



# Parallel KMeans

Parallel Programming for Machine Learning

Niccolò Arati

# Introduzione

Il programma scritto per questo progetto si occupa di implementare **Kmeans**, un algoritmo di clustering basato su apprendimento non supervisionato. Dato un dataset di punti e un numero **K** di cluster, KMeans assegna ogni punto ad uno dei K cluster, minimizzando l'errore quadratico medio.

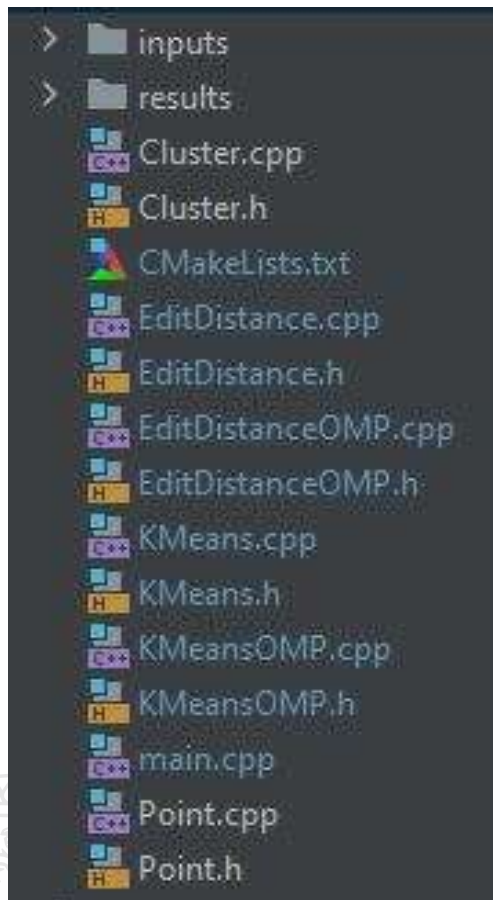
La **convergenza** dell'algoritmo è stata impostata tramite un numero massimo di iterazioni, 35, e tramite la percentuale di punti che variano di cluster dopo ogni iterazione. Se questa percentuale scende sotto lo 0.1%, l'algoritmo termina.

Sono presenti due implementazioni di KMeans: una **sequenziale** e una **parallela**. Lo scopo di questo elaborato è quello di osservare lo **speedup** ottenuto con la versione parallela rispetto a quella sequenziale.

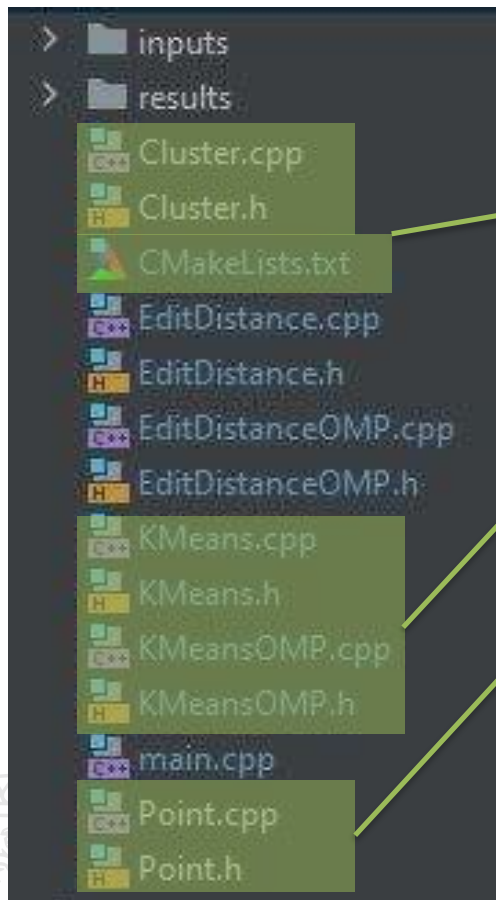
Il linguaggio di programmazione utilizzato è il C++ (compilatore MinGW, IDE CLion) e la parallelizzazione è stata svolta con **OpenMP**.

Tutti gli esperimenti sono stati svolti sul server ssh "papavero.dinfo.unifi.it".

