K-means - OpenMP

Alessandro Marinai, Eleonora Ristori

Aprile 2023

1 Introduzione

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

2 Struttura del codice

Il programma si articola in 4 file:

- s_kmeans.h che contiene l'implementazione sequenziale dell'algoritmo K-means;
- p_kmeans.h che contiene l'implementazione parallela 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 cluster già assegnati è maggiore di una soglia

prefissata.
Una volta assegnati i centroidi inizia l'algoritmo vero e proprio che si articola

```
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

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.

2.2.1 assign_cluster_s()

L'assegnazione dei punti al cluster avviene iterando sull'intero dataset e 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.

```
void assign_cluster_s(int* cluster_assignment, float* data_points, float* centroids, int num_data_points, int num_centroids, int data_point_dim) {
    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;
    }
}
```

Figure 3: 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 utilizzando l'assegnamento che è stato definito al punto precedente.

```
void calculate_centroid_s(float* data_points, int* cluster_assignment, float* centroids, float* old_centroids, int num_data_points, 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++) {
            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);
            }
        }
    }

}
```

Figure 4: Funzione per il calcolo dei nuovi centroidi

2.3 Versione parallela

Come nel caso sequenziale, anche in questo caso si procede inizializzando i centroidi in modo tale che siano sufficientemente distanti. Si avvia quindi il ciclo do-while che presenta le stesse condizioni di terminazione del caso precedente, 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 al calcolo dei 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 5: K-means parallelo

3 Analisi dei risultati

Al fine di valutare il beneficio offerto dalla versione parallela 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 due algoritmi.

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

In questo test si osserva lo speed-up che si ottiene al variare del numero di thread con cui viene eseguito il programma parallelo. I risultati sono mostrati in figura 7.

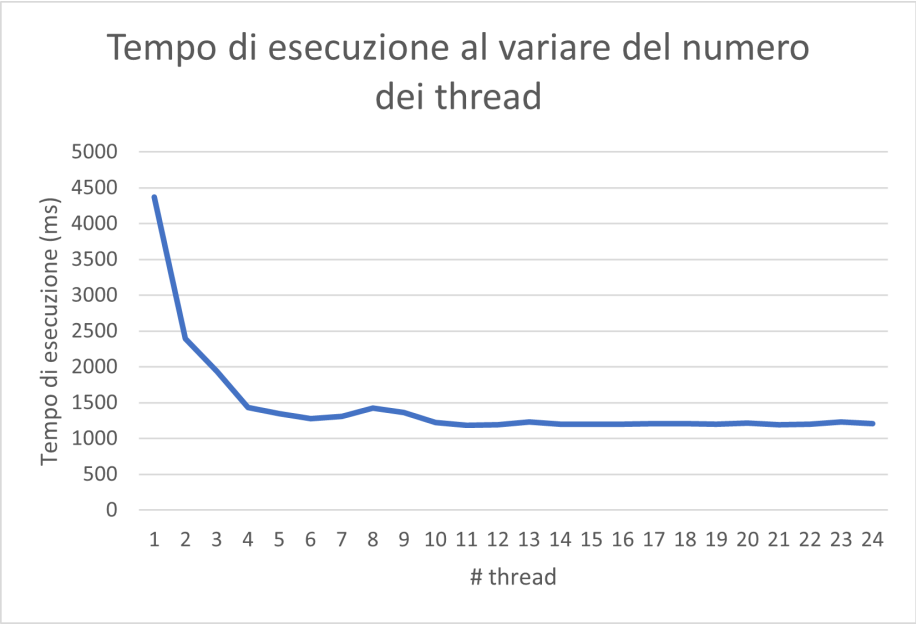

Figure 6: Grafico dei tempi di esecuzione al variare della dimensione dell'input

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.

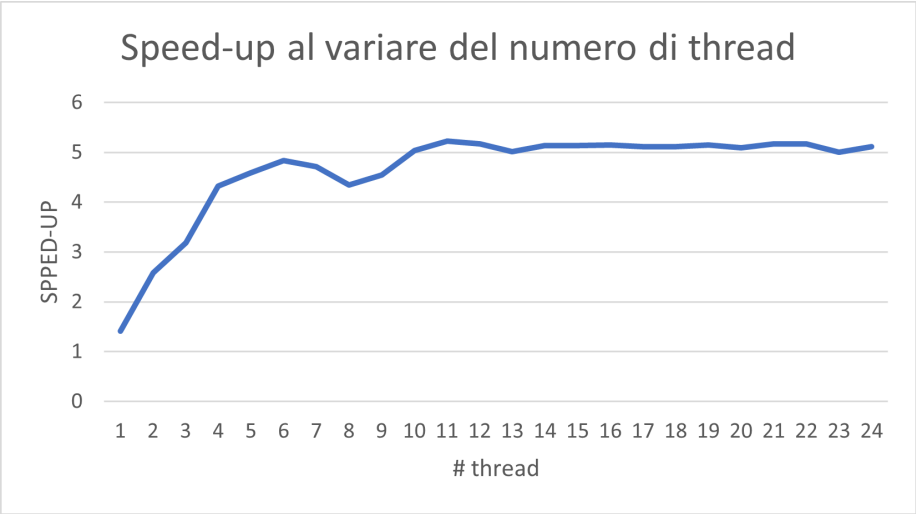

Figure 7: Grafico speed-up al variare del numero di thread

Il grafico in figura 8 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 6, mentre per un numero maggiore lo speed-up resta pressoché invariato.

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

Il secondo esperimento condotto, invece, è volto ad analizzare lo speed-up della versione parallela con numero di thread ottimale rispetto al caso sequenziale al variare della dimensione del problema.

