K-means OpenMP

Giovannoni Alberto

Abstract

In questa relazione, esploriamo l'implementazione di K-means utilizzando OpenMP, una libreria per la programmazione parallela su architetture con memoria condivisa. L'obiettivo principale è confrontare le prestazioni fra versione sequenziale e parallela.

1. Introduzione

Questo elaborato ha lo scopo di mostrare il confronto fra la versione sequenziale di K-means e la versione parallela, ottenuta tramite l'uso di OpenMP. Per K-means sequenziale è stata implementata sia una versione SoA (Structure of Array) che una AoS (Array of Structure) per confrontarne le prestazioni. In questo progetto, K-means opera su punti 2D generati tramite distribuzioni gaussiane attorno ai propri centri.

2. Struttura del Codice

All'interno del progetto sono contenuti tre programmi:

- **sequential.cpp:** contiene la versione AoS sequenziale di K-means
- **structure_of_array.cpp:** contiene la versione SoA sequenziale di K-means
- **parallel.cpp:** contiene la versione parallela OpenMP di K-means

Inoltre è presente una cartella chiamata *"dataset"* che contiene due file:

- **centroids.txt:** dove vengono salvate le coordinate iniziali dei centroidi
- **dataset.txt:** dove vengono salvate le coordinate dei punti del dataset

2.1. Creazione dataset

Per creare il dataset è stato implementato un metodo in sequential.cpp chiamato *generateDataset()* (Figura 1) che si occupa di inserire le coordinate dei punti in dataset.txt. Il metodo prende in ingresso: il numero di cluster da creare, il numero di punti per cluster e la distanza minima fra i cluster. Come prima cosa vengono generati i centri dei cluster, controllando che non siano troppo vicini fra di loro. Una volta trovata una posizione per ciascun cluster, viene usata una distribuzione gaussiana per creare i punti e salvarli nel file. La scelta di usare una distribuzione gaussiana è stata presa perché K-means opera meglio con cluster di forma globulare.

```cpp
void generateDataset(int numPointsPerCluster, int numClusters, int
    centerDist, int stdDev, int rangeX, int rangeY) {
    random_device rd;
    mt19937 gen(rd());
    ofstream outFile("../dataset/dataset.txt");
    vector<point> centers;

    // Genera posizioni casuali per i centroidi
    uniform_real_distribution<double> cenX(-rangeX, rangeX);
    uniform_real_distribution<double> cenY(-rangeY, rangeY);

    for (int i = 0; i < numClusters; ++i) {
        point center;
        bool validPos = false;

        while (!validPos) {
            double x = cenX(gen);
            double y = cenY(gen);
            center.x = x;
            center.y = y;
            validPos = true;

            for (const auto &existingCenter: centers) {
                if (sqrt(pow(existingCenter.x - x, 2) +
                         pow(existingCenter.y - y, 2))
                         < centerDist) {
                    validPos = false;
                    break;
                }
            }
        }
        centers.push_back(center); // Lista centroidi validi
        // Genera punti intorno al centroide con distribuzione normale
        normal_distribution<double> disX(center.x, stdDev);
        normal_distribution<double> disY(center.y, stdDev);

        for (int j = 0; j < numPointsPerCluster; ++j) {
            double x = disX(gen);
            double y = disY(gen);
            outFile << x << " " << y << endl;
        }
    }
    outFile.close();
}
```

Figure 1. Metodo per la generazione del dataset

```cpp
void createClusters(int numCluster, int rangeX, int rangeY, int centerDist)
    {
    random_device rd;
    mt19937 gen(rd());
    ofstream outFile("../dataset/centroids.txt");
    // Creazione centroidi
    uniform_real_distribution<double> disX(-rangeX, rangeX);
    uniform_real_distribution<double> disY(-rangeY, rangeY);
    point c{};
    vector<point> centroids;

    for (int i = 0; i < numCluster; i++) {
        bool validPosition = false;
        while (!validPosition) {
            c.x = disX(gen);
            c.y = disY(gen);
            validPosition = true;
            for (auto &centroid: centroids) {
                double distance = sqrt(pow(centroid.x - c.x, 2) +
                                       pow(centroid.y - c.y, 2));
                if (distance < centerDist) {
                    validPosition = false;
                    break;
                }
            }
        }
        outFile << c.x << " " << c.y << endl;
        centroids.push_back(c);
    }
    outFile.close();
}
```