## Organizzazione del codice

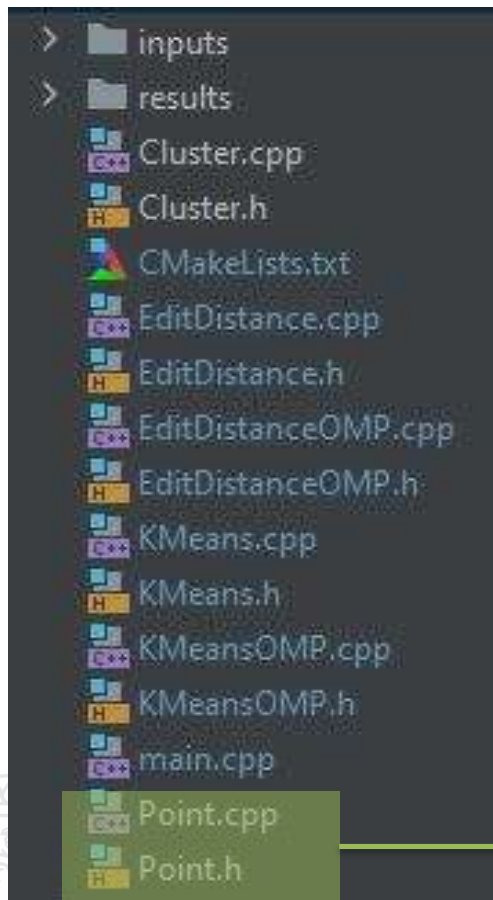


## Organizzazione del codice



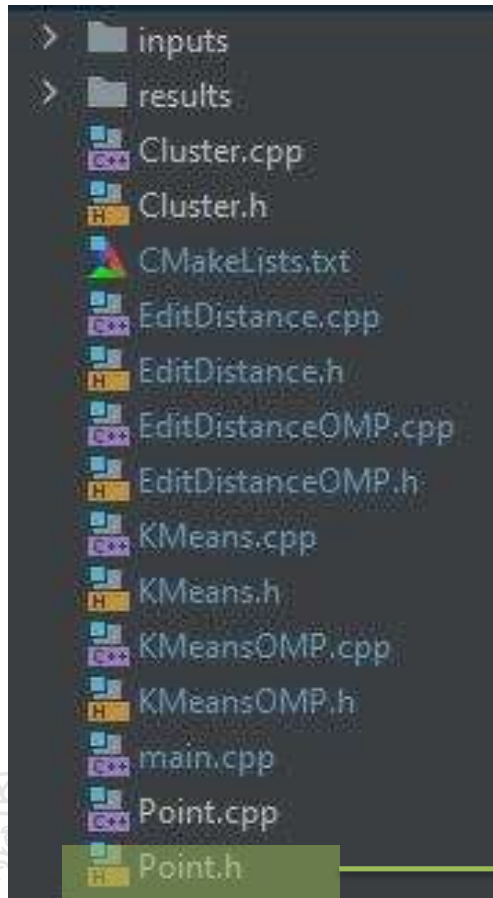
File compresi in questo progetto. I restanti sono relativi ad un altro progetto con directory comune.

## Organizzazione del codice



File relativi a definizione e dichiarazione della classe Point, che rappresenta i punti sui quali verrà eseguito Kmeans.

