluster. I risultati sono mostrati in figura 8.

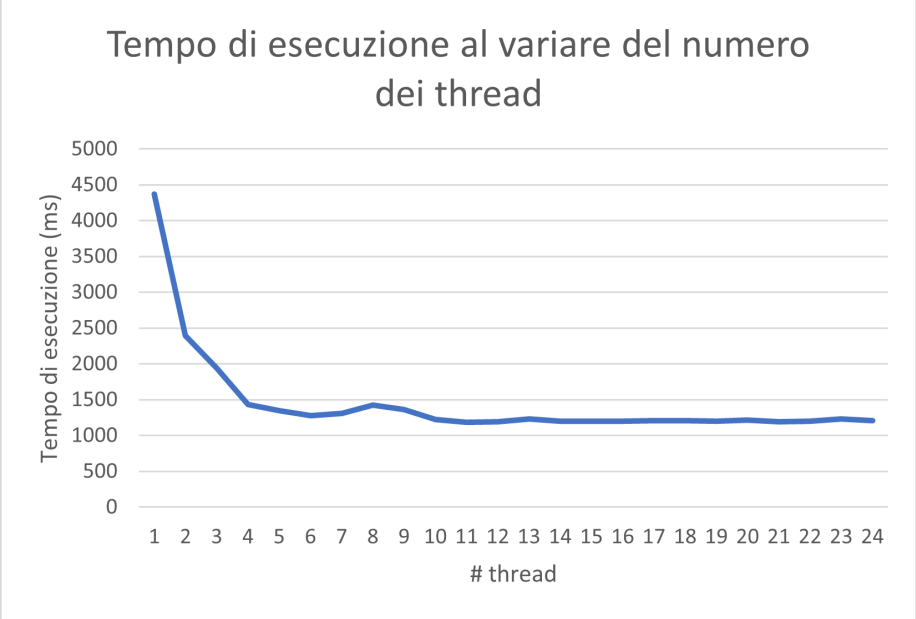

Figure 8: Grafico dei tempi di esecuzione al variare del numero di thread

Come era lecito attendersi si osserva una notevole riduzione del tempo di esecuzione all'aumentare del numero di thread già intorno al valore 6 che poi prosegue in modo molto più lento fino al valore di 12 che è il minimo valore ottenuto. Da lì in poi infatti non si apprezza più un significativo decremento in quanto il calcolatore su cui sono stati eseguiti i test ha una CPU Intel® Core™ i7-9750H a 6 core, quindi con 12 thread logici si raggiunge il massimo grado di

parallelizzazione.

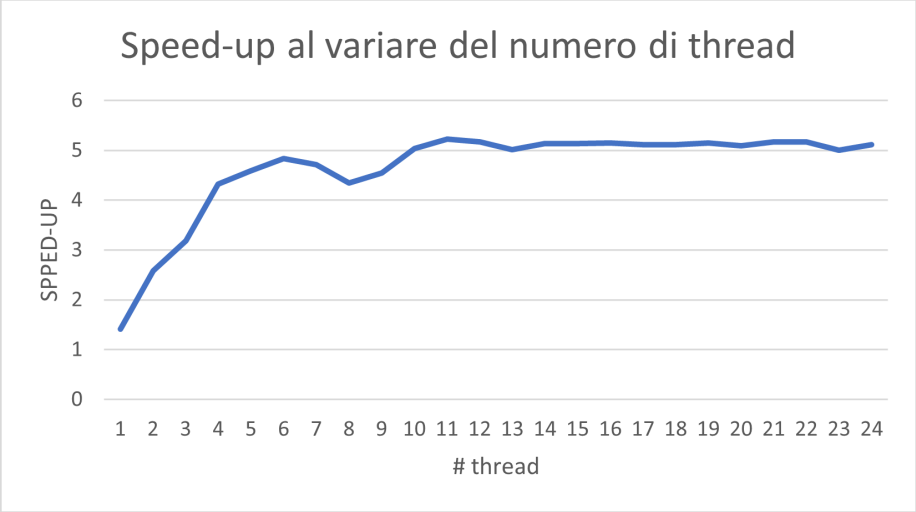

Figure 9: Grafico speed-up al variare del numero di thread

Il grafico in figura 9 mostra invece lo speed-up offerto dal programma parallelo rispetto alla versione sequenziale al variare del numero di thread. In accordo con quanto detto in precedenza, si osserva un aumento dello speed-up per un numero di thread che va da 1 a 12, mentre per un numero maggiore lo speed-up resta pressoché invariato assestandosi intorno al valore 5.

