UNIVERSITÀ DEGLI STUDI FIRENZE

Parallel KMeans

Parallel Programming for Machine Learning

Niccolò Arati

28/03/2024

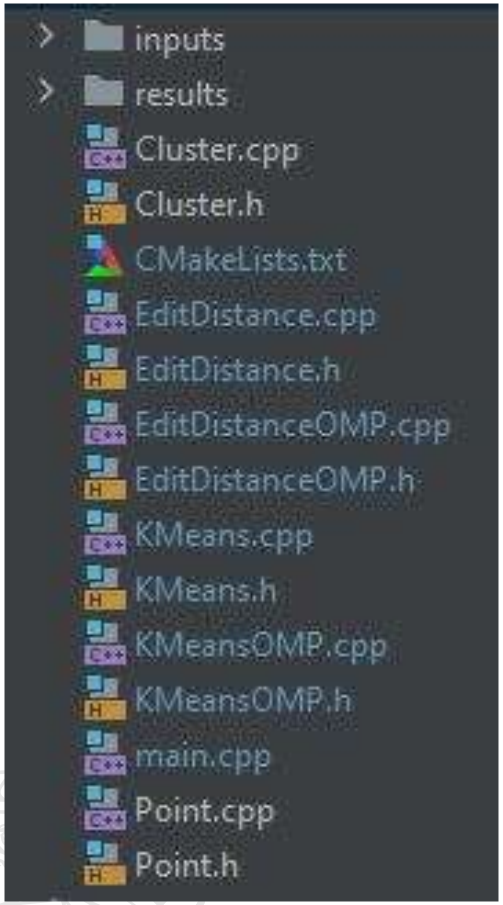
Introduction

The program written for this project implements **K-means**, an unsupervised learning clustering algorithm. Given a dataset of points and a number **K** of clusters, K-means assigns each point to one of the K clusters, minimizing the mean squared error.
The **convergence** of the algorithm is set by a maximum number of iterations, 35, and by the percentage of points that change clusters after each iteration. If this percentage falls below 0.1%, the algorithm terminates.

There are two implementations of K-means: one **sequential** and one **parallel**. The purpose of this work is to observe the speedup obtained with the parallel version compared to the sequential one.

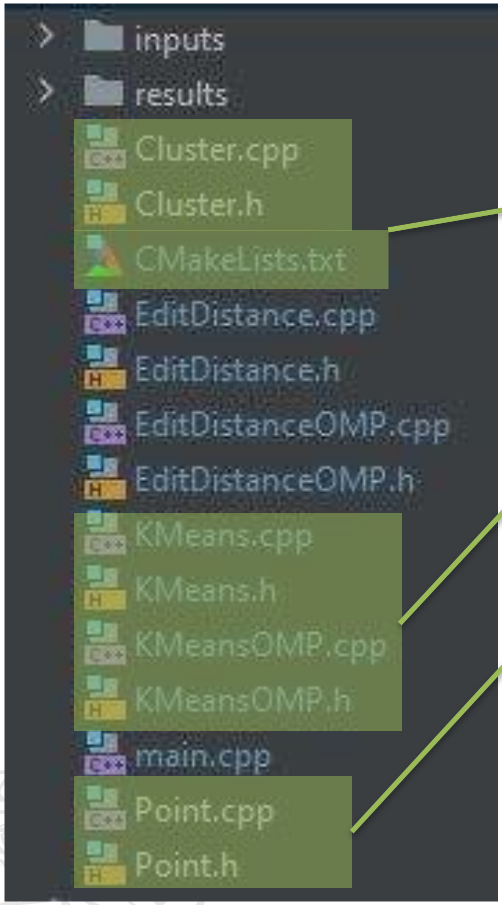
The programming language used is C++ (MinGW compiler, CLion IDE), and parallelization was done with **OpenMP**.
All experiments were conducted on the ssh server "papavero.dinfo.unifi.it".

Code organization



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

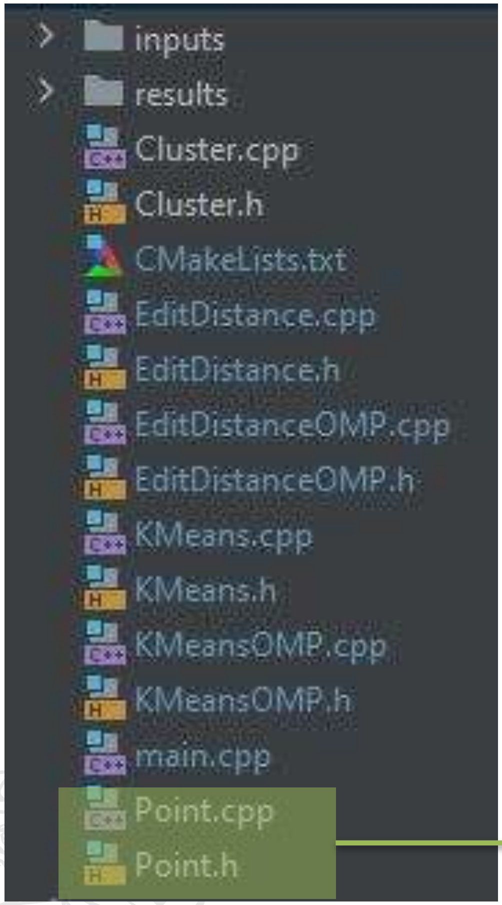
Code organization



```
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
```

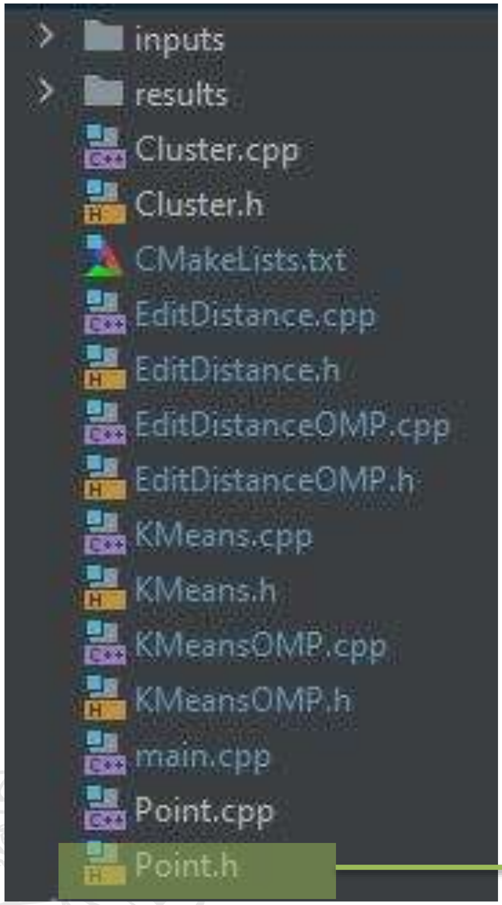
Files included in this project. The remaining ones are related to another project with a common directory.