## Organizzazione del codice



```
class Point {
public:
    Point(int id, const std::string& line, bool csv = false, int stop = 1000);

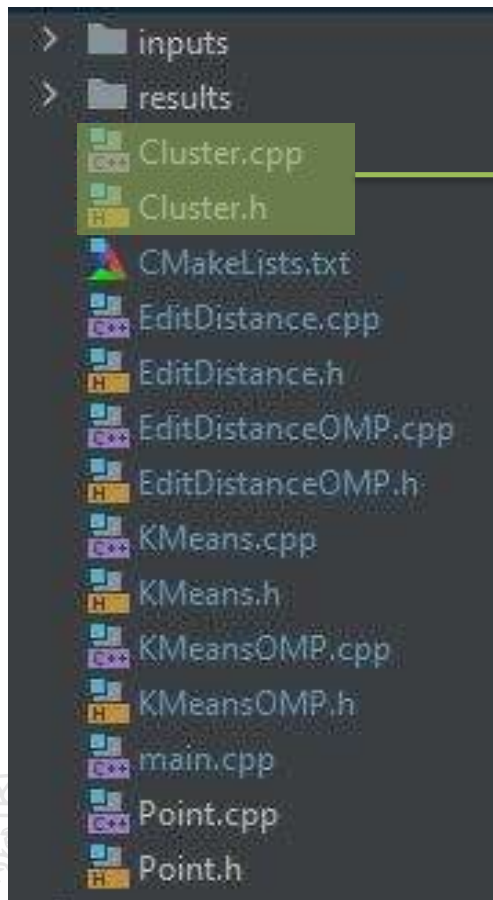
    Point (Point const &p);

    int getId() const;
    void setId(int id);
    int getClusterId() const;
    void setClusterId(int c);
    int getDimensions() const;
    void setDimensions (int d);
    double getVal(int pos) const;

private:
    int pointId, clusterId;
    int dimensions;
    std::vector<double> values;

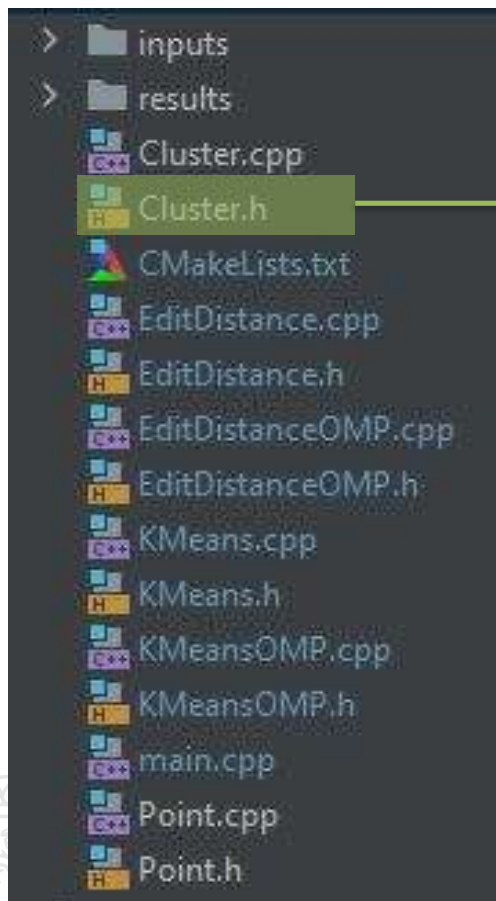
    std::vector<double> linetoVecTXT(const std::string& line);
    std::vector<double> linetoVecCSV(const std::string& line, int stop);
};
```

## Organizzazione del codice



File relativi a definizione e dichiarazione della classe Cluster, che rappresenta i cluster su cui verranno raggruppati i punti.

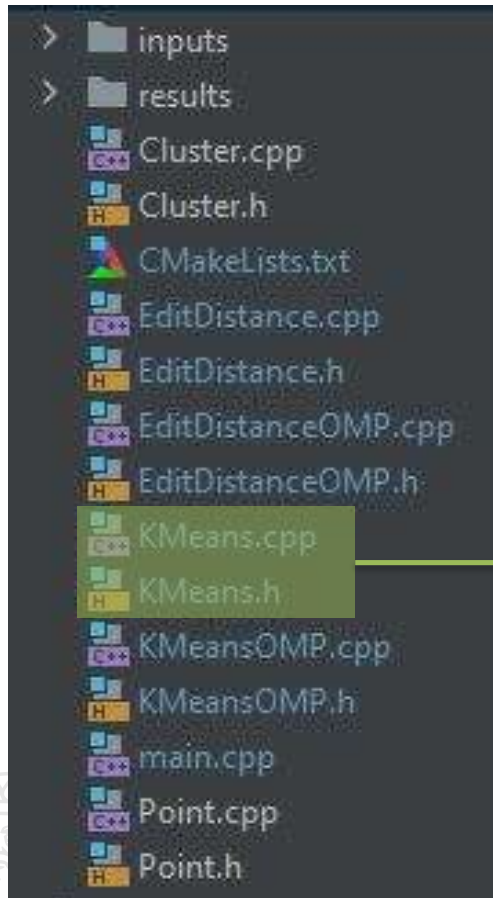
## Organizzazione del codice



```
class Cluster {  
public:  
    Cluster(int clusterId, const Point& centroid);  
  
    int getClusterId() const;  
    void setClusterId(int id);  
    double getCentroidPos(int pos);  
    void setCentroidPos(int pos, double value);  
    void addPoint(Point p);  
    void removeAllPoints();  
    Point getPoint(int pos);  
    int getClusterSize();  
  
private:  
    int clusterId;  
    std::vector<double> centroid;  
    std::vector<Point> points;  
};
```