Figure 2. Metodo per la generazione dei centroidi

Per creare i cluster è stato implementato un secondo metodo chiamato *createClusters()* (Figura 2), simile a *generateDataset()*, che si occupa di riempire il file *centroids.txt* con le coordinate iniziali dei centroidi. Anche in questo caso i centroidi vengono generati in modo che siano sufficientemente distanti fra loro.

Questi due metodi vengono utilizzati per riempire i file .txt, in modo che gli esperimenti siano replicabili. Attraverso l'uso di variabili booleane nel main (*changeDataset* e *changeCentroids*), è infatti possibile disabilitare l'aggiornamento del

```cpp
do {
    centerUpdated = false;
    i++;
    for (auto &cluster: clusters) {
        // reset valori in ogni cluster
        cluster.resetTotalX();
        cluster.resetTotalY();
        cluster.resetCountPoints();
    }
    for (auto &point: points) {
        minDist = INFINITY;
        for (int j = 0; j < clusters.size(); j++) {
            dist = sqrt(pow(clusters[j].getCentroid().x - point.x, 2) +
                    pow(clusters[j].getCentroid().y - point.y, 2));
            if (dist < minDist) {
                minDist = dist;
                minIndex = j;
            }
        }
        if (point.clusterID != minIndex) {
            centerUpdated = true;
        }
        point.clusterID = minIndex;
        clusters[minIndex].addTotalX(point.x);
        clusters[minIndex].addTotalY(point.y);
        clusters[minIndex].countPoints();
    }
    if (centerUpdated) {
        for (auto &cluster: clusters) {
            cluster.updateCentroid();
        }
    }
} while (centerUpdated && i <= maxIter);
```

Figure 3. Ciclo do-while K-means sequenziale

```cpp
void updateCentroid() {
    if (count > 0) {
        centroid.x = totalX / count;
        centroid.y = totalY / count;
    } else {
        // Se non ci sono punti assegnati al cluster, mantieni il centroide
    invariato
    }
}
```

Figure 4. Metodo di aggiornamento centroidi

dataset e dei centroidi.

2.2. K-means Sequenziale

La versione sequenziale di K-means è caratterizzata dall'uso di un ciclo do-while che termina nel momento in cui nessun punto cambia cluster, oppure se si supera una soglia *maxIter* di iterazioni.

Come si vede nella figura 3, il ciclo do-while si occupa di assegnare a ciascun punto l'ID del cluster a cui è più vicino, e successivamente di aggiornare la posizione del centroide di ciascun cluster. Questo viene fatto attraverso il metodo *updateCentroid()* che è definito nella classe Cluster. Nella figura 4 se ne può vedere l'implementazione.

Ogni cluster possiede tre variabili:

- **totalX:** somma delle x di tutti i punti nel cluster
- **totalY:** somma delle y di tutti i punti nel cluster
- **count:** numero totale di punti nel cluster

Il metodo *updateCentroid()* calcola quindi la media delle x e delle y, trovando le nuove coordinate del centroide.

2.3. K-means Sequenziale con SoA

Per K-means sequenziale è stata implementata anche una versione Structure of Array, per valutare quale delle due soluzioni sia più efficiente. Il codice per questa versione si differenzia dalla precedente, unicamente per l'utilizzo di vettori per contenere le coordinate dei punti e dei centroidi, invece che le strutture dati *point* e *cluster*.

2.4. K-means Parallelo

La versione parallela di K-means è strutturata in modo analogo alla versione sequenziale AoS. Mi concentrerò quindi nel sottolineare come OpenMP viene usato.