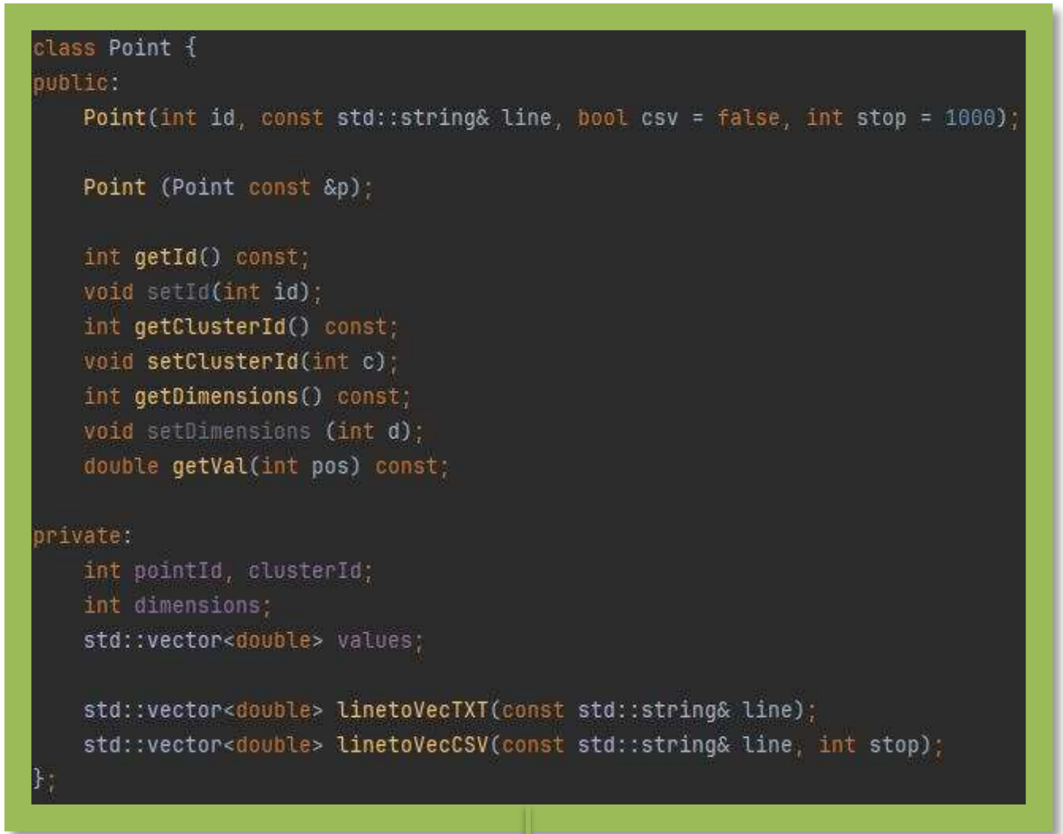
Code organization


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

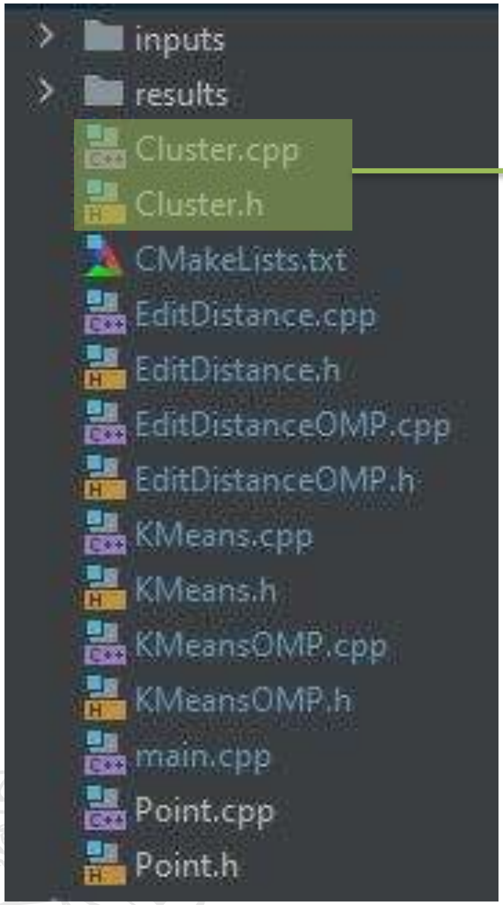
Files related to the definition and declaration of the Point class, which represents the points on which K-means will be executed.

Code organization


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


```
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);
};
```

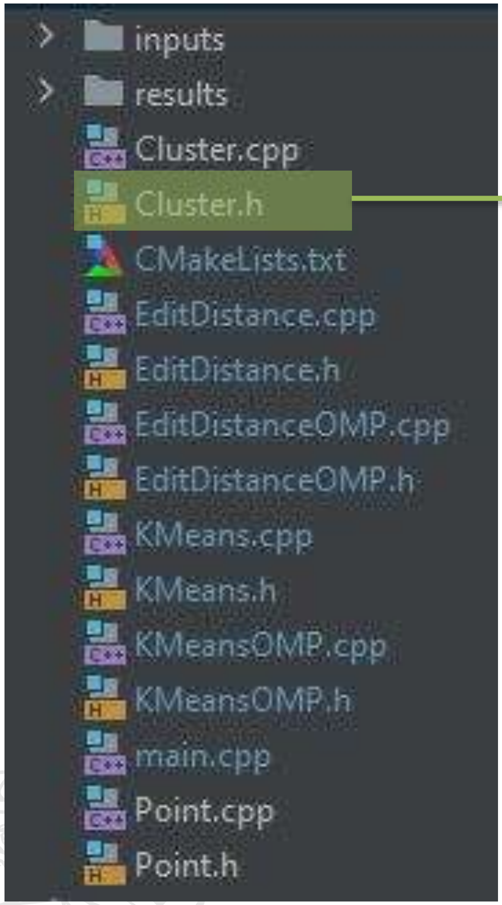
Code organization



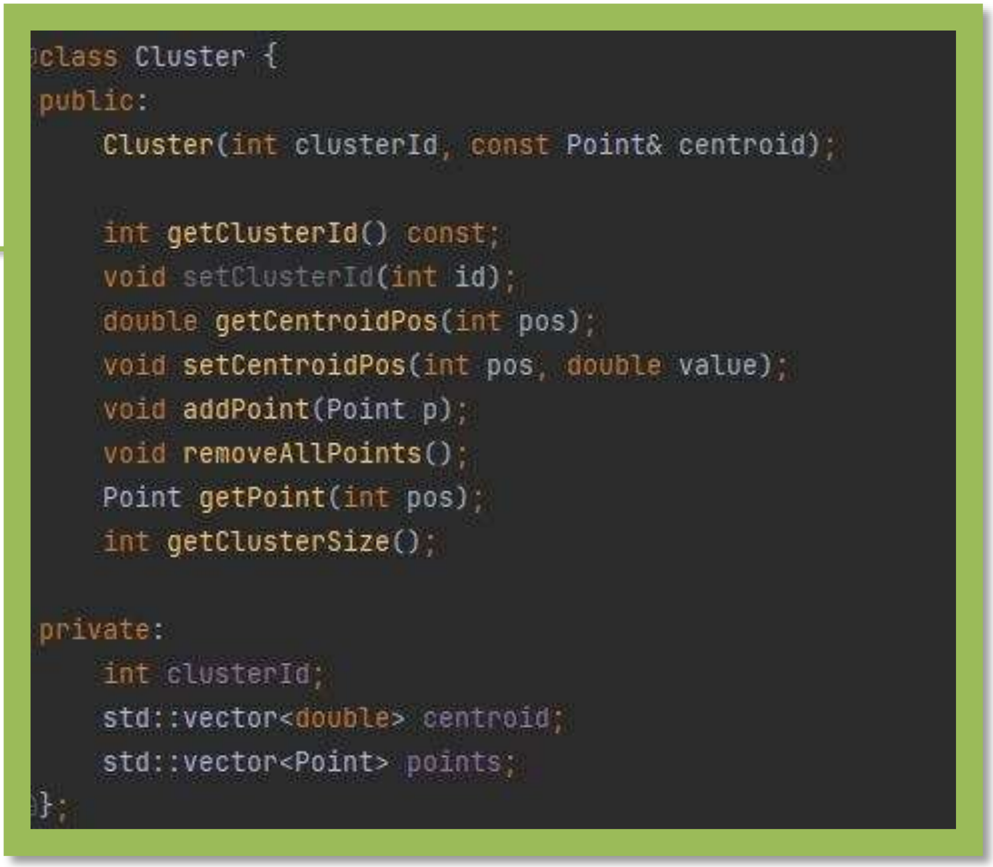
```
inputs
results
Cluster.cpp
Cluster.h
CMakeLists.txt
EditDistance.cpp
EditDistance.h
EditDistanceOMP.cpp
EditDistanceOMP.h
KMeans.cpp
KMeans.h
KMeansOMP.cpp
KMeansOMP.h
main.cpp
Point.cpp
Point.h
```