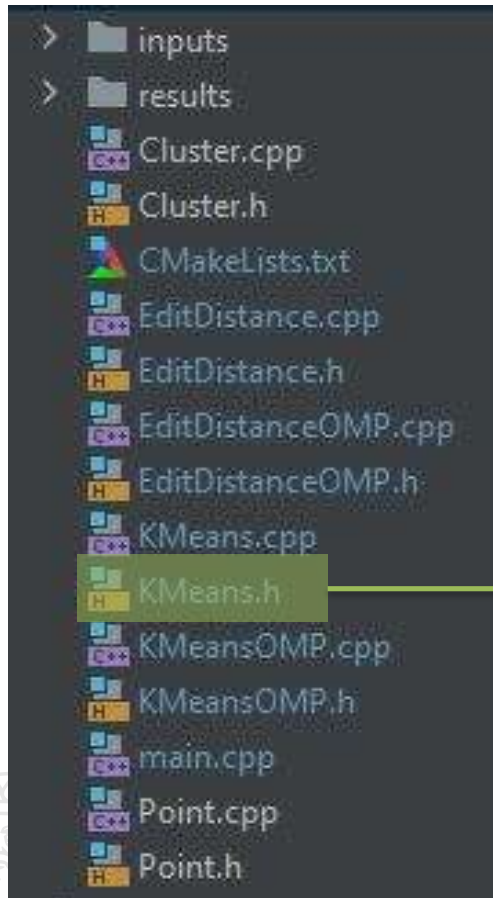


## Organizzazione del codice



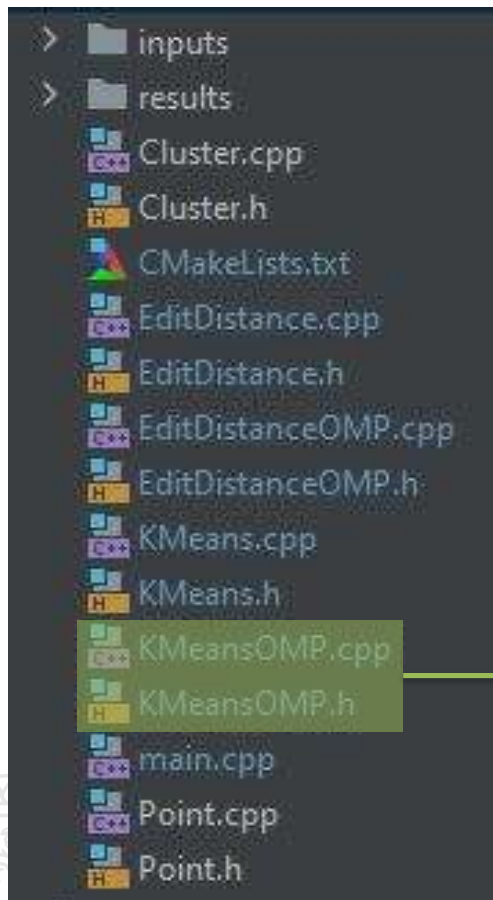
File relativi a definizione e dichiarazione della classe KMeans, che racchiude i parametri e il metodo per eseguire l'algoritmo.

