
UNIVERSITÀ
DEGLI STUDI
FIRENZE

# K-means

## Progetto di Parallel Computing

a cura di Jacopo Manetti

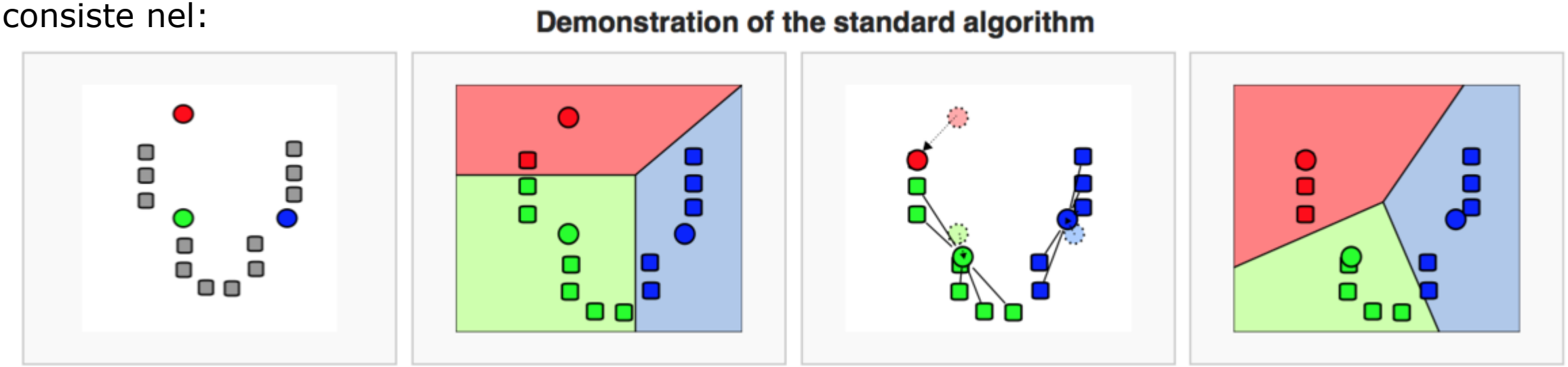
UNIVERSITÀ DEGLI STUDI FIRENZE

# Il Progetto

Lo scopo di questo progetto è stato quello di confrontare le prestazioni dell'algoritmo k-means in versione parallela con quella sequenziale, utilizzando il linguaggio di programmazione **C++** e la libreria **OpenMP**, al fine di valutare l'efficienza dell'implementazione parallela sull'hardware specifico, ovvero un computer dotato di un processore **Intel i7 11800h a 8 core** (16 logici). I risultati hanno dimostrato che l'utilizzo della versione parallela ha permesso di ottenere uno speed-up medio di circa 6,5 rispetto alla versione sequenziale, dimostrando l'efficacia delle tecniche di programmazione parallela nell'ottimizzare i tempi di esecuzione di un algoritmo. Il progetto ha anche fornito una panoramica delle tecniche di programmazione parallela e dei loro vantaggi nell'ambito del data clustering.

# L’algoritmo k-means

L'algoritmo k-means è un algoritmo di clustering utilizzato per dividere un insieme di punti in gruppi (cluster) basati sulle loro similarità. Il funzionamento dell'algoritmo consiste nel:




1.Scegliere il numero di cluster k che si vuole creare e selezionare casualmente k punti nell'insieme di punti da clusterizzare come centroidi iniziali.

2.Assegnare ciascun punto al centroide più vicino.

3.Calcolare la media di ogni gruppo (cluster) e utilizzarla per spostare il centroide del cluster corrispondente.

4.Ripetere i passi 2 e 3 finché i centroidi smettono di spostarsi.

# Il codice sequenziale - point

```
class Point {

public:

    Point(double x_coord, double y_coord){
        this->x_coord = x_coord;
        this->y_coord = y_coord;
        id_cluster = 0;
    }

    Point(){
        x_coord = 0;
        y_coord = 0;
        id_cluster = 0;
    }

    double get_x_coord(){
        return this->x_coord;
    }

    double get_y_coord(){
        return this->y_coord;
    }

    int get_id_cluster(){
        return id_cluster;
    }

    void set_id_cluster(int id_cluster){
        this->id_cluster = id_cluster;
    }
```

La classe "**Point**" rappresenta un punto nello spazio bidimensionale, descritto da due coordinate (x e y) e da un identificatore del cluster a cui appartiene.