Come si può vedere nella figura 5, le tre fasi principali di K-means sono parallelizzate usando *#pragma omp parallel for*. In questo modo, il calcolo delle distanze dai centroidi, l'aggiornamento delle coordinate dei centroidi ed il reset dei metadati nei vari cluster, sono completamente gestiti dai thread. Come evidenziato nei commenti nelle righe 29, 30 e 31 del codice, l'aggiornamento dei metadati nei cluster viene eseguito utilizzando istruzioni atomiche. Questo viene fatto per gestire la concorrenza quando più thread tentano di accedere o modificare contemporaneamente i valori di un cluster. Senza l'utilizzo delle istruzioni atomiche, potrebbe verificarsi una 'dirty read', nel quale un thread legge un valore non ancora aggiornato da un altro thread. Le istruzioni atomiche assicurano che l'aggiornamento dei valori avvenga come un unico blocco di istruzioni, risolvendo il problema.

Nella riga 26, durante l'aggiornamento del valore di *centerUpdated*, è interessante notare che non è necessario utilizzare istruzioni atomiche. Questo perché non è rilevante quale thread imposti il suo valore a *true*, e eventuali problemi di concorrenza come la dirty read non influenzerebbero l'esecuzione del codice. L'aggiunta di *#pragma omp atomic* per l'aggiornamento di

```
1     do {
2         centerUpdated = false;
3         // Reset dei valori nei cluster
4 #pragma omp parallel for
5         for (int c = 0; c < clusters.size(); c++) {
6             clusters[c].resetTotalX();
7             clusters[c].resetTotalY();
8             clusters[c].resetCountPoints();
9         }
10        int minIndex;
11        double minDist;
12        double dist;
13        // Assegnazione dei punti ai cluster piu' vicini
14 #pragma omp parallel for private(minIndex, minDist, dist)
15        for (int p = 0; p < points.size(); p++) {
16            minIndex = -1;
17            minDist = INFINITY;
18            for (int c = 0; c < clusters.size(); c++) {
19                dist = sqrt(pow(clusters[c].getCentroid().x - points[p].x,
      2) + pow(clusters[c].getCentroid().y - points[p].y, 2));
20                if (dist < minDist) {
21                    minDist = dist;
22                    minIndex = c;
23                }
24            }
25            if (points[p].clusterID != minIndex) {
26                centerUpdated = true;
27            }
28            points[p].clusterID = minIndex;
29            clusters[minIndex].addTotalX(points[p].x); // uso di atomic
30            clusters[minIndex].addTotalY(points[p].y); // uso di atomic
31            clusters[minIndex].countPoints(); // uso di atomic
32        }
33        // Aggiornamento dei centroidi
34 #pragma omp parallel for
35        for (int c = 0; c < clusters.size(); c++) {
36            clusters[c].updateCentroid();
37        }
38    } while (centerUpdated && i <= maxIter);
```

Figure 5. Ciclo do-while K-means parallelo

```
1     void addTotalX(double x) {
2 #pragma omp atomic
3         totalX += x;
4     }
5
6     void addTotalY(double y) {
7 #pragma omp atomic
8         totalY += y;
9     }
10
11    void countPoints() {
12 #pragma omp atomic
13        count++;
14    }
```

Figure 6. Metodi atomici per l'incremento dei metadati nella classe cluster

questo valore introdurrebbe solo un overhead inutile, senza alcun beneficio per l'esecuzione del programma.

In figura 6, sono mostrate le implementazioni dei tre metodi *addTotalX*, *addTotalY* e *countPoints* e della sincronizzazione ottenuta attraverso l'uso dell'istruzione *#pragma omp atomic*.

2.5. Correttezza del codice

Per verificare che tutte e 3 le versioni fossero corrette, sono state valutate sullo stesso dataset, con la stessa inizializzazione dei centroidi. Come risultato, esse sono terminate dopo lo stesso numero di iterazioni disegnando lo stesso grafico, che è stato riportato in figura 7.

