# Clustering K-Means in OpenMP

Mattia Bacci
Matricola 7060637
mattia.bacci@stud.unifi.it

## Abstract

*Questo elaborato verte sull'implementazione di un applicativo che effettua Clustering K-Means utilizzando OpenMp partendo da una distribuzione casuale di punti e Cluster. Sono state analizzate le performance, attraverso il calcolo di speed-up ed efficienza,in un intel-i5 con 4 core, confrontando la versione sequenziale con quella parallela.*

## 1. Introduzione

### 1.1. Clustering K-Means

Il K-Means è un algoritmo di apprendimento non supervisionato che prevede di suddividere uno spazio dato in K parti denominati Cluster, ognuno di essi raggruppa più oggetti(nel nostro caso punti in 2D) in base ad una certa proprietà. Per ogni Cluster si definisce un centroide ossia un punto immaginario al centro del Cluster.

### 1.2. L'algoritmo

L'algoritmo procede nel seguente modo:

1. Si generano N punti casuali nello spazio bidimensionale(di dimensione finita scelta).

2. Si generano K centroidi casuali per i K Cluster voluti.

3. Si calcola la distanza di ogni punto con tutti i K centroidi e si assegna il punto al Cluster più vicino, con il termine distanza ci si riferisce alla distanza euclidea.

4. La posizione del centroide viene ricalcolata all'interno del cluster come media della posizioni dei punti appartenente al cluster stesso.

5. Si ripete i punti 3 e 4 fino alla condizione di interruzione.

```
Input:
   D= {t1, t2, …. Tn }   // Set of elements
   K                     // Number of desired clusters
Output:
   K                     // Set of clusters
K-Means algorithm:
  Assign initial values for m1, m2,.... mk
  repeat
    assign each item ti   to the clusters which has the closest mean;
    calculate new mean for each cluster;
  until convergence criteria is met;
```

Figure 1. Pseudocodice K-Means

### 1.3. Convergenza e Interruzione

E' in algoritmo che converge velocemente ma non è presente la garanzia di trovare l'ottimo globale. Ci sono varie condizioni per stabilire la convergenza:

- I centroidi non si muovono più.
- Nessun punto cambia cluster.
- La somma delle distanze è ridotta al minimo.

In questo elaborato è stata scelta la prima condizione, un centroide non si muove se la differenza della sua posizione prima e dopo l'aggiornamento della stessa risulta minore di $\epsilon = 10^{-1}$.

Visto che tali condizioni possono non soddisfarsi è necessaria introdurre inoltre una condizione di interruzione basata su un numero massimo di iterazioni.

## 2. Implementazione

### 2.1. Parametri

I paramentri selezionabili risultano:

- Numero di Punti e Cluster.
- Range Massimo, indica la dimensione del quadrato sul quale verranno generati casualmente i punti e i cluster, per i test è un valore fisso uguale a 100000.
- Numero di iterazioni massime, fissato a 100.

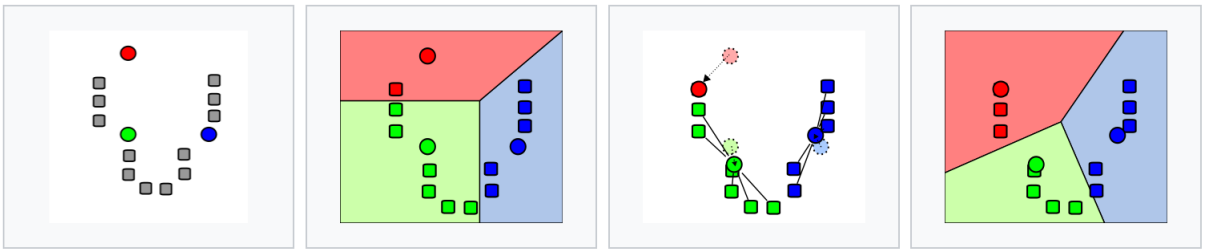

Figure 2. Passi algoritmo

### 2.2. Classi

Sono state create 2 classi principali:

1. **Cluster:** gli oggetti Cluster sono composti dalle coordinate *x* e *y* dei loro centroidi e da un attributo *size* indicante il numero di punti assegnati a tale Cluster. Troviamo inoltre le funzioni *addPoint(Point)* per assegnare un nuvo punto, *releasePoints* per svuotare il Cluster e *update* per aggiornargli la posizione.

2. **Point:** gli oggetti Point sono composti dalle loro coordinate *x* e *y* e da un valore intero rappresentante del Cluster ai quali appartengono. Durante l'inizializzazione casuale dei punti essi sono tutti assegnati al Cluster 0.

## 3. Versione Sequenziale

Il versione sequenziale prevede l'implementazione dell'algoritmo come scritto nella sezione *1.2*. Per raggiungere l'intento sono state sviluppate due funzioni pricipali: *kMeansIteration* e *updateClusters*

### 3.1. kMeansIteration

```
void kMeansIteration(std::vector<Point> &points,  std::
     vector<Cluster> &clusters) {
    int clusterIndex;
    double minDist;

    for (int i = 0; i < POINTS_NUMBER; i++) {
        Point &point = points[i];
        clusterIndex = 0;
        minDist = distance(point, clusters[0]);

        for (int j = 1; j < CLUSTERS_NUMBER; j++) {
            Cluster &cluster = clusters[j];
            double dist = distance(point, cluster);
            if (dist < minDist) {
                minDist = dist;
                clusterIndex = j;
            }
        }
        clusters[clusterIndex].addPoint(points[i]);
        points[i].setCluster(clusterIndex);
    }
}
```

La funzione prende in input due oggetti vector all'interno dei quali si trovano i punti e i Cluster generati in modo casuale, per ogni punto si procede a calcolare la distanza euclidea da ogni centroide dei Cluster per trovarne la minima. Come passo finale viene aggiunto il punto al Cluster chiamando la funzione *addPoint(Point)* e settando sul punto il numero del Cluster di appartenenza.