Files related to the definition and declaration of the Cluster class, which represents the clusters that the points will be grouped into.

Code organization

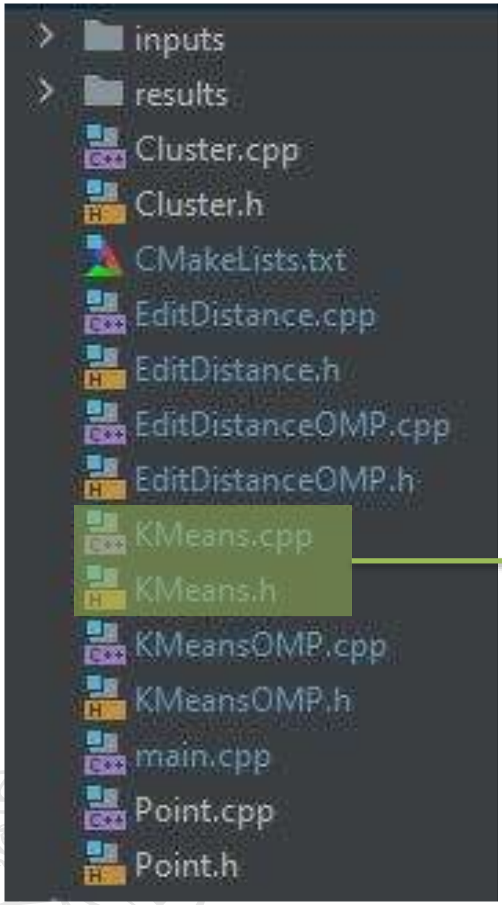


- inputs
- results
- Cluster.cpp
- Cluster.h
- CMakeLists.txt
- EditDistance.cpp
- EditDistance.h
- EditDistanceOMP.cpp
- EditDistanceOMP.h
- KMeans.cpp
- KMeans.h
- KMeansOMP.cpp
- KMeansOMP.h
- main.cpp
- Point.cpp
- Point.h



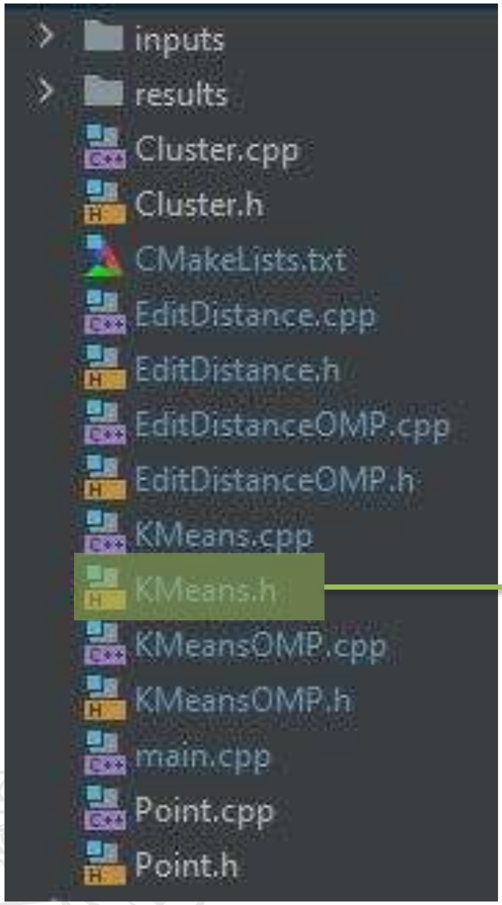
```
class Cluster {
public:
    Cluster(int clusterId, const Point& centroid);

    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;
};
```

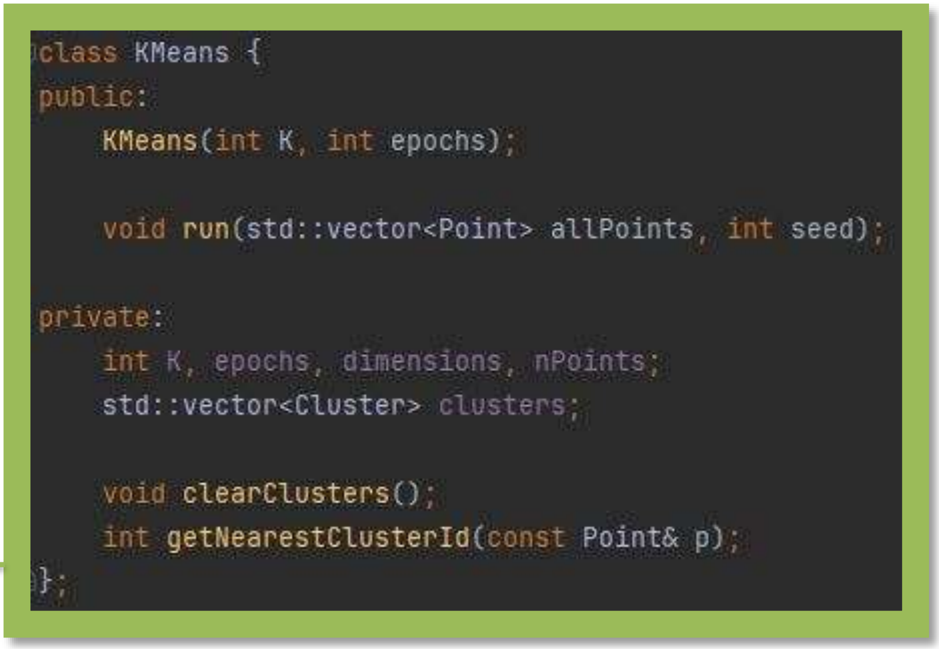
Code organization


```
> 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
```

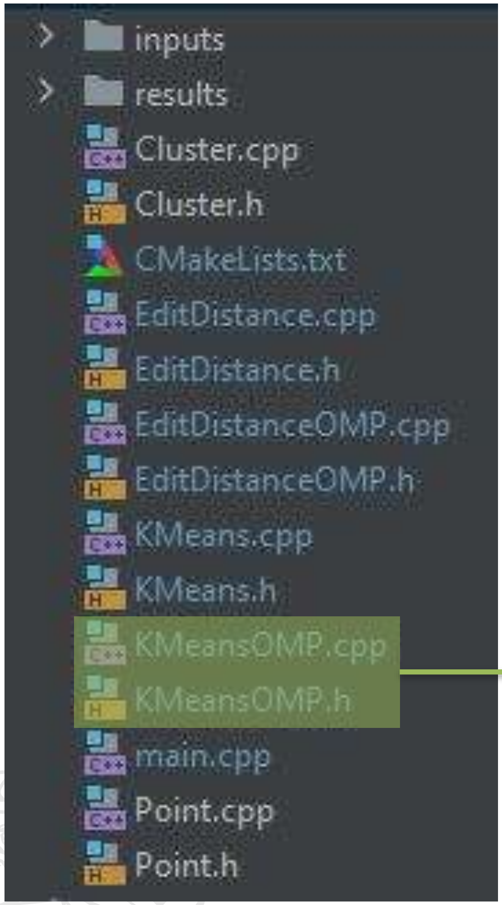
Files related to the definition and declaration of the KMeans class, which encapsulates the parameters and the method to execute the algorithm.