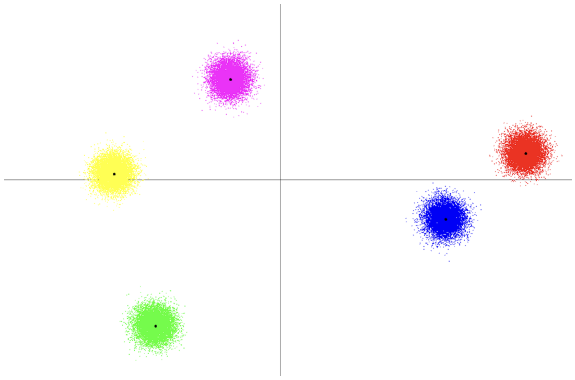

Figure 7. Grafico K-means

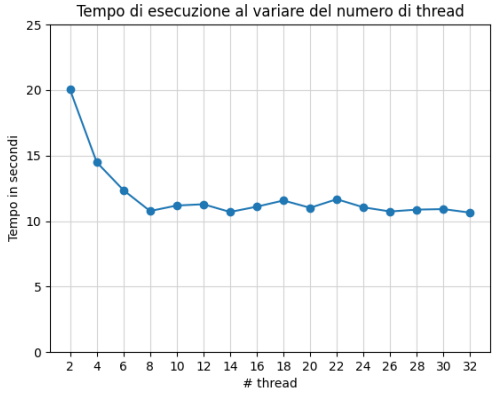

Figure 8. Tempo di esecuzione al variare del numero di thread

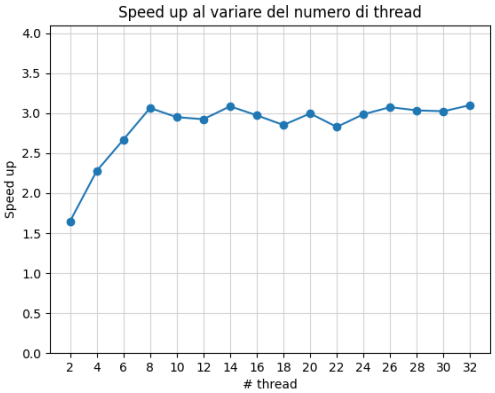

Figure 9. Speed up al variare del numero di thread

3. Sperimentazione

La fase di sperimentazione si è composta di 4 diversi esperimenti, mirati a confrontare la versione sequenziale di K-means con quella parallelizzata con OpenMP. Come ultimo esperimento è stato fatto un confronto fra le due versioni sequenziali (AoS e SoA).

Tutti gli esperimenti sono stati eseguiti su un pc dotato di una CPU Intel® Core™ i7-8565U con 4 cores fisici e 8 thread logici.

3.1. Sequenziale vs Parallela al variare del numero di thread

In questo esperimento si vuole confrontare il tempo di esecuzione della versione parallela di K-means, aumentando progressivamente il numero di thread, con la versione sequenziale. L'esperimento è stato condotto con 10 cluster contenenti 100.000 punti ciascuno e arrestando l'esecuzione alla 20esima iterazione.

Come si può osservare nella figura 8, il tempo di esecuzione decresce velocemente all'aumentare del numero di thread impiegati, ma si arresta dopo gli 8 thread. Questo è dato dal fatto che la CPU utilizzata dispone appunto di 8 thread logici.

Per il calcolo dello speed up, è stata eseguita una run della versione sequenziale, ed il tempo impiegato è stato di circa 33 secondi. É stato quindi possibile rappresentare il grafico in figura 9 calcolando lo speed up con la seguente formula:

$$\text{Speedup} = \frac{T_{\text{sequenziale}}}{T_{\text{parallelo}}}$$

Grazie a questo grafico si può osservare che in corrispondenza degli 8 thread lo speed up si assesta ad un valore di 3.

3.2. Sequenziale vs Parallela al variare del numero di cluster e del numero di punti

In questi esperimenti si vogliono valutare le differenze fra la versione sequenziale e quella parallela al variare del numero di cluster e del numero di punti nei cluster.