## Organizzazione del codice



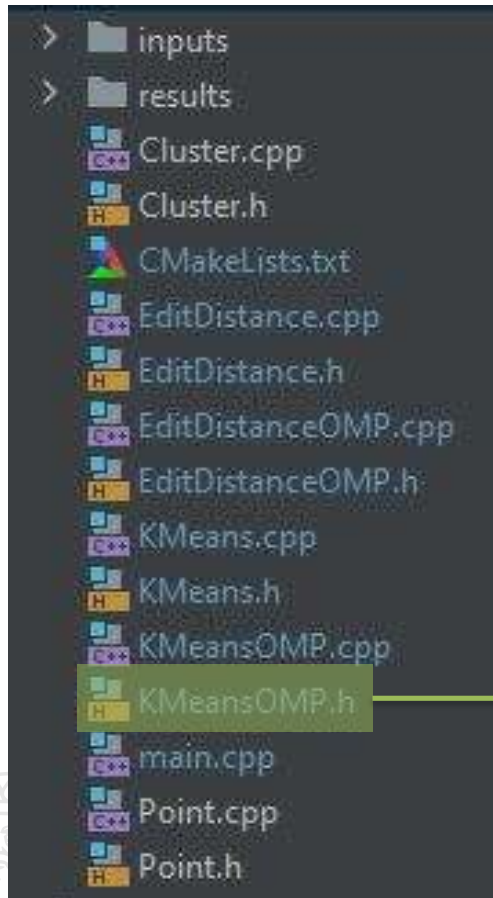
```
class KMeans {  
public:  
    KMeans(int K, int epochs);  
  
    void run(std::vector<Point> allPoints, int seed);  
  
private:  
    int K, epochs, dimensions, nPoints;  
    std::vector<Cluster> clusters;  
  
    void clearClusters();  
    int getNearestClusterId(const Point& p);  
};
```

## Organizzazione del codice



File relativi a definizione e dichiarazione della classe KMeansOMP, che racchiude i parametri e il metodo parallelizzato per eseguire l'algoritmo.

## Organizzazione del codice



```
class KMeansOMP {  
public:  
    KMeansOMP(int K, int epochs);  
  
    void run(std::vector<Point> allPoints, int seed, int threads);  
  
private:  
    int K, epochs, dimensions, nPoints;  
    std::vector<Cluster> clusters;  
  
    void clearClusters();  
    int getNearestClusterId(const Point& p);  
};
```

# Algoritmo sequenziale

```
void KMeans::run(std::vector<Point> algPoints, int seed) {
    nPoints = (int)algPoints.size();
    dimensions = algPoints[0].getDimensions();
    //initializing clusters
    std::vector<int> usedPointsIds;
    std::random_device rd;
    std::default_random_engine eng( s rd());
    eng.seed( s seed);
    std::uniform_int_distribution<int> distr( a: 0, b: nPoints);
    for (int i = 1; i <= K; i++) {
        int index = distr( &eng);
        while(std::find( first: usedPointsIds.begin(), last: usedPointsIds.end(), val: index) != usedPointsIds.end()) {
            index = distr( &eng);
        }
        usedPointsIds.push_back(index);
        algPoints[index].setClusterId( c: i);
        Cluster cluster( clusterId: i, centroid: algPoints[index]);
        clusters.push_back(cluster);
    }
    std::cout << "Clusters initialized = " << clusters.size() << std::endl << std::endl;
    std::cout << "Running K-Means clustering.." << std::endl;
    int epoch = 1;
    bool run = true;
    while(run) {
        std::cout << "Epoch " << epoch << " / " << epochs << std::endl;
        int changed = 0;
```

# Algoritmo sequenziale

```
//add all points to their nearest cluster
for (int i = 0; i < nPoints; i++) {
    int currentClusterId = algPoints[i].getClusterId();
    int nearestClusterId = getNearestClusterId(p, algPoints[i]);
    //std::cout << "Current: " << currentClusterId << ", nearest: " << nearestClusterId << std::endl;
    if (currentClusterId != nearestClusterId) {
        algPoints[i].setClusterId(nearestClusterId);
        changed++;
    }
}

//clear all existing clusters
clearClusters();

//reassign points to their new clusters
for (int i = 0; i < nPoints; i++) {
    //cluster index is ID-1
    clusters[algPoints[i].getClusterId() - 1].addPoint(p, algPoints[i]);
}

//recalculating the center of each cluster
for (int i = 0; i < K; i++) {
    int clusterSize = clusters[i].getClusterSize();
    for (int j = 0; j < dimensions; j++) {
        double sum = 0.0;
        if (clusterSize > 0) {
            for (int p = 0; p < clusterSize; p++) {
                sum += clusters[i].getPoint(pos, p).getVal(pos, j);
            }
            clusters[i].setCentroidPos(pos, j, value: sum / clusterSize);
        }
    }
}

if ((float)changed / (float)nPoints <= 0.001 || epoch >= epochs) {
    std::cout << "Clustering completed in epoch : " << epoch << std::endl << std::endl;
    run = false;
}

epoch++;
```

Terminazione  
dell'algoritmo.