Code organization


- 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

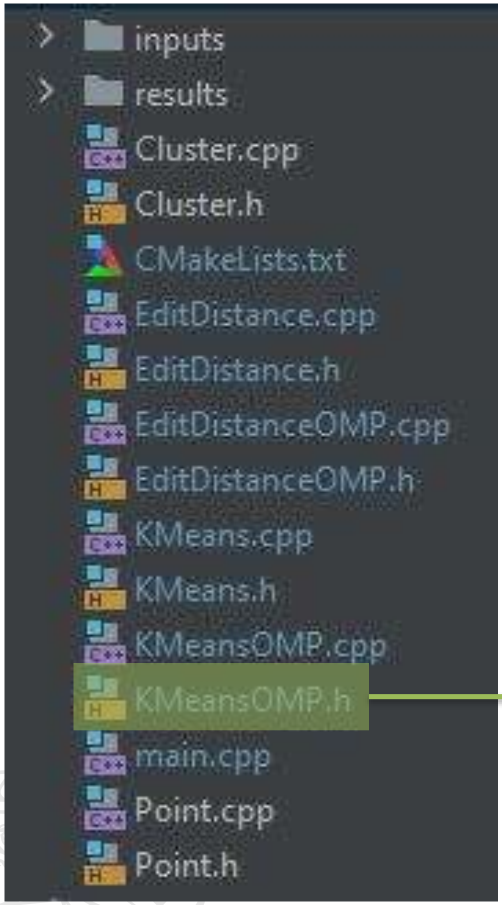

```
class KMeans {
public:
    KMeans(int K, int epochs);

    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);
};
```

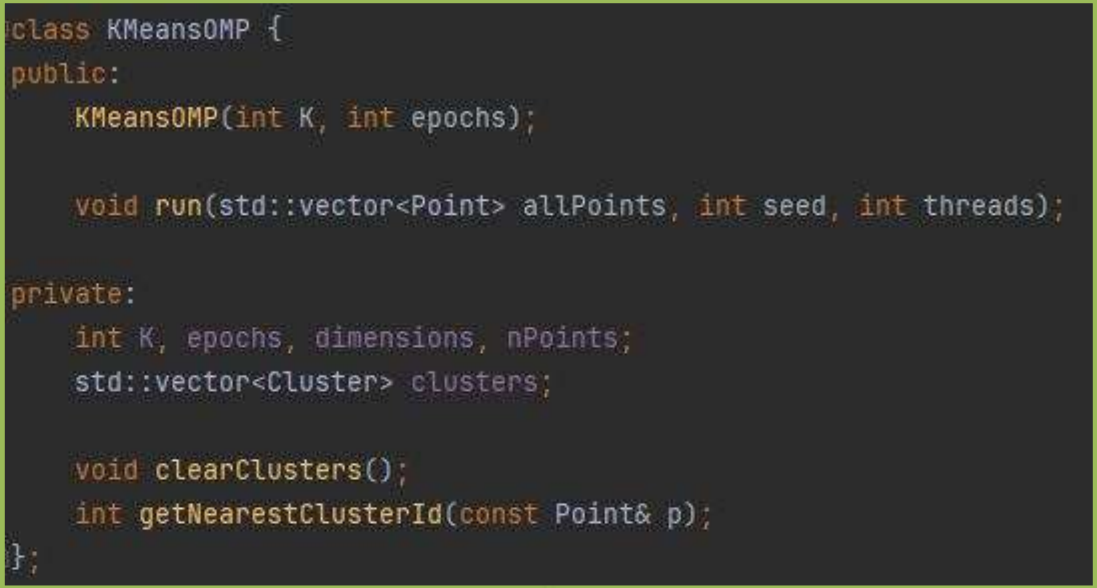
Code organization


```
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
```

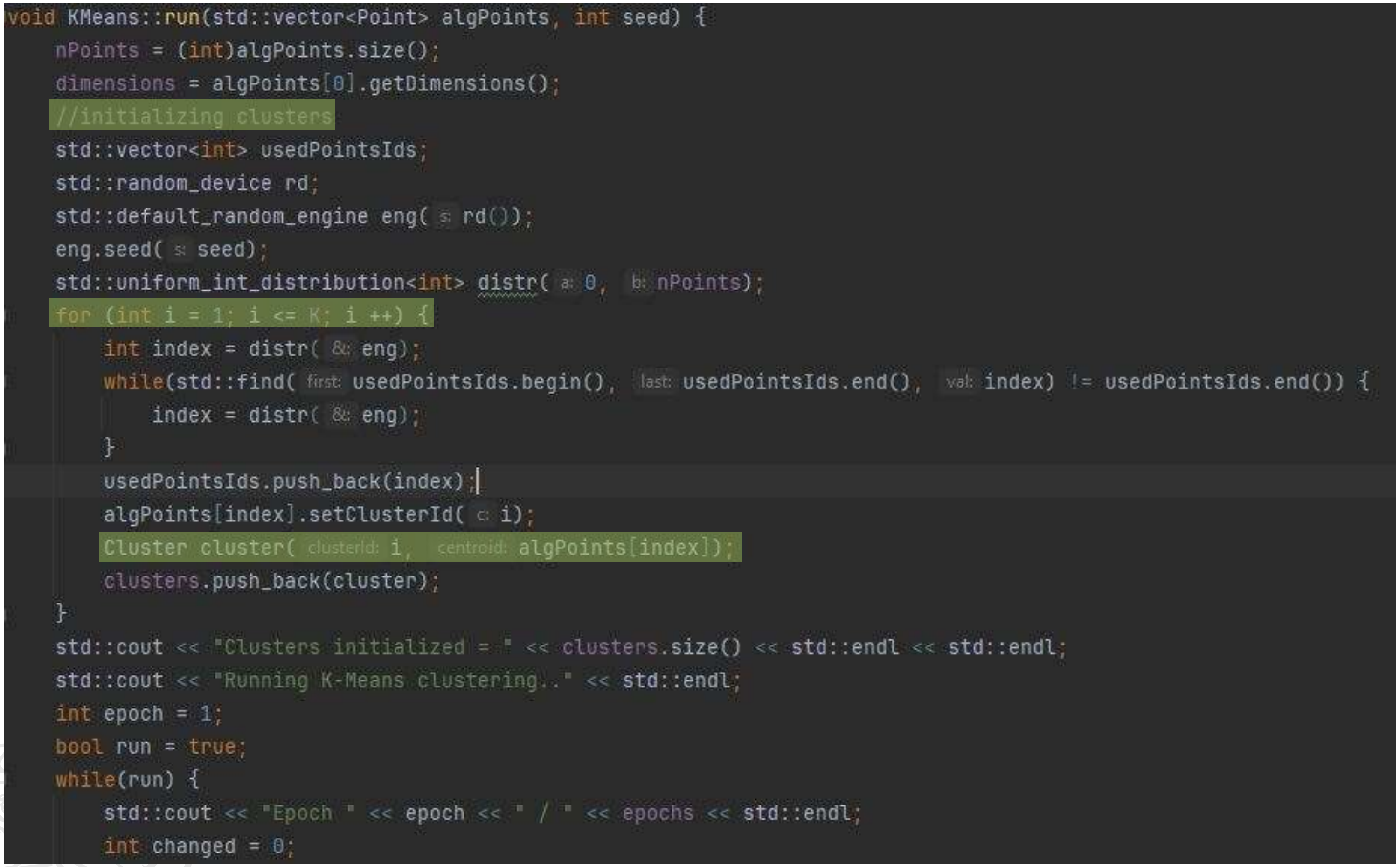
Files related to the definition and declaration of the KMeansOMP class, which encapsulates the parameters and the parallelized method to execute the algorithm.