# Il codice sequenziale - point

```
class Point {

public:

    Point(double x_coord, double y_coord){
        this->x_coord = x_coord;
        this->y_coord = y_coord;
        id_cluster = 0;
    }

    Point(){
        x_coord = 0;
        y_coord = 0;
        id_cluster = 0;
    }

    double get_x_coord(){
        return this->x_coord;
    }

    double get_y_coord(){
        return this->y_coord;
    }

    int get_id_cluster(){
        return id_cluster;
    }

    void set_id_cluster(int id_cluster){
        this->id_cluster = id_cluster;
    }
```

La classe "**Point**" rappresenta un punto nello spazio bidimensionale, descritto da due coordinate (x e y) e da un identificatore del cluster a cui appartiene.

La classe ha due costruttori: uno che richiede l'inizializzazione delle coordinate (x_coord e y_coord) e imposta l'id_cluster a 0, e l'altro che imposta tutte le variabili a 0.

# Il codice sequenziale - point

```
class Point {

public:

    Point(double x_coord, double y_coord){
        this->x_coord = x_coord;
        this->y_coord = y_coord;
        id_cluster = 0;
    }

    Point(){
        x_coord = 0;
        y_coord = 0;
        id_cluster = 0;
    }

    double get_x_coord(){
        return this->x_coord;
    }

    double get_y_coord(){
        return this->y_coord;
    }

    int get_id_cluster(){
        return id_cluster;
    }

    void set_id_cluster(int id_cluster){
        this->id_cluster = id_cluster;
    }
```

La classe "**Point**" rappresenta un punto nello spazio bidimensionale, descritto da due coordinate (x e y) e da un identificatore del cluster a cui appartiene.

La classe ha due costruttori: uno che richiede l'inizializzazione delle coordinate (x_coord e y_coord) e imposta l'id_cluster a 0, e l'altro che imposta tutte le variabili a 0.

La classe fornisce quattro metodi pubblici:

- "get_x_coord" restituisce le coordinate x del punto.
- "get_y_coord" restituisce le coordinate y del punto.
- "get_id_cluster" restituisce l'identificatore del cluster a cui il punto appartiene.
- "set_id_cluster" imposta l'identificatore del cluster a cui il punto appartiene.

# Il codice sequenziale - cluster

```
class Cluster {
public:
    Cluster(double x_coord, double y_coord) {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = x_coord;
        this->y_coord = y_coord;
    }

    Cluster() {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = 0;
        this->y_coord = 0;
    }

    void add_point(Point point) {
        points_x_coord += point.get_x_coord();
        points_y_coord += point.get_y_coord();
        size++;
    }

    void free_point() {
        this->size = 0;
        this->points_x_coord = 0;
        this->points_y_coord = 0;
    }
```

La classe "**Cluster**" rappresenta un cluster con un centroide, cioè un punto con coordinate x e y che rappresenta il centro del cluster.

# Il codice sequenziale - cluster

```
class Cluster {
public:
    Cluster(double x_coord, double y_coord) {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = x_coord;
        this->y_coord = y_coord;
    }

    Cluster() {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = 0;
        this->y_coord = 0;
    }

    void add_point(Point point) {
        points_x_coord += point.get_x_coord();
        points_y_coord += point.get_y_coord();
        size++;
    }

    void free_point() {
        this->size = 0;
        this->points_x_coord = 0;
        this->points_y_coord = 0;
    }
```

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.

# Il codice sequenziale - cluster

```
class Cluster {
public:
    Cluster(double x_coord, double y_coord) {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = x_coord;
        this->y_coord = y_coord;
    }

    Cluster() {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = 0;
        this->y_coord = 0;
    }

    void add_point(Point point) {
        points_x_coord += point.get_x_coord();
        points_y_coord += point.get_y_coord();
        size++;
    }

    void free_point() {
        this->size = 0;
        this->points_x_coord = 0;
        this->points_y_coord = 0;
    }
```

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.

**add point(Point point):** aggiunge il punto specificato al cluster e aggiorna le coordinate del centroide del cluster.

# Il codice sequenziale - cluster

```
class Cluster {
public:
    Cluster(double x_coord, double y_coord) {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = x_coord;
        this->y_coord = y_coord;
    }

    Cluster() {
        points_x_coord = 0;
        points_y_coord = 0;
        size = 0;
        this->x_coord = 0;
        this->y_coord = 0;
    }

    void add_point(Point point) {
        points_x_coord += point.get_x_coord();
        points_y_coord += point.get_y_coord();
        size++;
    }

    void free_point() {
        this->size = 0;
        this->points_x_coord = 0;
        this->points_y_coord = 0;
    }
```

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.