## Funzione getNearestClusterId()

```
int KMeans::getNearestClusterId(const Point& p) {  
    double sum, min_dist = DBL_MAX;  
    int nearestClusterId;  
    for (int i = 0; i < K; i++) {  
        double dist;  
        sum = 0.0;  
        if (dimensions == 1) {  
            dist = fabs(x: clusters[i].getCentroidPos( pos: 0) - p.getVal( pos: 0));  
        }  
        else {  
            for (int j = 0; j < dimensions; j++) {  
                sum += pow(x: clusters[i].getCentroidPos( pos: j) - p.getVal( pos: j), y: 2.0);  
            }  
            dist = sqrt(x: sum);  
        }  
        if (dist < min_dist) {  
            min_dist = dist;  
            nearestClusterId = clusters[i].getClusterId();  
        }  
    }  
    return nearestClusterId;  
}
```

## Algoritmo parallelo

Parallelizzazione dell'assegnazione dei punti ai cluster più vicini

```
//add all points to their nearest cluster
#pragma omp parallel for default(none) shared(algPoints, changed) num_threads(threads)
for (int i = 0; i < nPoints; i++) {
    int currentClusterId = algPoints[i].getClusterId();
    int nearestClusterId = getNearestClusterId(p: algPoints[i]);
    if (currentClusterId != nearestClusterId) {
        #pragma omp atomic
        changed++;
        algPoints[i].setClusterId(c: nearestClusterId);
    }
}
```



## Algoritmo parallelo

Parallelizzazione dell'assegnazione dei punti ai cluster più vicini

```
//add all points to their nearest cluster
#pragma omp parallel for default(none) shared(algPoints, changed) num_threads(threads)
for (int i = 0; i < nPoints; i++) {
    int currentClusterId = algPoints[i].getClusterId();
    int nearestClusterId = getNearestClusterId(p: algPoints[i]);
    if (currentClusterId != nearestClusterId) {
        #pragma omp atomic
        changed++;
        algPoints[i].setClusterId(c: nearestClusterId);
    }
}
```

Gestione dell'incremento della  
variabile condivisa changed.

## Algoritmo parallelo

Parallelizzazione del calcolo delle coordinate dei nuovi centroidi

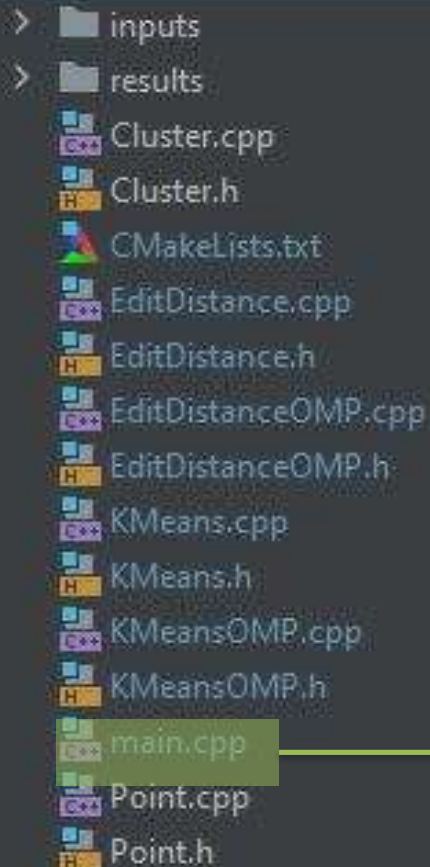
```
//recalculating the center of each cluster
for (int i = 0; i < K; i++) {
    int clusterSize = clusters[i].getClusterSize();
    for (int j = 0; j < dimensions; j++) {
        double sum = 0.0;
        #pragma omp parallel for default(none) firstprivate(i, j) shared(clusterSize) \
        reduction(+: sum) num_threads(threads)
        for (int p = 0; p < clusterSize; p++) {
            sum += clusters[i].getPoint( pos: p).getVal( pos: j);
        }
        if (clusterSize > 0) {
            clusters[i].setCentroidPos( pos: j, value: sum / clusterSize);
        }
    }
}
```