3.2.1 Variare il numero di cluster

In questo esperimento sono stati valutati il tempo di esecuzione e lo speed up della versione parallela rispetto a quella sequenziali, utilizzando un

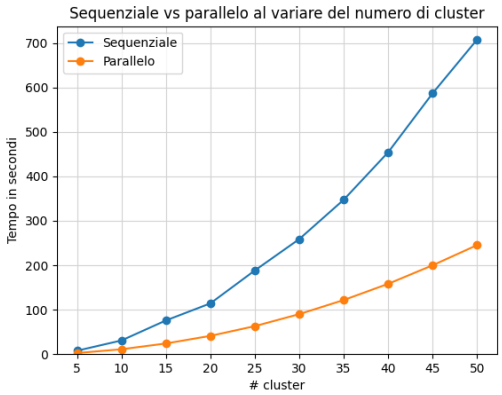

Figure 10. Tempo di esecuzione al variare del numero di cluster

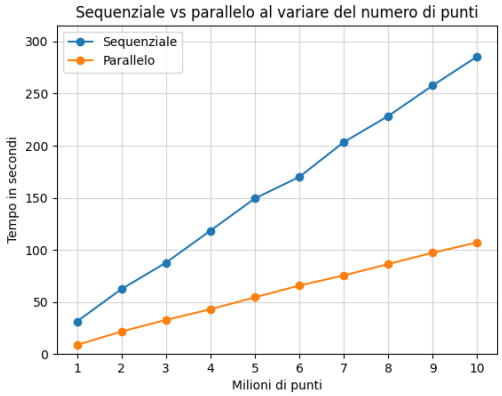

Figure 12. Tempo di esecuzione al variare del numero di punti

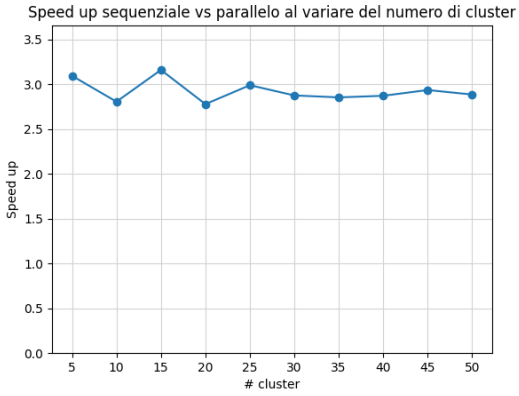

Figure 11. Speed up al variare del numero di cluster

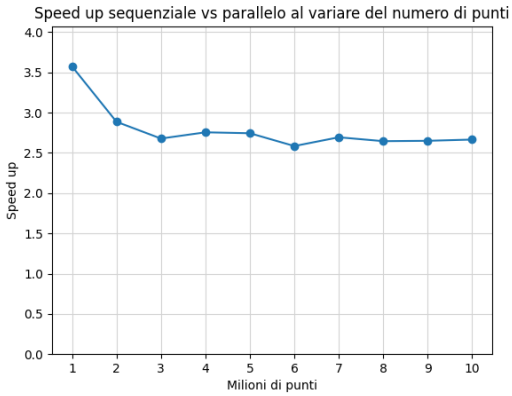

Figure 13. Speed up al variare del numero di punti

numero crescente di cluster. Ogni cluster contiene 100.000 punti e la versione parallela è stata eseguita usando 8 thread.

Come si vede dalla figura 10, la versione parallela risulta molto più efficiente.

Per vedere chiaramente il guadagno della versione parallela rispetto a quella sequenziale, è stato calcolato lo speed up per ogni rilevazione, ottenendo il grafico in figura 11. Anche in questo esperimento si può vedere come lo speed up si mantiene intorno al valore 3, con una media di 2.92.

3.2.2 Variare il numero di punti

In questo esperimento sono state fatte le stesse valutazioni di quello precedente, variando però il numero di punti per cluster, mantenendo 10 cluster per esperimento. Anche in questo caso, nella versione parallela sono stati usati 8 thread. La figura 12 racchiude i risultati del confronto nei tempi di esecuzione fra le due versioni di K-means, e risulta chiaro che la versione parallela è più veloce, come atteso.

In figura 13 si osserva come varia lo speed up nei vari casi ed è interessante notare come questo raggiunga un picco di 3.5 proprio per il primo valore (1.000.000 di punti). Si può quindi ipotizzare che la versione parallela risulti più efficiente con un numero minore di punti per cluster. In questo caso, la media dello speed up risulta più bassa, intorno a 2.79.