**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.

UNIVERSITÀ DEGLI STUDI FIRENZE

# Il codice sequenziale – cluster pt.2

```
double get_x_coord() {
    return this->x_coord;
}

double get_y_coord() {
    return this->y_coord;
}

// boolean method to check if the centroid has changed and if so update it
bool update_coords() {

    if (this->x_coord == points_x_coord / this->size && this->y_coord == points_y_coord / this->size) {
        return false;
    }

    this->x_coord = points_x_coord / this->size ;
    this->y_coord = points_y_coord / this->size ;

    return true;
}
```

**get x coord():** restituisce la coordinata x del centroide del cluster.

**get y coord():** restituisce la coordinata y del centroide del cluster.

**update coords():** Il metodo controlla se il centroide è cambiato rispetto alla sua precedente posizione.

Se il centroide è rimasto nella stessa posizione, il metodo restituisce false.

Altrimenti, il metodo calcola la media delle coordinate di tutti i punti nel cluster e aggiorna le coordinate del centroide con le coordinate medie calcolate. Infine, il metodo restituisce true per indicare che il centroide è stato aggiornato.

# Il codice sequenziale - main

Nel main i 2 metodi principali sono:

```
void calculateDistance (vector<Point> &points, vector<Cluster> &clusters){

    double min_distance;
    int index;

    for (int i = 0; i < points.size() ; ++i) {

        min_distance = euclideanDistance( point: points[i], cluster: clusters[0]);
        index = 0;

        for (int j = 0; j < clusters.size(); ++j) {

            double distance = euclideanDistance( point: points[i], cluster: clusters[j]);

                    if (distance < min_distance) {

                        min_distance=distance;
                        index = j;
                    }
        }

        points[i].set_id_cluster(index);
        clusters[index].add_point( point: points[i]);
    }

}
```

**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.

UNIVERSITÀ DEGLI STUDI FIRENZE

# Il codice sequenziale - main

Nel main i 2 metodi principali sono:

```
//update the centroid of clusters
bool updateClusters(vector<Cluster> &clusters){

    bool conv = false;



    for(int i = 0; i < clusters.size(); i++){

        conv = clusters[i].update_coords();
        clusters[i].free_point();

    }

    return conv;
}
```

**updateClusters():** Questo metodo prende in input un vettore Cluster. Il metodo esegue un ciclo su tutti i Cluster, e per ognuno di essi invoca il metodo update_coords(). Questo metodo calcola le nuove coordinate del centroide del cluster e restituisce un valore booleano che indica se le coordinate sono state adattate o meno. Il valore booleano restituito da update_coords()viene assegnato alla variabile conv.

Dopo aver chiamato update_coords(), il metodo free_point()viene invocato per liberare la lista dei punti del cluster.

Infine, il metodo ritorna il valore booleano conv, che indica se almeno un cluster ha avuto le coordinate del centroide adattate.

Se conv è false, significa che i centroidi non sono stati aggiornati e quindi l'algoritmo ha raggiunto la convergenza.

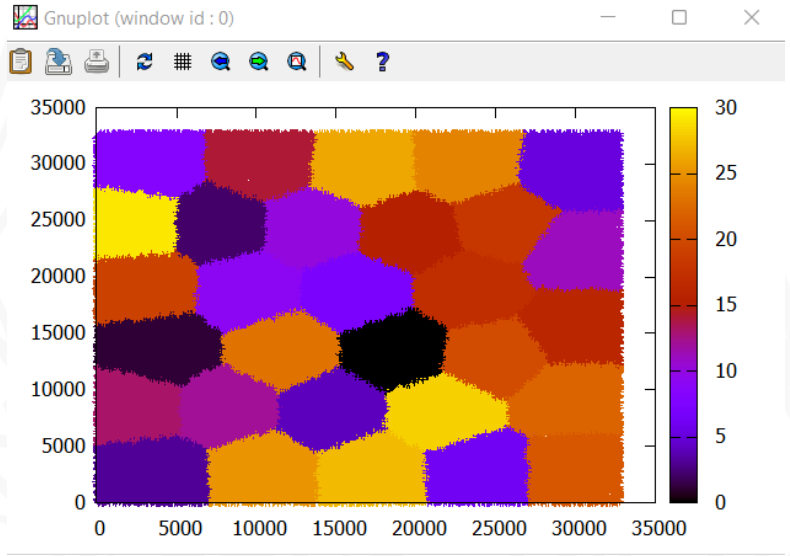
# Il codice sequenziale - main

```
int main() {

    srand( Seed: seed);

    printf( format "Enter the number of points to generate:");
    cin>>num_point;

    printf( format "Enter the number of centroids to generate:");
    cin>>num_cluster;

    printf( format "Number of points: %d\n", num_point);
    printf( format "Number of cluster: %d\n", num_cluster);

    double time1 = omp_get_wtime(); //start cronometer

    vector<Point> points = initializePoint(num_point);

    vector<Cluster> clusters = initializeCluster(num_cluster);

    double time2 = omp_get_wtime();
    printf( format "Points and Cluster are generated in %f seconds\n", time2-time1 );

    bool conv = true;

    while (conv){

        calculateDistance( & points, & clusters);
        conv = updateClusters( & clusters);
    }

    double time3 = omp_get_wtime();
    printf( format "Sequential k-means has taken %f seconds\n", time3-time2);


    printf( format "Drawing the chart...\n");
    drawPlot(points);
```

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().