## Algoritmo parallelo

Parallelizzazione del calcolo delle coordinate dei nuovi centroidi

```
//recalculating the center of each cluster
for (int i = 0; i < K; i++) {
    int clusterSize = clusters[i].getClusterSize();
    for (int j = 0; j < dimensions; j++) {
        double sum = 0.0;
        #pragma omp parallel for default(none) firstprivate(i, j) shared(clusterSize) \
        reduction(+: sum) num_threads(threads)
        for (int p = 0; p < clusterSize; p++) {
            sum += clusters[i].getPoint( pos: p).getVal( pos: j);
        }
        if (clusterSize > 0) {
            clusters[i].setCentroidPos( pos: j, value: sum / clusterSize);
        }
    }
}
```

## Test



- > inputs
- > results
- Cluster.cpp
- Cluster.h
- CMakeLists.txt
- EditDistance.cpp
- EditDistance.h
- EditDistanceOMP.cpp
- EditDistanceOMP.h
- KMeans.cpp
- KMeans.h
- KMeansOMP.cpp
- KMeansOMP.h
- main.cpp
- Point.cpp
- Point.h

```
//test KMeans  
std::vector<Point> readPointsCSV(const std::string& fileName, int stop = 1000) {...}  
  
double testKMeansTime(int K, int epochs, const std::vector<Point>& points) {...}  
  
double testKMeansOMPTIME(int K, int epochs, const std::vector<Point>& points, int nThreads) {...}  
  
void testKMeans() {...}
```

I test si trovano  
nel file main.cpp

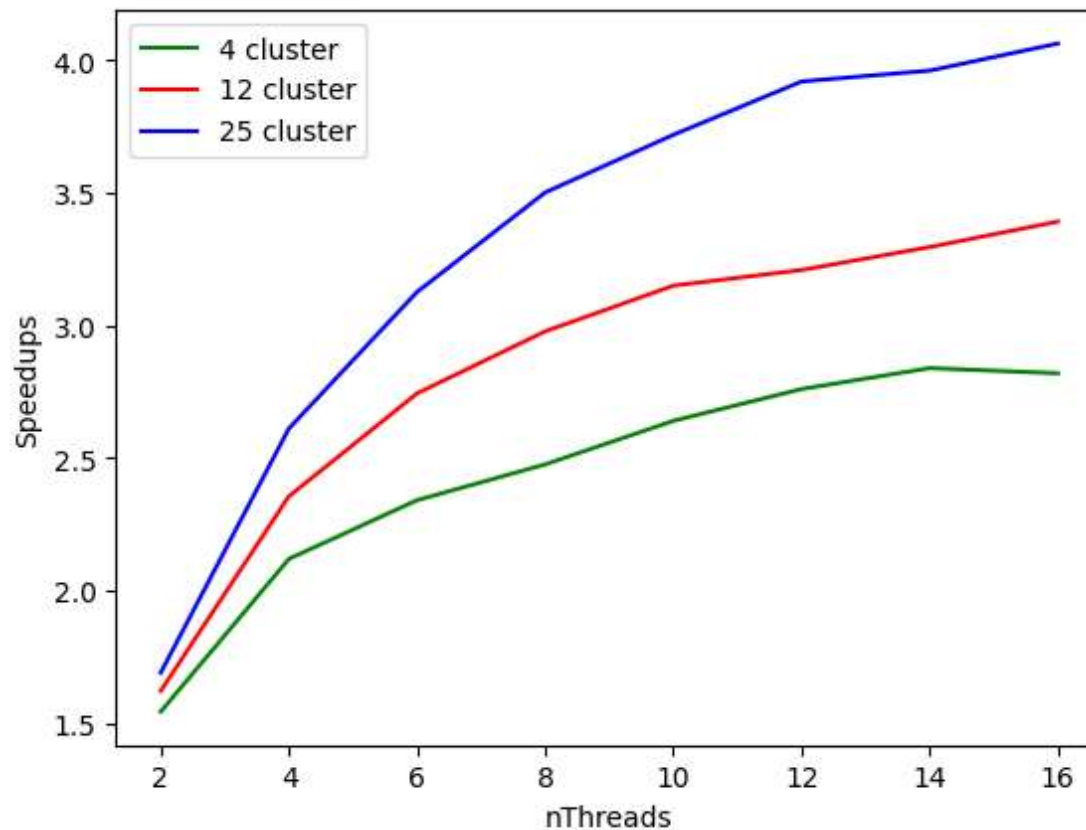
## Test 1

Numero minimo di punti per preferire esecuzione parallela

	3000 points 2D	19000 points 8D
4 threads, $K = 3$	0.946291	2.23869
8 threads, $K = 6$	0.948598	3.27011
4 threads, $K = 5$	0.989534	2.64693
8 threads, $K = 8$	0.964235	3.29542