Code organization


```
> 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
```

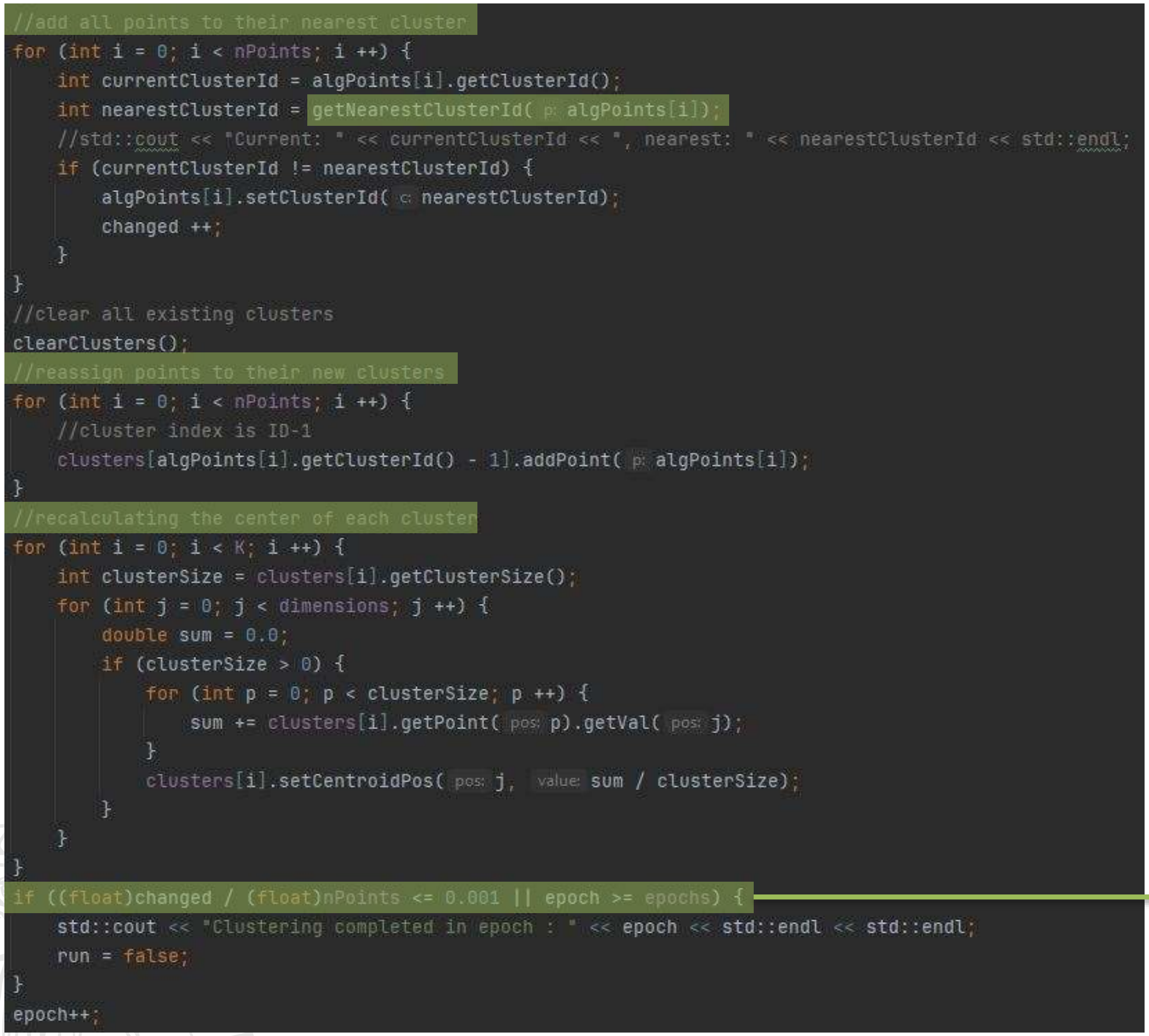

```
class KMeansOMP {
public:
    KMeansOMP(int K, int epochs);

    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);
};
```

Sequential Algorithm



```
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;
```

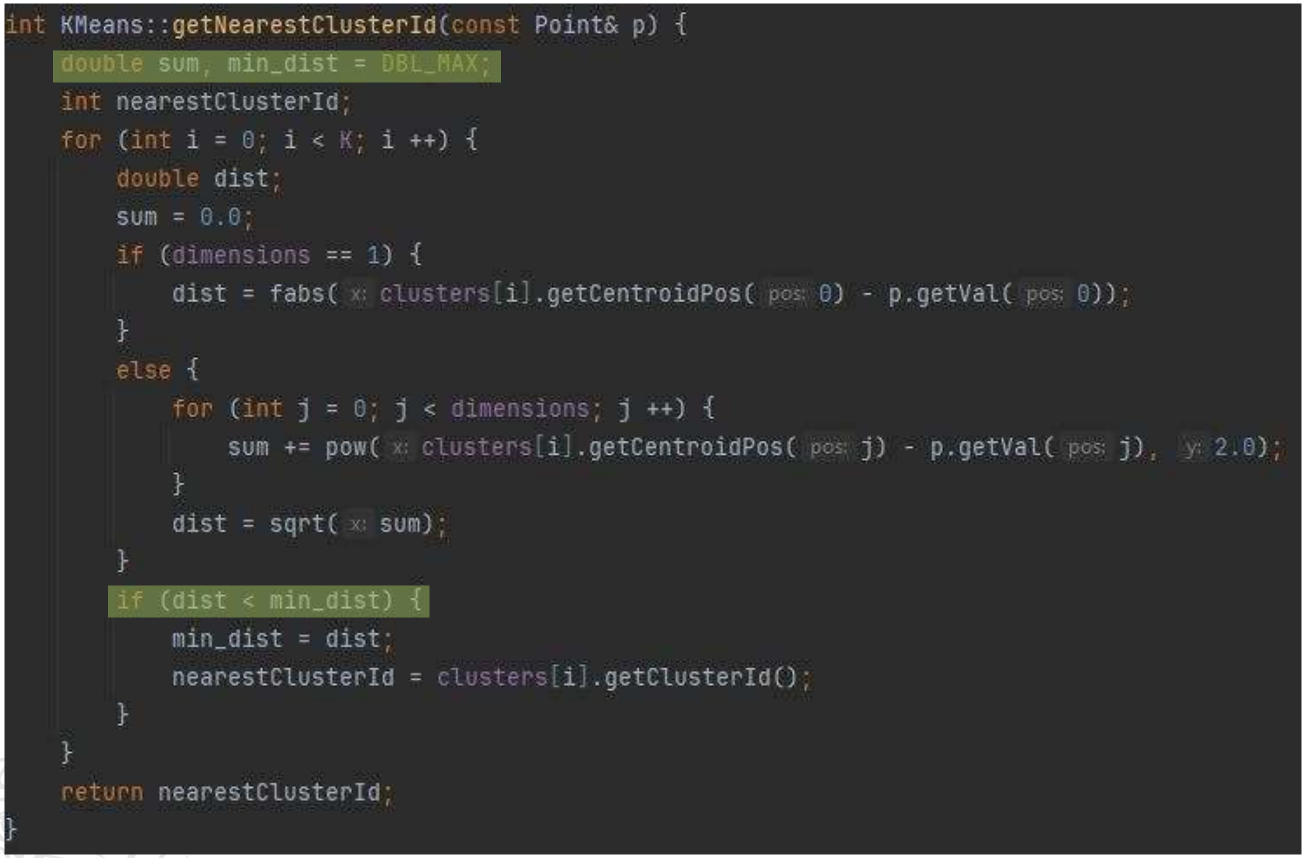
Sequential Algorithm



```
//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( c: 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++;
```