Finchè conv è a true vengono calcolate le distanze dei punti dai centroidi e questi vengono aggiornati.

Dopo che è stata raggiunta la convergenza viene plottato il risultato utilizzando la libreria gnuplot.



UNIVERSITÀ DEGLI STUDI FIRENZE

# Il codice parallelo

La parte parallela del codice è gestita tramite l'utilizzo della libreria **OpenMP**

```
void calculateDistance (vector<Point> &points, vector<Cluster> &clusters){

    long points_size = points.size();
    long clusters_size = clusters.size();

    double min_distance;
    int index;

#pragma omp parallel default(shared) private(min_distance, index) firstprivate(points_size, clusters_size) num_threads(16)
    {
#pragma omp for schedule(static)

        for (int i = 0; i < points_size; ++i) {

            min_distance = euclideanDistance( point: points[i], cluster: clusters[0]);
            index = 0;

            for (int j = 0; j < clusters_size; ++j) {

                double distance = euclideanDistance( point: points[i], cluster: clusters[j]);

                if (distance < min_distance) {

                    min_distance = distance;
                    index = j;
                }
            }

            points[i].set_id_cluster(index);

            clusters[index].add_point( point: points[i]);
        }
    }
}
```

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.

# Il codice parallelo

La parte parallela del codice è gestita tramite l'utilizzo della libreria **OpenMP**

```
void calculateDistance (vector<Point> &points, vector<Cluster> &clusters){

    long points_size = points.size();
    long clusters_size = clusters.size();

    double min_distance;
    int index;

#pragma omp parallel default(shared) private(min_distance, index) firstprivate(points_size, clusters_size) num_threads(16)
    {
#pragma omp for schedule(static)

        for (int i = 0; i < points_size; ++i) {

            min_dis
            index =

            for (in

                do

                if

                }
            }

            points[i].set_id_cluster(index);

            clusters[index].add_point( point points[i]);
        }
    }
}
```

**default(shared):** specifica che le variabili che non vengono dichiarate come private o firstprivate saranno condivise tra i thread. In questo caso, tutti i thread avranno accesso alle variabili dichiarate come shared.

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.

# Il codice parallelo

La parte parallela del codice è gestita tramite l'utilizzo della libreria **OpenMP**

```
void calculateDistance (vector<Point> &points, vector<Cluster> &clusters){

    long points_size = points.size();
    long clusters_size = clusters.size();

    double min_distance;
    int index;

#pragma omp parallel default(shared) private(min_distance, index) firstprivate(points_size, clusters_size) num_threads(16)
    {
#pragma omp for schedule(static)

        for (int i = 0; i < points_size; ++i) {

            min_dis
            index =

            for (in

                dou

                if

                    min_distance = distance;
                    index = j;
                }
            }

            points[i].set_id_cluster(index);

            clusters[index].add_point( point points[i]);
        }
    }
}
```