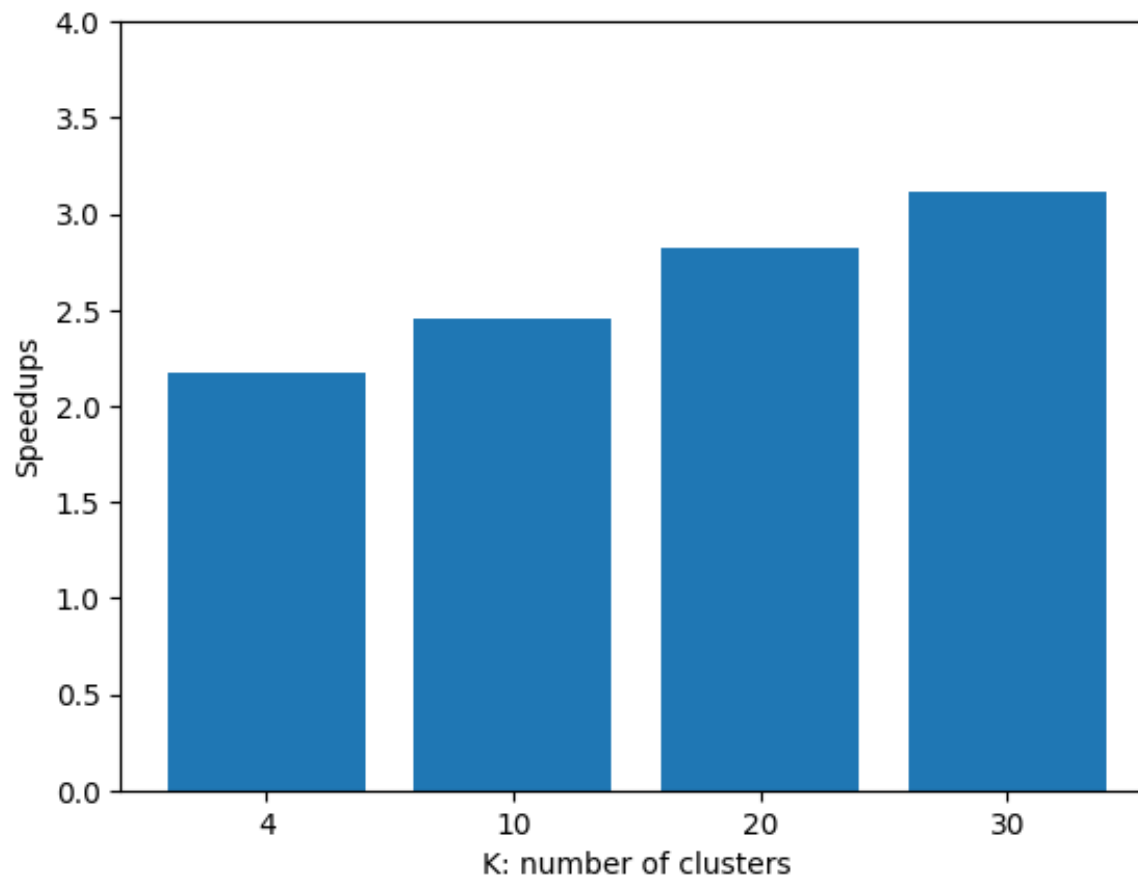
## Test 2

Variare numero di threads e valore di K, con 500000 punti a 5D



## Test 3

Variare valore di K, su 4mln di punti a 3D e numero di threads pari a 8



## Test 4

Considerazioni sul numero di punti su cui eseguire KMeans

	Speedup con 8 threads
3000 punti 2D, $K = 3$	0.948598
19000 punti 8D, $K = 5$	3.27011
500000 punti 5D, $K = 4$	2.47619
4mln di punti 3D, $K = 4$	2.17211