3.1.2 Versione parallela 2

Lo stesso esperimento è stato condotto per la seconda versione. In questo caso il test è stato condotto con 5.000.000 e 50 cluster visto che, come verrà mostrato in seguito, questa versione ha performance migliori con un numero di cluster più elevato. I risultati sono mostrati in figura 10. Anche in questo caso si osserva la miglior prestazione con 12 thread con uno speed-up che si assesta attorno al valore 5.

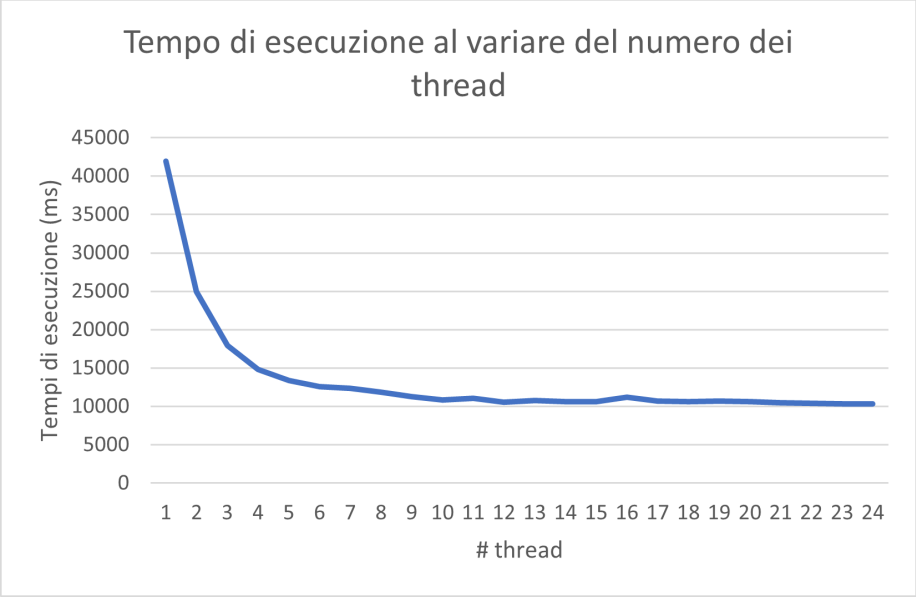

Figure 10: Grafico dei tempi di esecuzione al variare del numero di thread

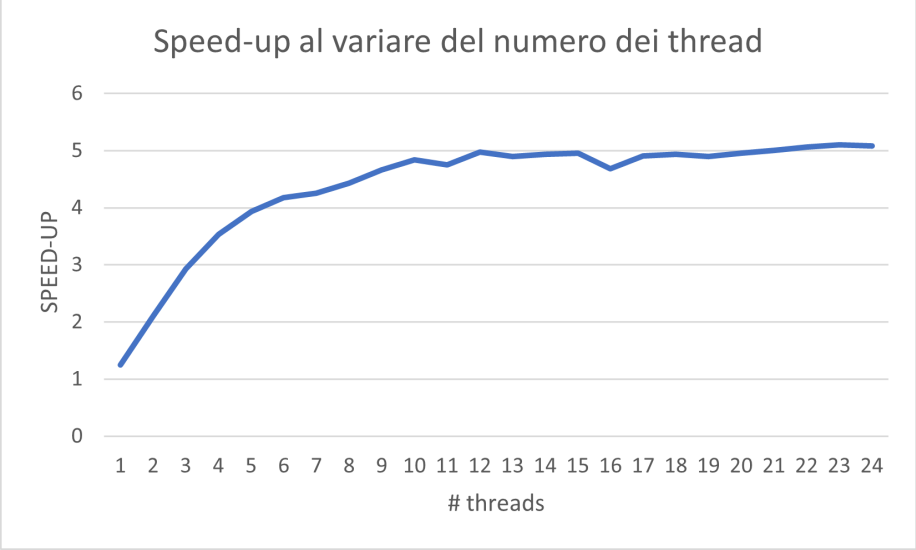

Figure 11: Grafico dello speed-up al variare del numero di thread

3.2 Confronto delle prestazioni delle due versioni parallele

In questo secondo esperimento, si confrontano le performance in termini di tempi di esecuzione delle due implementazioni parallele dell'algoritmo all'aumentare del numero di centroidi dato che la differenza tra le due riguarda proprio l'aggiornamento di questi ultimi.