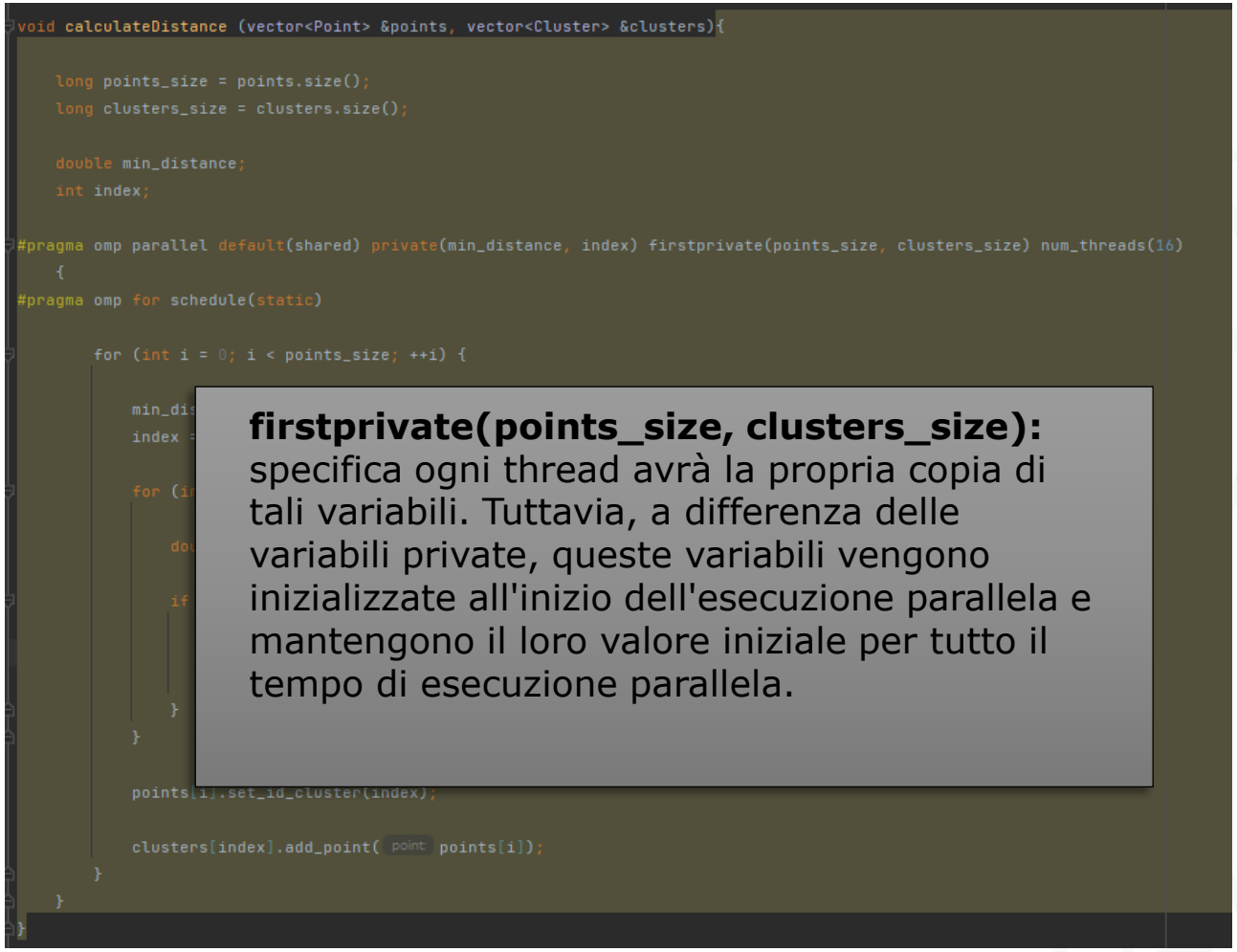
**private(min_distance, index):** specifica che le variabili saranno dichiarate come variabili private e ogni thread avrà la propria copia di tali variabili.

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.

UNIVERSITÀ DEGLI STUDI FIRENZE

# Il codice parallelo

La parte parallela del codice è gestita tramite l'utilizzo della libreria **OpenMP**




**firstprivate(points_size, clusters_size):** specifica ogni thread avrà la propria copia di tali variabili. Tuttavia, a differenza delle variabili private, queste variabili vengono inizializzate all'inizio dell'esecuzione parallela e mantengono il loro valore iniziale per tutto il tempo di esecuzione parallela.

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.

# Il codice parallelo

La parte parallela del codice è gestita tramite l'utilizzo della libreria **OpenMP**

```
void calculateDistance (vector<Point> &points, vector<Cluster> &clusters){

    long points_size = points.size();
    long clusters_size = clusters.size();

    double min_distance;
    int index;

#pragma omp parallel default(shared) private(min_distance, index) firstprivate(points_size, clusters_size) num_threads(16)
    {
#pragma omp for schedule(static)

        for (int i = 0; i < points_size; ++i) {

            min_di
            index =

            for (i

                do

                if (distance < min_distance) {

                    min_distance = distance;
                    index = j;
                }
            }

            points[i].set_id_cluster(index);

            clusters[index].add_point( point points[i]);
        }
    }
}
```

**num_threads(16):** specifica il numero di thread che verranno utilizzati per eseguire la sezione parallela.

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.