### 3.2. updateClusters

```
bool updateClusters(std::vector<Cluster> &clusters) {
    bool convergence = true;

    for (int i = 0; i < clusters.size(); i++) {
        if (clusters[i].update()) {
            convergence = false;
        }
        clusters[i].releasePoints();
    }

    return convergence;
}
```

Come specificato nella sezione *1.3* abbiamo bisogno di una condizione di convergenza, la funzione *update()* dei Cluster verifica tale condizione per un singolo Cluster e ritorna *TRUE* se il Cluster si è spostato dalla posizione precedente. Quindi se anche un solo Cluster si muove dalla sua posizione la funzione *updateCluster* ritorna *FALSE* che corrisponde alla condizione di convergenza non verificata. Il codice chiamante continuerà l'esecuzione finchè la funzione non ritornerà *TRUE* o non si saranno raggiunte il numero massimo di iterazioni.

## 4. Codice Parallelo

La versione sequenziale del problema impiega la maggior parte del tempo a calcolare le distanze di tutti i punti dai Cluster in modo da trovare il Cluster a distanza minore. Basti pensare che per ogni iterazione ci sono da calcolare esattamente N * K distanze, con 2 milioni di punti e 15 Cluster si devono calcolare 30 milioni di distanze per iterazione.

Vista l'assenza di risorse condivise, il codice può essere parallelizzato su più core senza necessità di alcuna sincronizzazione(se non con l'atomicità di qualche operazione). L'integrazione alla paralelizzazione con OpenMP permette di lasciare il codice scritto fino ora invariato, si aggiungono solamente le direttive OpenMP.

| Punti | Cluster | Tempo Sequenziale | Tempo MultiThread | SpeedUp | Efficienza |
|---|---|---|---|---|---|
| 100000 | 10 | 7100 ms(*) | 2300 ms(*) | 3 | 75% |
| 100000 | 20 | 15500 ms | 5200 ms | 3 | 75% |
| 100000 | 30 | 23200 ms | 7200 ms | 3.2 | 80% |
| 500000 | 10 | 37700 ms(*) | 11800 ms(*) | 3.2 | 80% |
| 500000 | 20 | 78300 ms | 23600 ms | 3.3 | 82% |
| 500000 | 30 | 117200 ms | 34600 ms | 3.4 | 85% |
| 1000000 | 10 | 82200 ms | 24400 ms | 3.4 | 85% |
| 1000000 | 20 | 157100 ms | 45900 ms | 3.4 | 85% |
| 1000000 | 30 | 233800 ms | 68300 ms | 3.4 | 85% |

Figure 3. Intel Core i5 (I5-4570S) 4-Core

```
void kMeansIterationParallel(std::vector<Point> &points,
      std::vector<Cluster> &clusters) {
    int clusterIndex;
    double minDist;

#pragma omp parallel default(shared) private(minDist,
     clusterIndex)
    {
#pragma omp for schedule(static)
        for (int i = 0; i < POINTS_NUMBER; i++) {
            Point &point = points[i];
            clusterIndex = 0;
            minDist = distance(point, clusters[0]);

            for (int j = 1; j < CLUSTERS_NUMBER; j++) {
                Cluster &cluster = clusters[j];
                double dist = distance(point, cluster);
                if (dist < minDist) {
                    minDist = dist;
                    clusterIndex = j;
                }
            }
            clusters[clusterIndex].addPoint(points[i]);
            points[i].setCluster(clusterIndex);
        }
    }
}
```

E' stato applicato un *parallel for* al ciclo esterno lasciando i vector dei punti e dei Cluster condivisi e le variabili *minDist* e *clusterIndex* private in tutti i Thread. Da notare che benché non sia richiesta alcuna sincronizzazione più punti su diversi Thread potrebbero chiamare la funzione *addPoint(Point)* contemporaneamente. Tale funzione incrementa la variabili temporanee di x e y(attraverso le quali verrà ricalcolata la posizione del centroide) e incrementa il *size* di 1. Tali operazione devono essere rese atomiche all'interno del processore per non generare risultati errati.

```
 void addPoint(const Point &point) {
#pragma omp atomic
        tmpX += point.getX();
#pragma omp atomic
        tmpY += point.getY();
#pragma omp atomic
        size++;
    }
```

### 4.1. Altre ottimizzazioni

Le fasi di inizializzazione dei punti e dei Cluster sono state inserite in due *section* diverse, l'obiettivo non è quello di velocizzare la computazione in quanto Essendo il numero di Cluster molto inferiore rispetto ai punti il guadagno è praticamente nullo. Si è voluto solamente solo mostrare un esempio di applicazione delle *section*.

## 5. Analisi

Per l'analisi delle performance sono stati variati sia il numero di punti che di Cluster. I test sono stati effettuati con un processore Intel Core i5 (I5-4570S) con 4-Core. I risultati sono presenti nella Fig. 3.

### 5.1. Note

Per poter visualizzare il risultato finale è necessaria la libreria gnuplot e la libreria imagemagick. Oltre a poter generare un immagine statica, si può attivare le animazioni in modo da generare una gif che mostra l'esecuzione dell'algoritmo, il tempo di esecuzione della gif risulta uguale al tempo realmente impiegato dal programma per portare al termine la computazione.