Termination of the algorithm

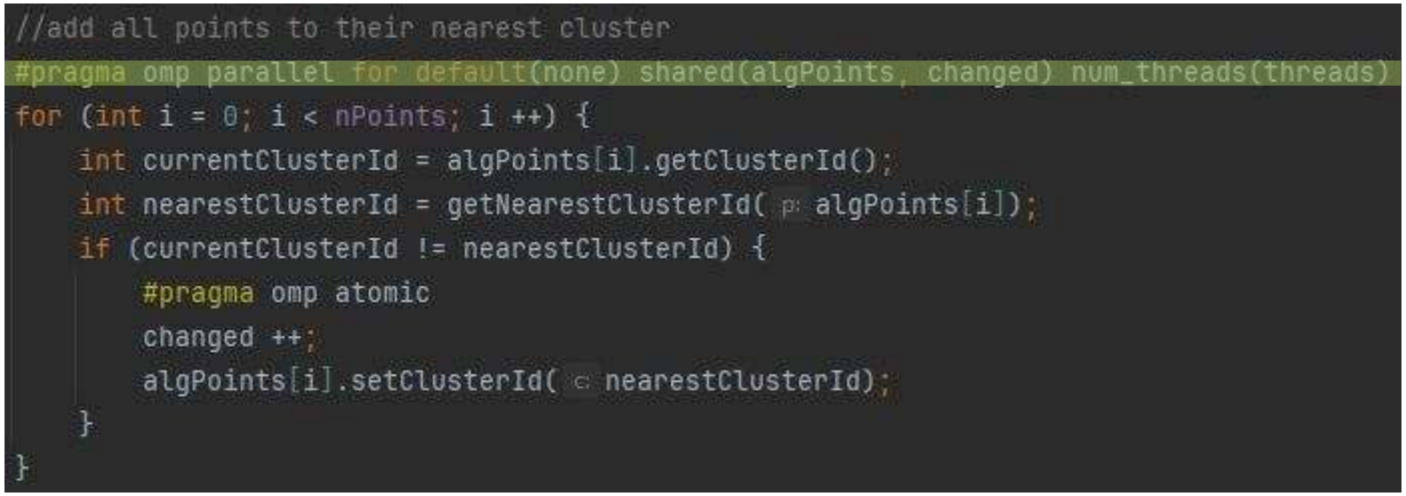
Function getNearestClusterId()



```cpp
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;
}
```

Parallel Algorithm

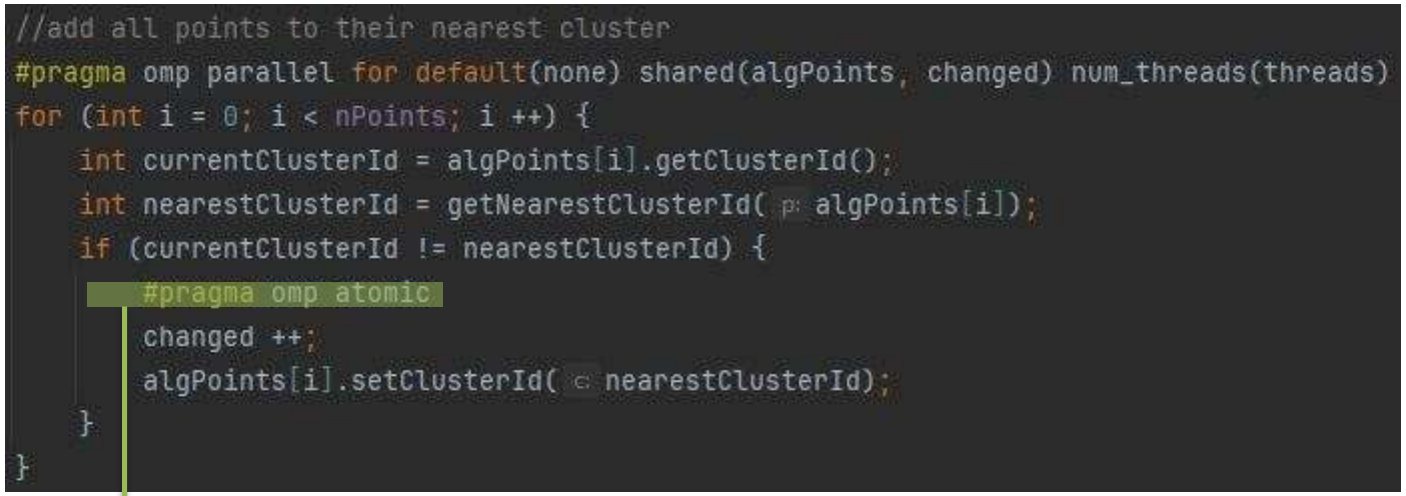
Parallelization of assigning points to the nearest clusters



```
//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);
    }
}
```