# Il codice parallelo

La parte parallela del codice è gestita tramite l'utilizzo della libreria **OpenMP**

```
void calculateDistance (vector<Point> &points, vector<Cluster> &clusters){

    long points_size = points.size();
    long clusters_size = clusters.size();

    double min_distance;
    int index;

#pragma omp parallel default(shared) private(min_distance, index) firstprivate(points_size, clusters_size) num_threads(16)
    {
#pragma omp for schedule(static)

        for (int i = 0; i < points_size; ++i) {

            min_distance = euclideanDistance( point
            index = 0;

            for (int j = 0; j < clusters_size; ++j)

                double distance = euclideanDistance

                if (distance < min_distance) {

                    min_distance = distance;
                    index = j;
                }
            }

            points[i].set_id_cluster(index);

            clusters[index].add_point( point points[i]);
        }
    }
}
```

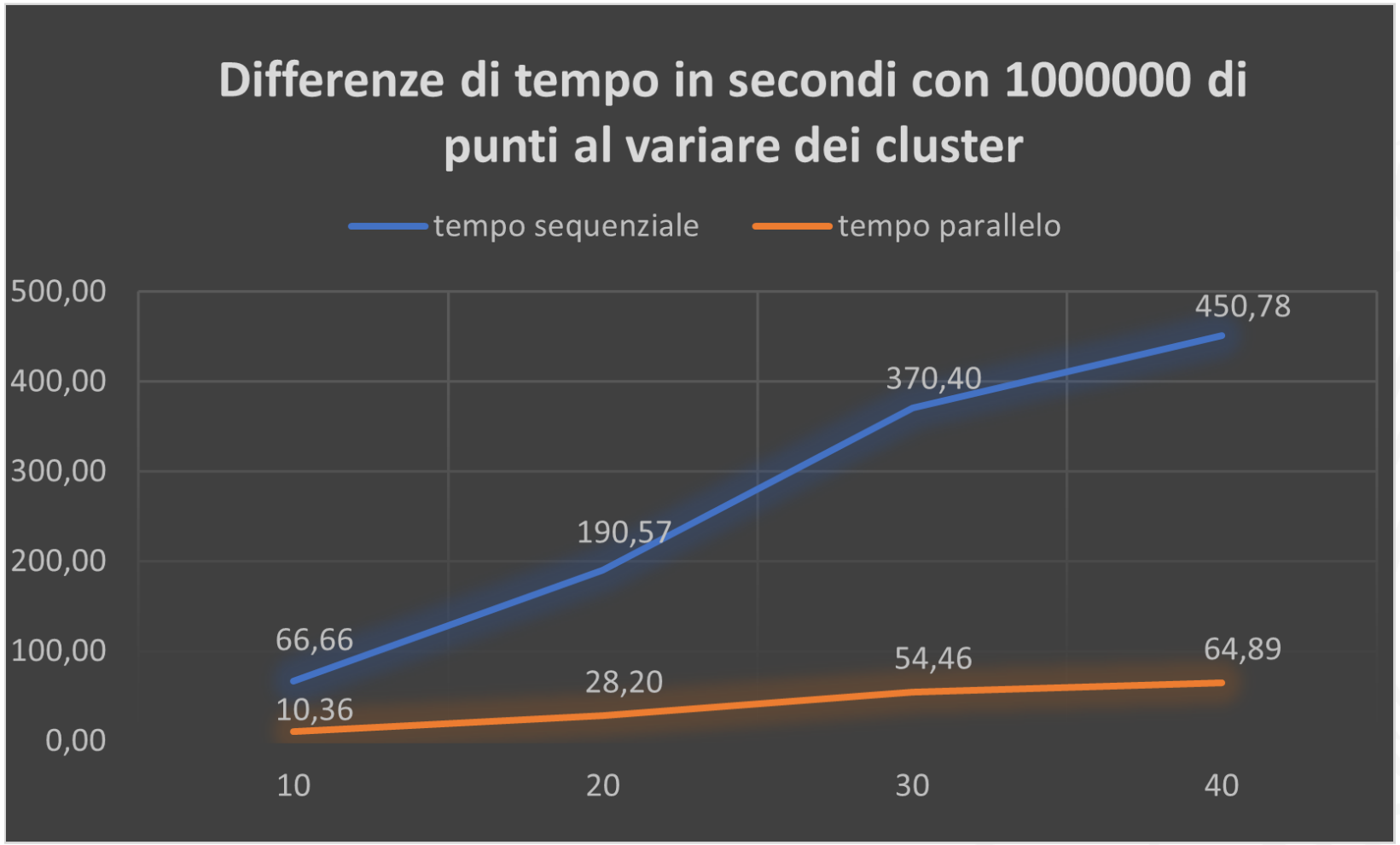
```
    void add_point(Point point) {
#pragma omp atomic
        points_x_coord += point.get_x_coord();
#pragma omp atomic
        points_y_coord += point.get_y_coord();
#pragma omp atomic
        size++;

    }
```

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.

Nel metodo **add_point**, le tre variabili points_x_coord, points_y_coord, e size sono condivise tra i thread e vengono aggiornate attraverso le direttive atomic per garantire che l'aggiornamento avvenga in modo consistente e senza interferenze.