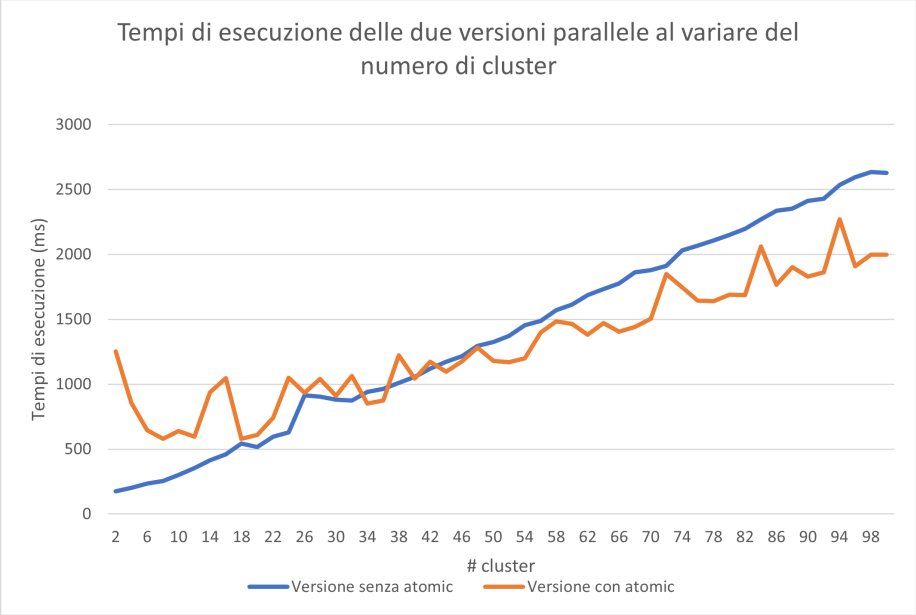

Figure 12: Grafico dei tempi di esecuzione delle due versioni parallele al variare del numero di cluster

Come si osserva dal grafico in figura 12, le performance dei due algoritmi dipendono dal numero di cluster utilizzati.

- Se il numero di centroidi è contenuto (minore di 30-35) l'algoritmo che non fa uso delle istruzioni atomiche risulta più veloce. Ciò trova spiegazione nel fatto che avendo pochi cluster la probabilità che due thread concorrentemente provino ad accedere alla stessa locazione di memoria dove avviene accumulazione delle coordinate dei punti assegnati risulta essere molto elevata e ciò fa sì che spesso alcuni thread siano bloccati in attesa che gli altri completino la scrittura.

- All'aumentare del numero di centroidi, invece, tale probabilità si riduce notevolmente e ciò fa sì che la seconda implementazione che fa uso delle istruzioni atomiche risulti più performante.

Un'ultima osservazione sul grafico in figura 12 riguarda l'andamento nettamente più oscillatorio della versione che fa uso di atomic rispetto all'altra. Ciò è legato

al fatto che le istruzioni atomiche fanno si che alcuni thread rimangano in attesa per evitare una scrittura concorrente con conseguente corruzione della memoria. Questo determina tempi di esecuzione non deterministici.

3.3 Valutazione dello speed-up al variare del numero di punti

Il terzo esperimento condotto è volto ad analizzare lo speed-up delle versioni parallele rispetto al caso sequenziale al variare della dimensione del problema. Per questo esperimento sono stati eseguite tre misurazioni:

- la prima è stata eseguita considerando la condizione di early termination. Si riportano i risultati ottenuti nel grafico in figura 13.

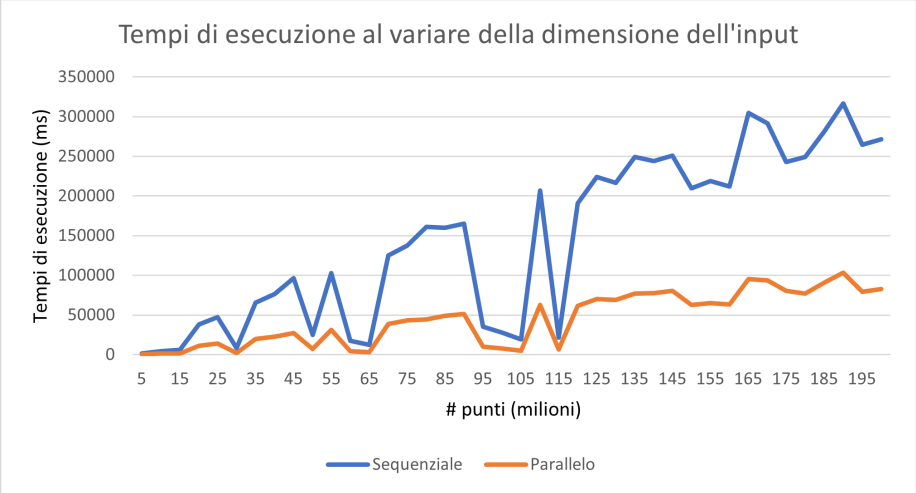

Figure 13: Grafico dei tempi di esecuzione al variare della dimensione dell'input con condizione di early termination