Parallel Algorithm

Parallelization of assigning points to the nearest clusters

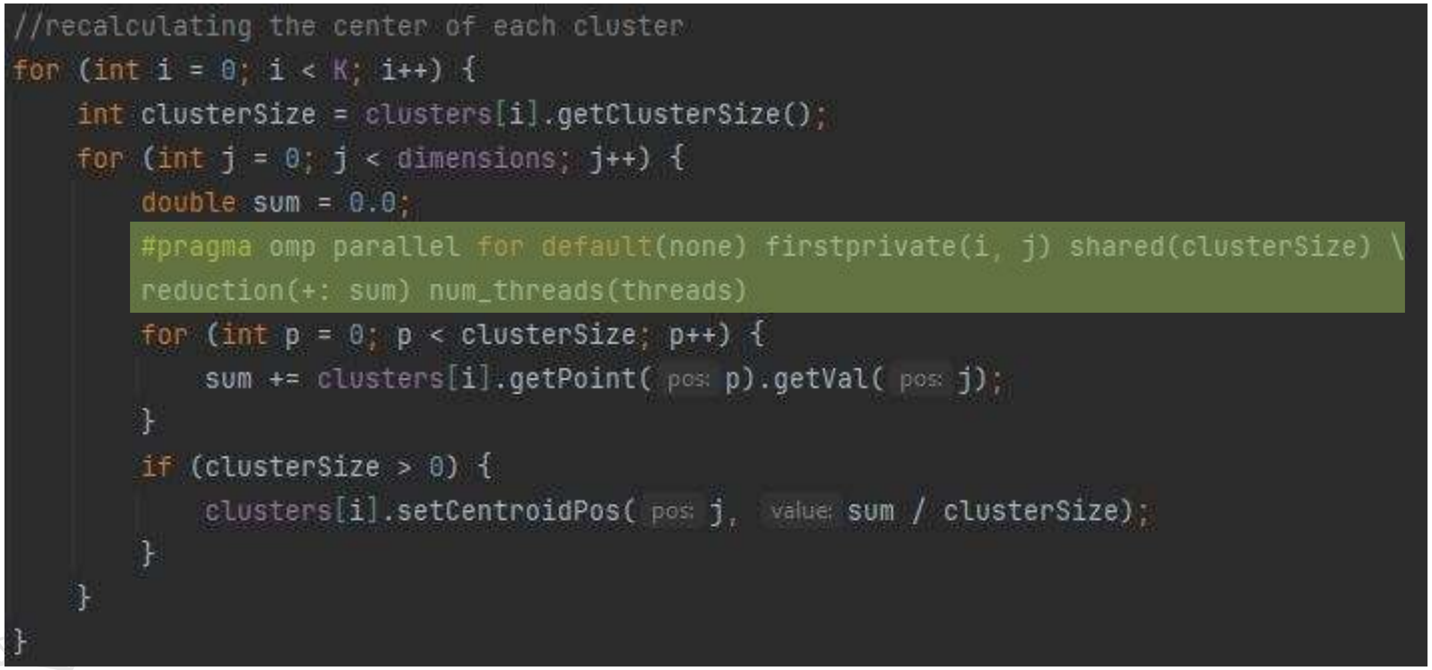


```
//add all points to their nearest cluster
#pragma omp parallel for default(none) shared(algPoints, changed) num_threads(threads)
for (int i = 0; i < nPoints; i ++) {
    int currentClusterId = algPoints[i].getClusterId();
    int nearestClusterId = getNearestClusterId( p: algPoints[i]);
    if (currentClusterId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

Increment management of the shared variable changed.

Parallel Algorithm

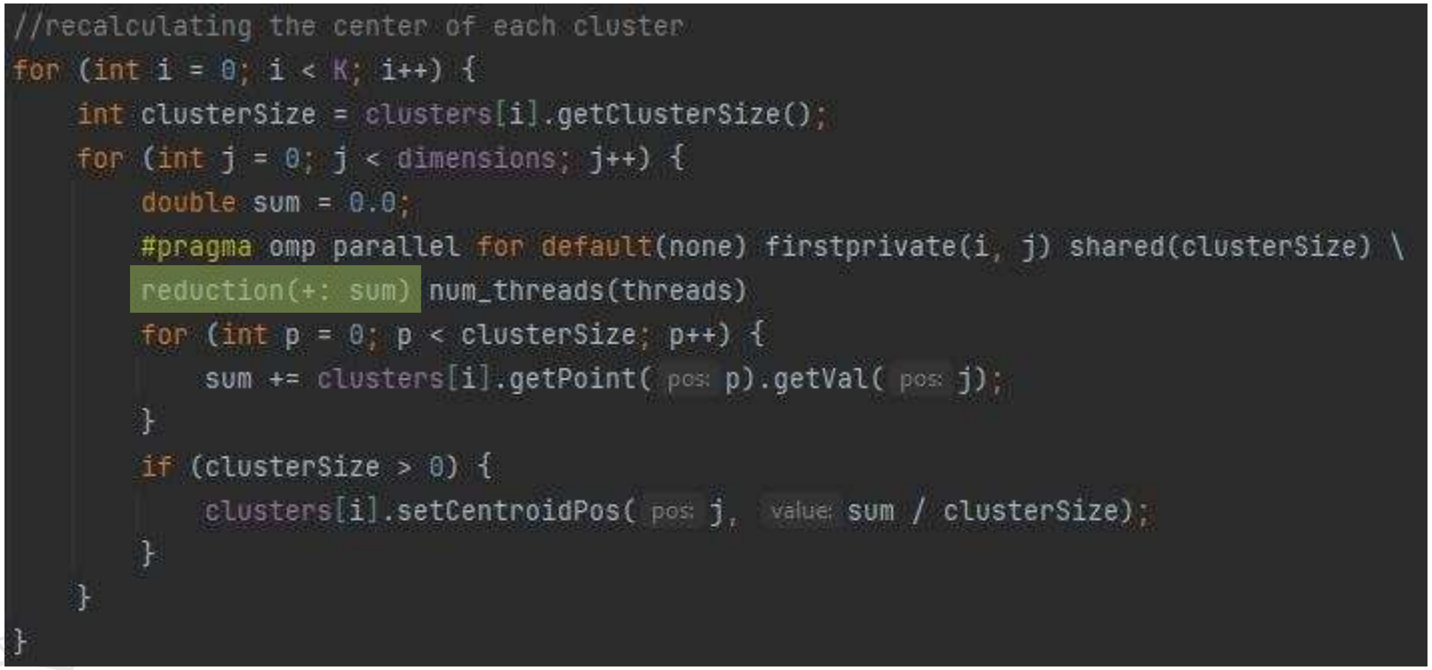
Parallelized computation of the coordinates of the new centroids



```
//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);
        }
    }
}
```

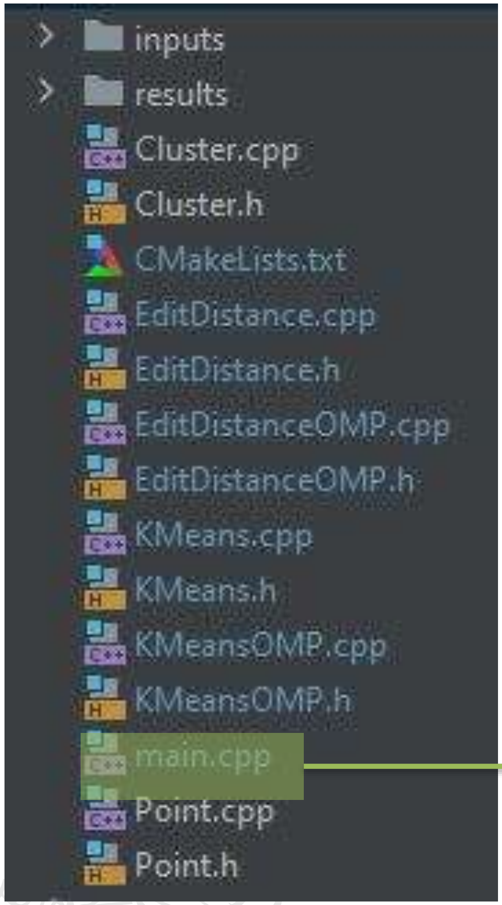
Parallel Algorithm

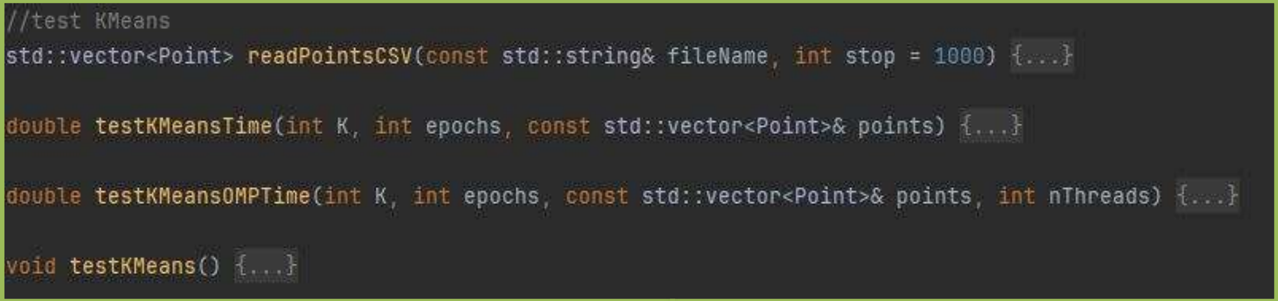
Parallelized computation of the coordinates of the new centroids



```
//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





```
//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() {...}
```

Tests are located in the file main.cpp.