UNIVERSITÀ DEGLI STUDI FIRENZE

# Analisi delle prestazioni




Si vede subito come l'esecuzione parallela sia efficace.

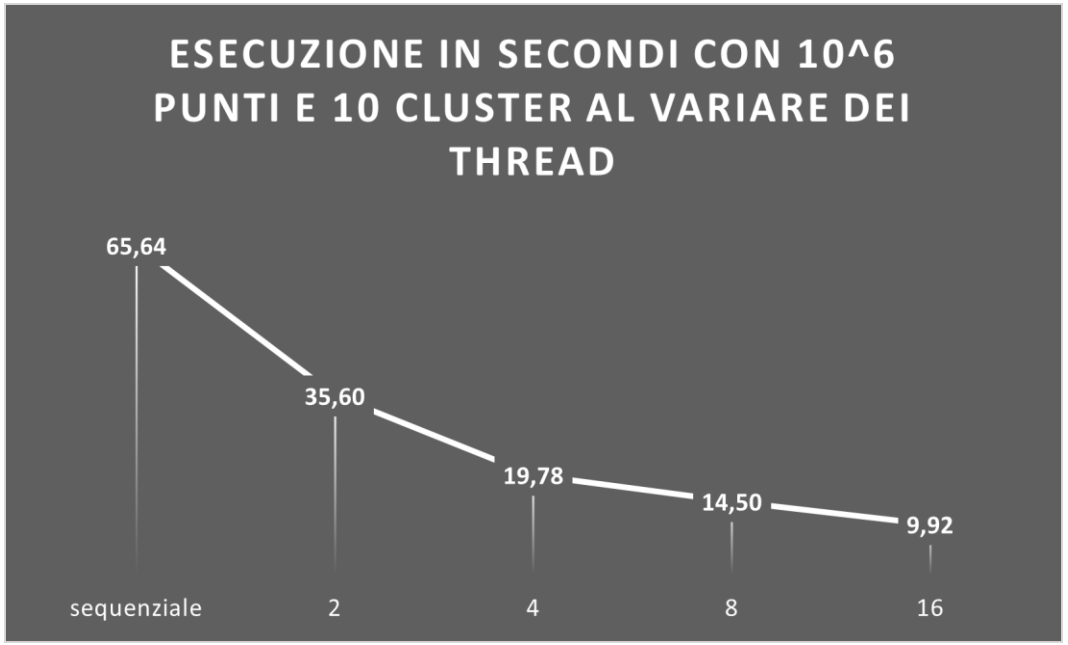
Variando il numero di cluster (10,20,30,40) si ottiene uno speed up di circa 6,5.

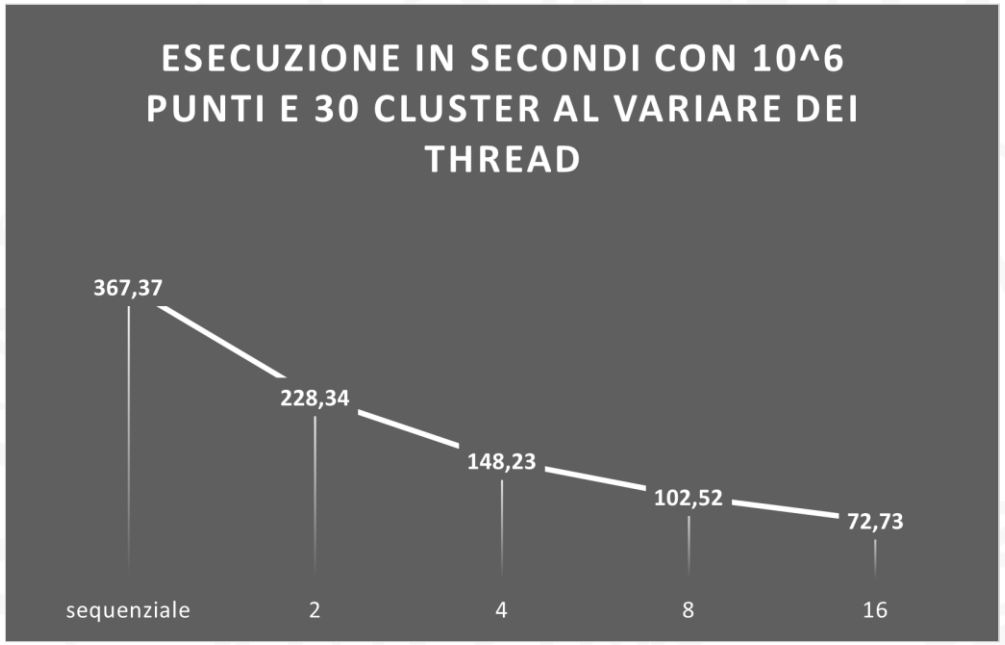
UNIVERSITÀ DEGLI STUDI FIRENZE

# Analisi delle prestazioni


Anche mantenendo fisso il numero dei cluster e variando il numero di punti lo speed up ottenuto è di circa 6.