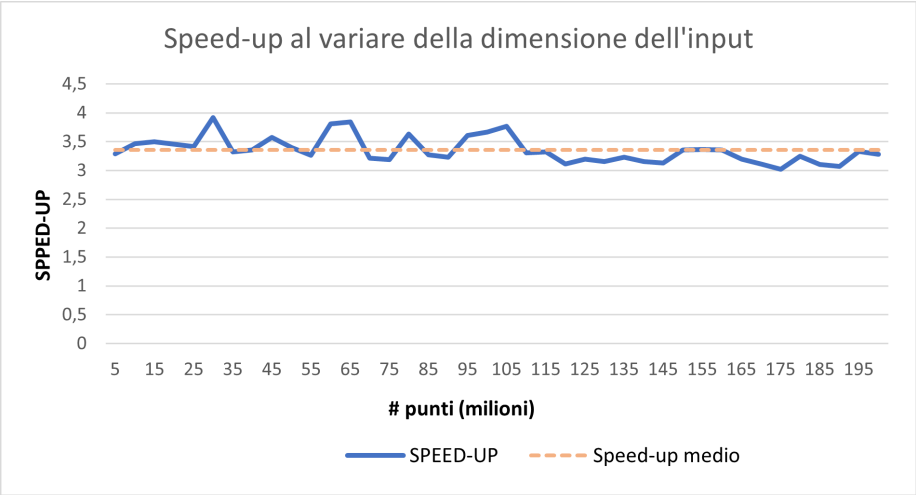

Figure 14: Grafico dello speed-up al variare della dimensione dell'input con condizione di early termination

- vista la poca chiarezza dei risultati dell'esperimento precedente la seconda misurazione invece è stata eseguita con un numero di iterazioni fisso escludendo la condizione precedente. In questo esperimento è stata utilizzata la prima versione parallela con un numero di centroidi fissato a 5. I grafici in figura 15 e 16 mostrano i risultati ottenuti.

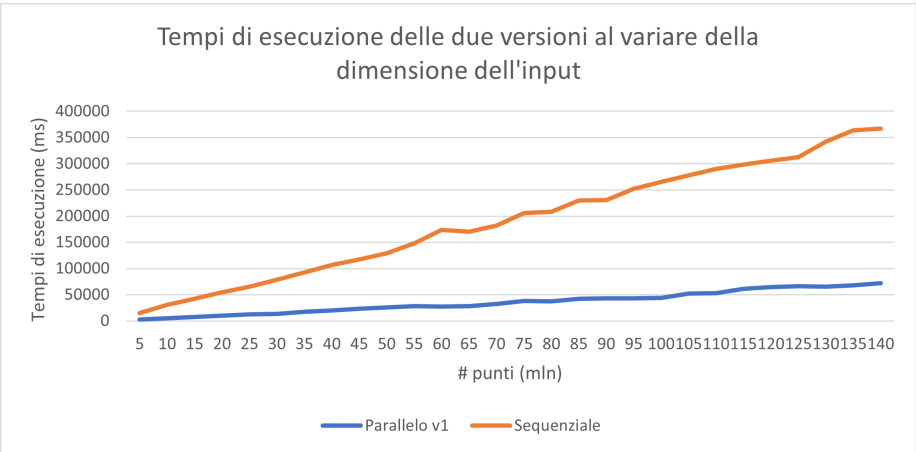

Figure 15: Grafico dei tempi di esecuzione della prima versione parallela e di quella sequenziale al variare della dimensione dell'input con un numero di iterazioni fisso