3.3. AoS vs SoA

In questa fase si vuole porre l'attenzione su quale delle due implementazioni sequenziali sia la più efficiente per il problema K-means.

Figure 14. Tempo di esecuzione variando il numero di cluster

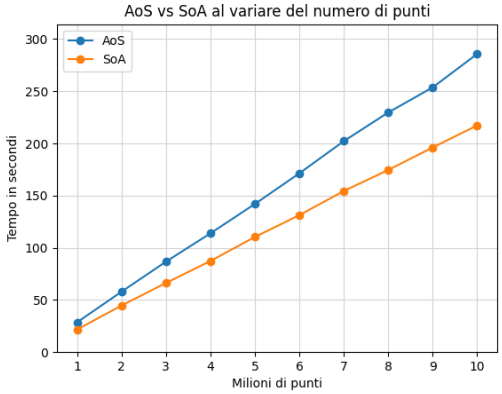

Figure 15. Tempo di esecuzione variando il numero di cluster

3.3.1 Tempo di esecuzione variando il numero di cluster

In questo esperimento entrambe le versioni sequenziali sono state valutate incrementando progressivamente il numero di cluster, mantenendo costante il numero di punti in ciascun cluster (100.000).

Come si può vedere dalla figura 14, la versione SoA risulta migliore di quella AoS. Questo può essere spiegato dal fatto che, nel calcolo delle distanze fra i punti e i centroidi, tutto l'array dei centroidi viene salvato in cache, rendendo più veloce il calcolo.

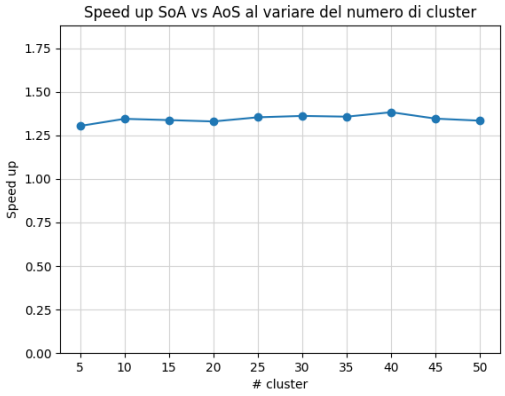

Figure 16. Tempo di esecuzione variando il numero di cluster

3.3.2 Tempo di esecuzione variando il numero di punti

In questo esperimento entrambe le versioni sequenziali sono state valutate incrementando progressivamente il numero di punti per cluster, mantenendo costante il numero di cluster (10).

Anche in questo caso, come si può vedere in figura 15, la versione SoA risulta più efficiente. Di seguito, nelle figure 16 e 17 sono riportati i grafici degli speed up relativi a questi due esperimenti e si può notare che i due andamenti sono molto simili.

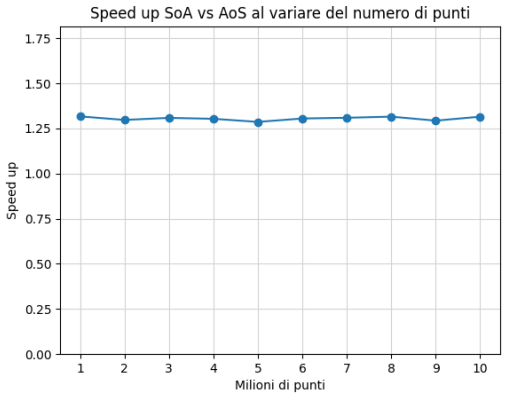

Figure 17. Tempo di esecuzione variando il numero di cluster

Si può quindi concludere che per l'algoritmo K-means, la versione SoA è la scelta migliore in quanto fornisce uno speed up medio di circa 1.3.

4. Conclusioni

Si può concludere che per quanto riguarda il confronto fra la versione sequenziale e quella parallela, lo speed up complessivo è circa 3, prestando però attenzione ai casi in cui si hanno molti

punti per cluster, dove questo valore tende a scendere.

Per quanto riguarda il confronto fra le versioni SoA e AoS, la prima risulta leggermente migliore, viste le numerose letture delle coordinate che devono essere fatte per calcolare le distanze dai centroidi.