# Analisi delle prestazioni

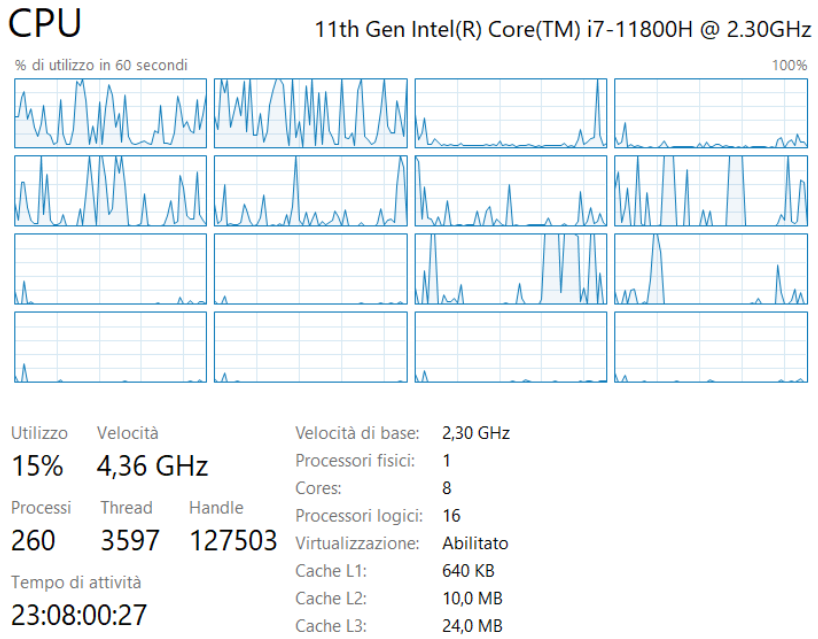


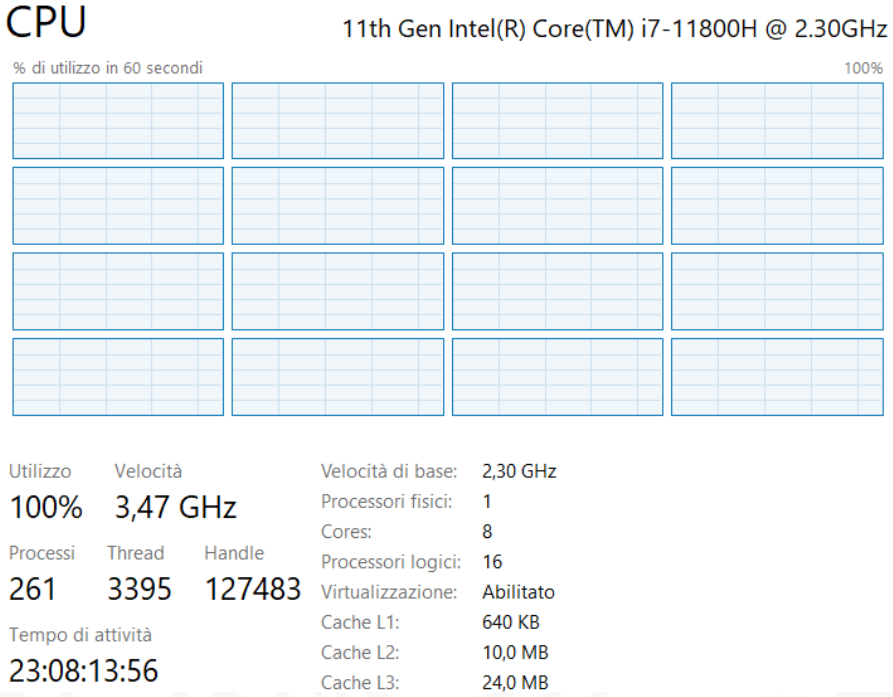

Esecuzioni ottenute modificando il numero di thread durante la parallelizzazione, come ci si aspetta all'aumentare del numero di thread disponibili l'esecuzione è più veloce.

# Conclusione

In conclusione 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.




***CPU durante l'esecuzione sequenziale***




***CPU durante l'esecuzione parallela***

UNIVERSITÀ DEGLI STUDI FIRENZE

# Fine