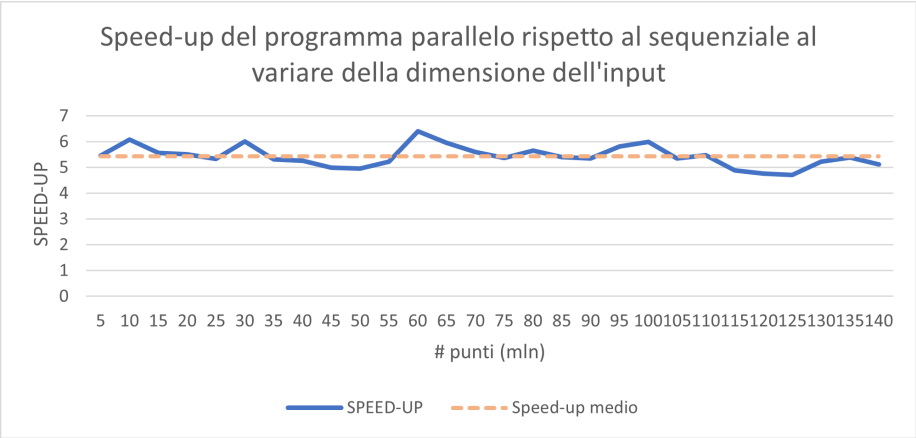

Figure 16: Grafico dello speed-up della prima versione parallela al variare della dimensione dell'input con un numero di iterazioni fisso

Dal grafico in figura 16 si osserva che la prima versione parallela offre uno speed-up di circa 5.4 rispetto alla versione sequenziale.

- La terza misurazione replica l'esperimento precedente fissando però il numero di centroidi a 50 ed utilizzando quindi la seconda implementazione parallela che risulta vantaggiosa quando il numero di centroidi è maggiore. I risultati dell'esperimento sono mostrati in figura 17 e 18.

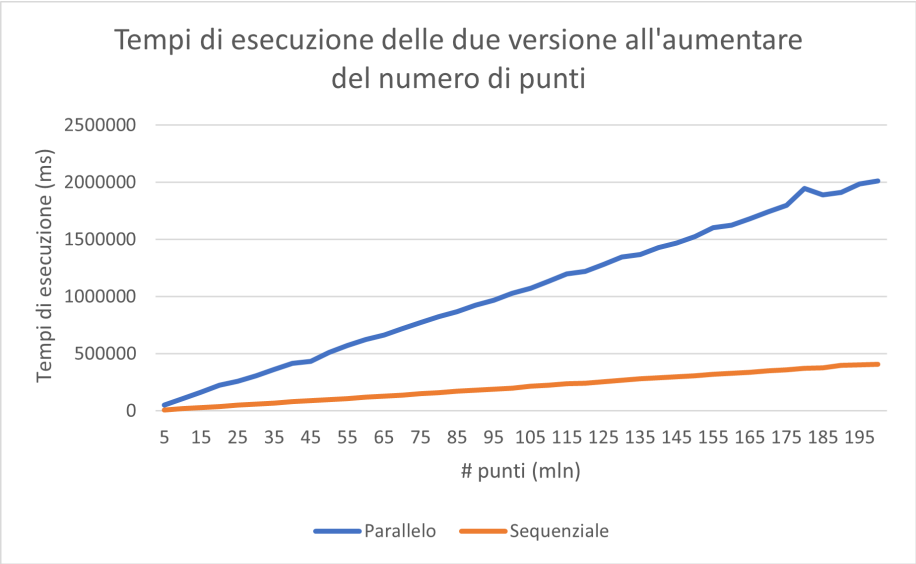

Figure 17: Grafico dei tempi di esecuzione della seconda versione parallela e di quella sequenziale al variare della dimensione dell'input con un numero di iterazioni fisso

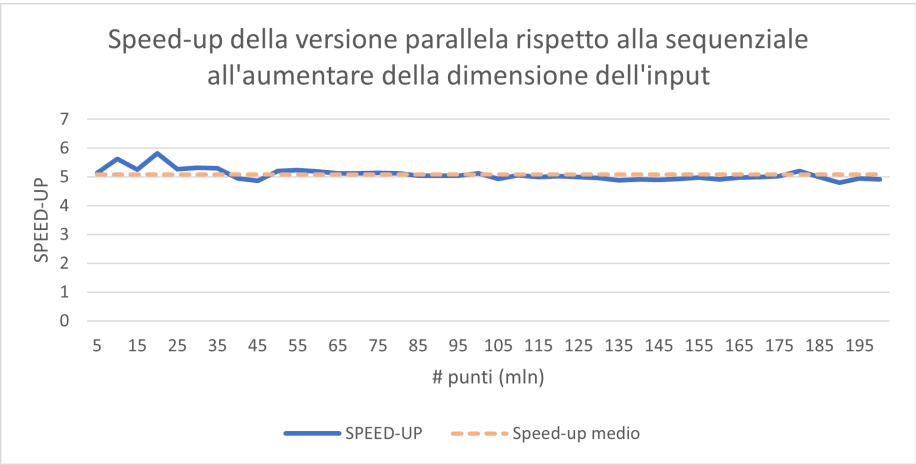

Figure 18: Grafico dello speed-up della seconda versione parallela al variare della dimensione dell'input con un numero di iterazioni fisso

In questo caso il grafico in figura 18 suggerisce uno speed-up della versione parallela rispetto alla sequenziale che supera 5.