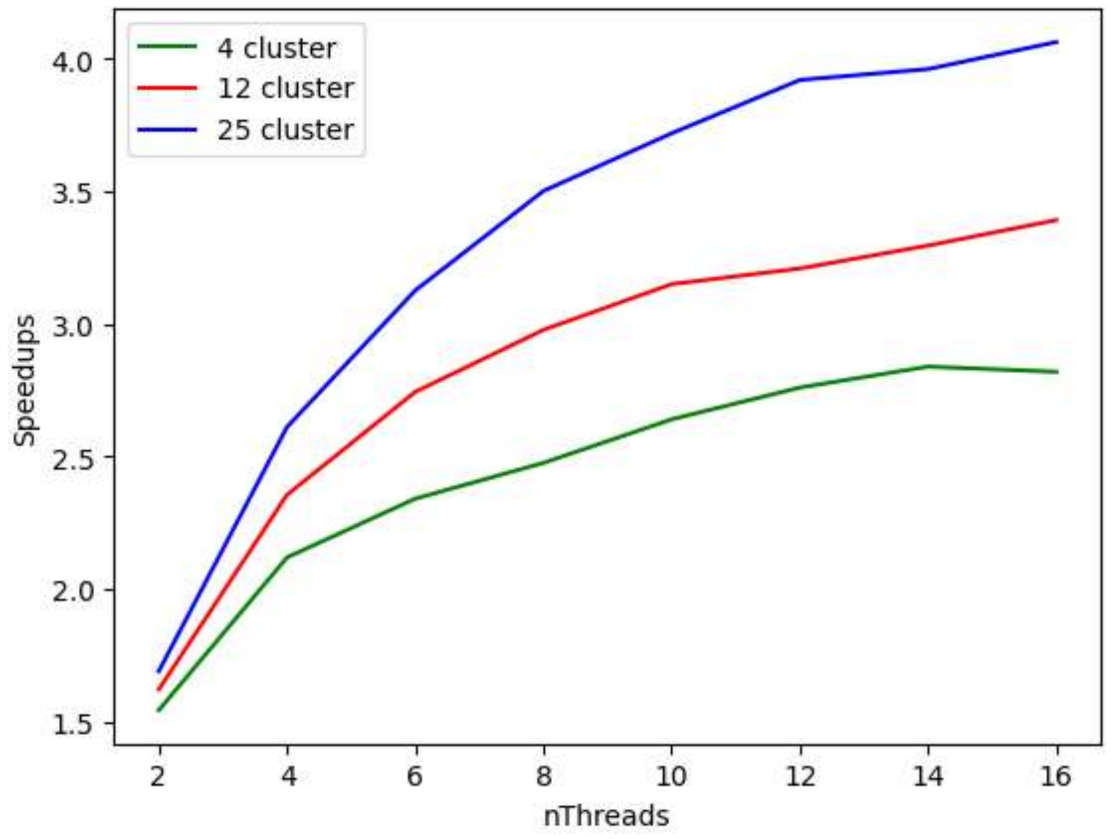
Test 1

Minimum number of points to prefer parallel execution.

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

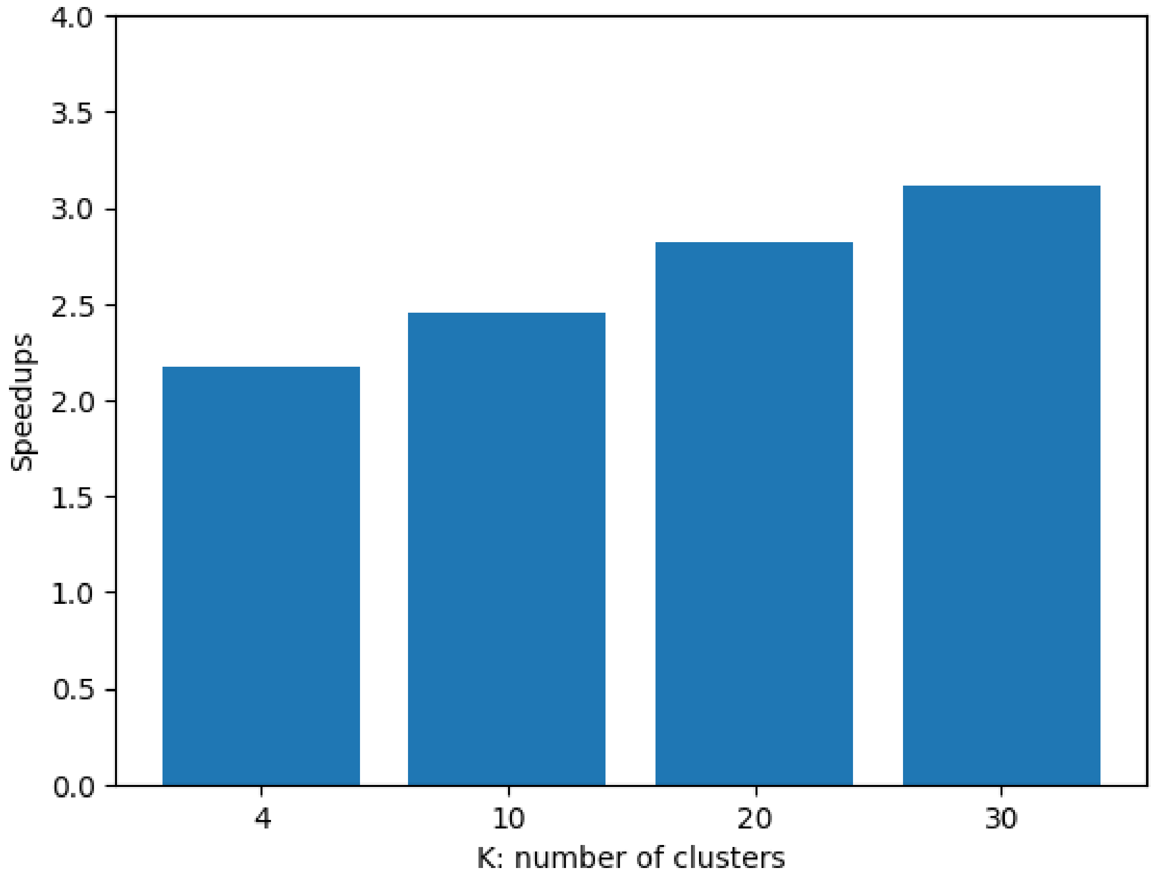
Test 2

Varying the number of threads and the value of K, with 500,000 points in 5D



Test 3

Varying the value of K, on 4 million points in 3D and with 8 threads



Test 4

Considerations about the number of points to run KMeans on

	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