Il grafico in figura 9 mostra l'andamento dei tempi di esecuzione per input

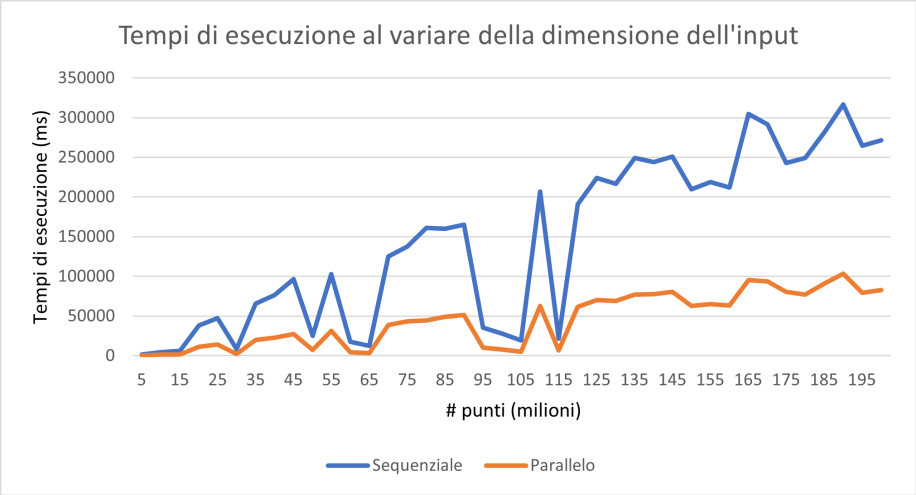

Figure 8: Grafico dei tempi di esecuzione al variare della dimensione dell'input

che variano da 5 a 200 milioni di punti con un netto speed-up della versione parallela che si assesta intorno al valore 3.35. L'andamento oscillatorio dei tempi è dovuto alla dipendenza da dati generati casualmente. Infatti le run con tempi di esecuzione minori hanno avuto un numero di iterazioni totali minori a causa della condizione di early termination. È stato scelto di lasciare questi risultati perchè, nonostante le oscillazioni dipendano dal numero di cicli effettuati per quel particolare insieme di dati, è abbastanza chiaro l'andamento dei tempi all'aumentare del numero dei punti.

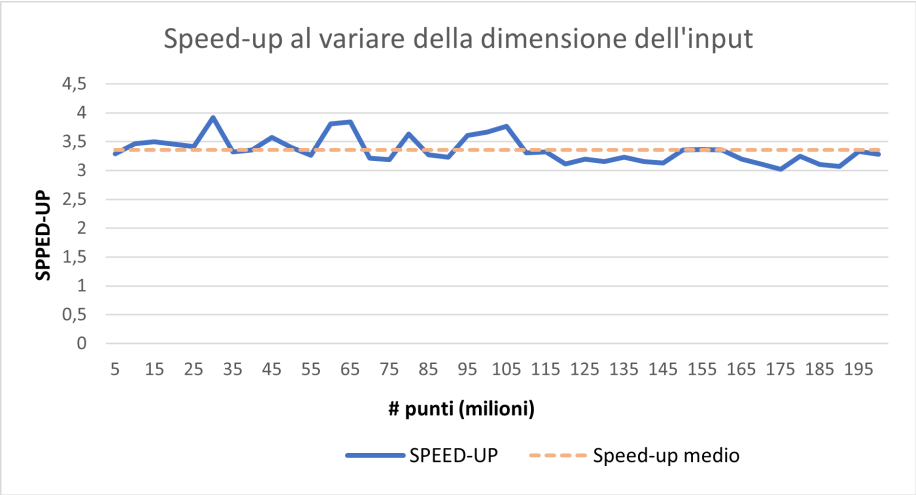

Figure 9: Grafico dei tempi di esecuzione al variare della dimensione dell'input

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

In quest'ultimo caso sono state analizzate le prestazioni delle due versioni (sempre utilizzando il numero ottimale di thread) al variare del numero di centroidi. I risultati sono mostrati nel grafico in figura 11.

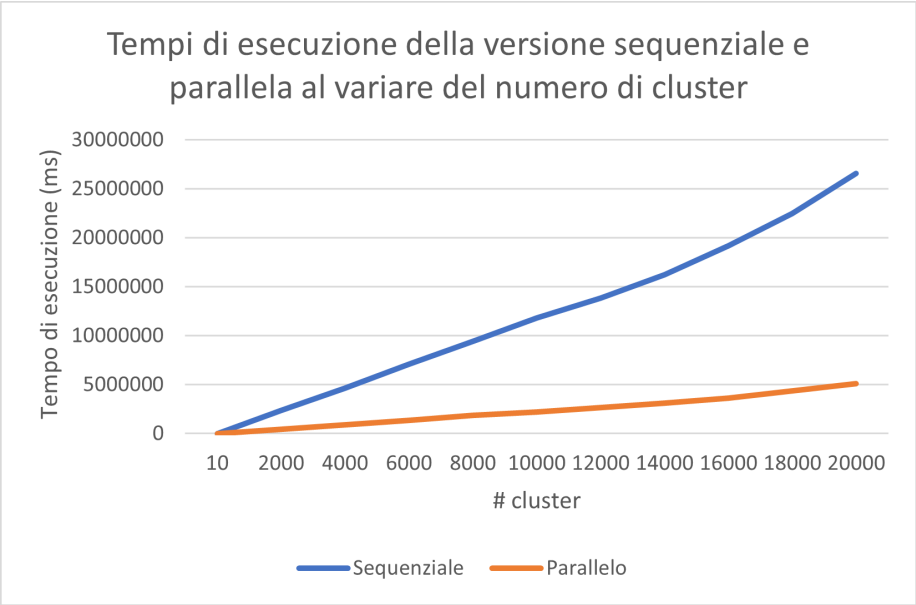

Figure 10: Grafico dei tempi di esecuzione al variare del numero di centroidi

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.

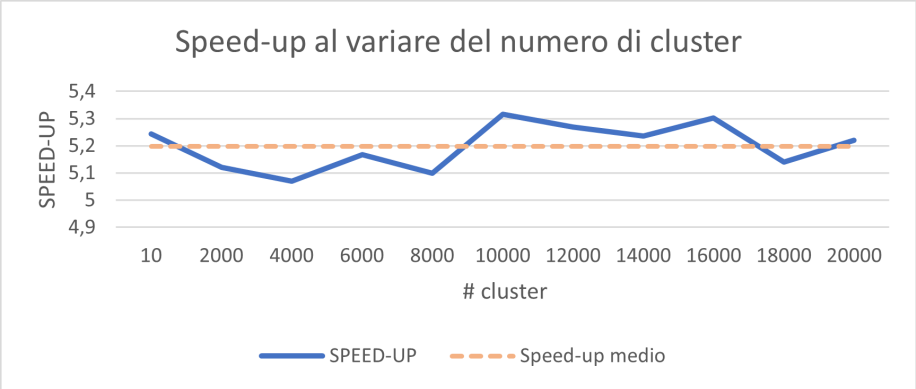

Figure 11: Grafico relativo allo speed-up al variare del numero di centroidi