3.4 Valutazione dello speed-up al variare del numero di centroidi

In quest'ultimo caso sono state analizzate le prestazioni della versione sequenziale e della seconda versione parallela (sempre utilizzando il numero ottimale di thread) al variare del numero di centroidi da 10 a 20000. È stata utilizzata solo la seconda versione dato che la prima a partire da un numero di cluster pari a 50, come mostrato nella sezione 3.2, ha prestazioni peggiori della seconda.
I risultati sono mostrati nel grafico in figura 19.

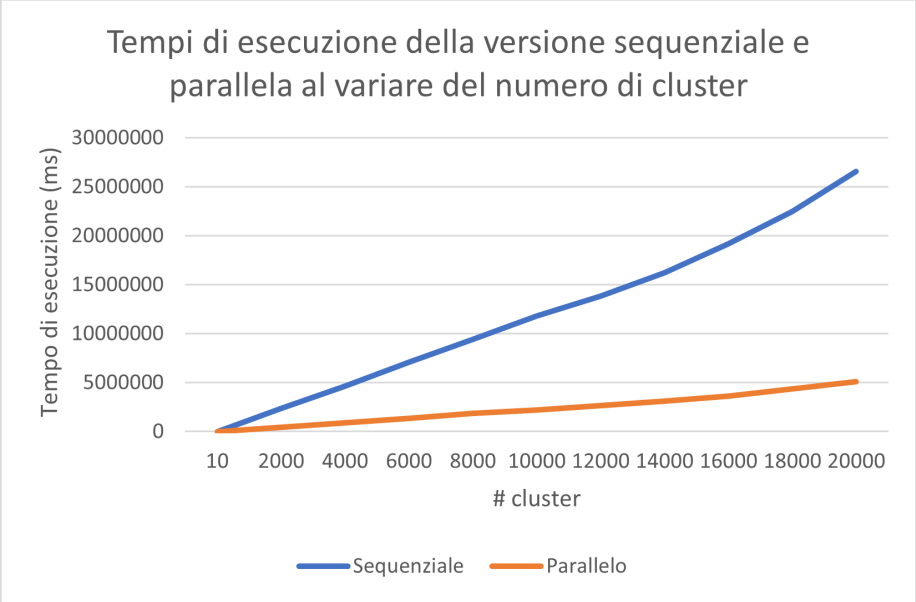

Figure 19: Grafico dei tempi di esecuzione al variare del numero di centroidi

Come ci si attendeva, anche in questo caso si osservano performance migliori della versione parallela con uno speed-up medio che si assesta intorno al valore 5.2.

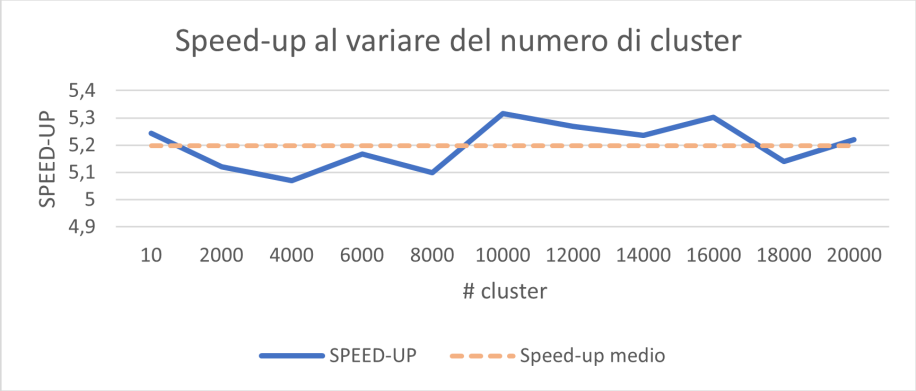

Figure 20: Grafico relativo allo speed-up al variare del numero di centroidi