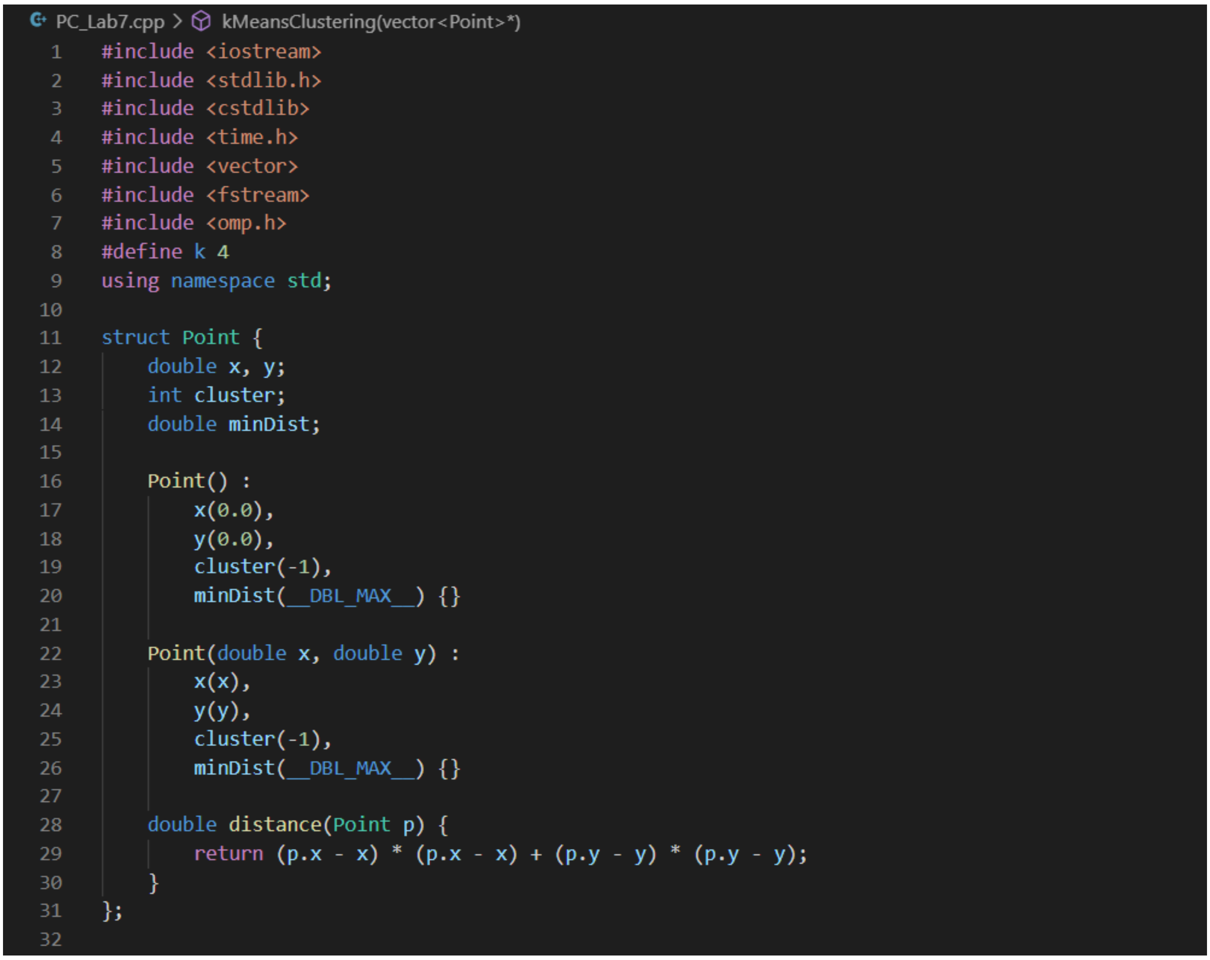
DEPARTMENT OF INFORMATION TECHNOLOGY, NITK SURATHKAL
IT 301 PARALLEL COMPUTING
PC LAB 7
Date: 30 September 2020
Submission Date : 7th October 2020

Name: Chinmayi C. Ramakrishna

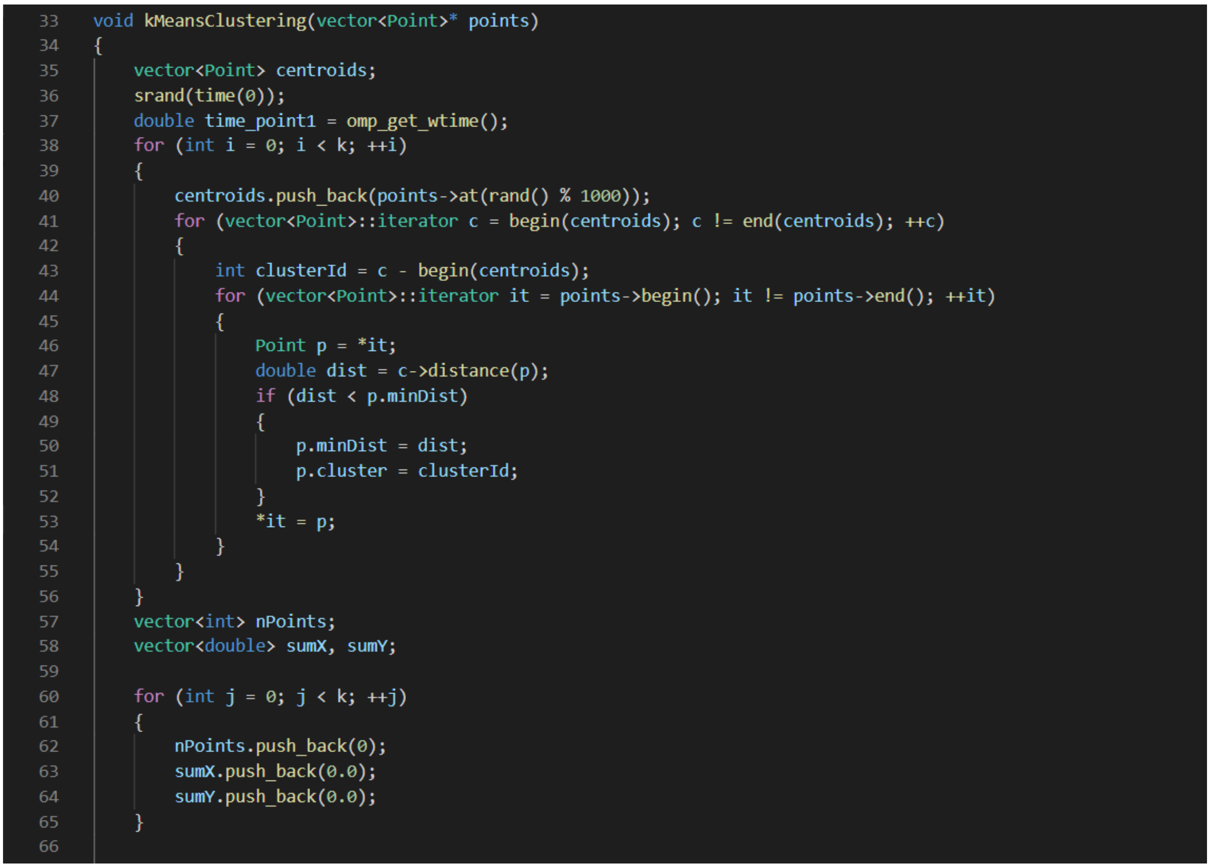
Roll No.: 181IT113

Sequential k-means Clustering

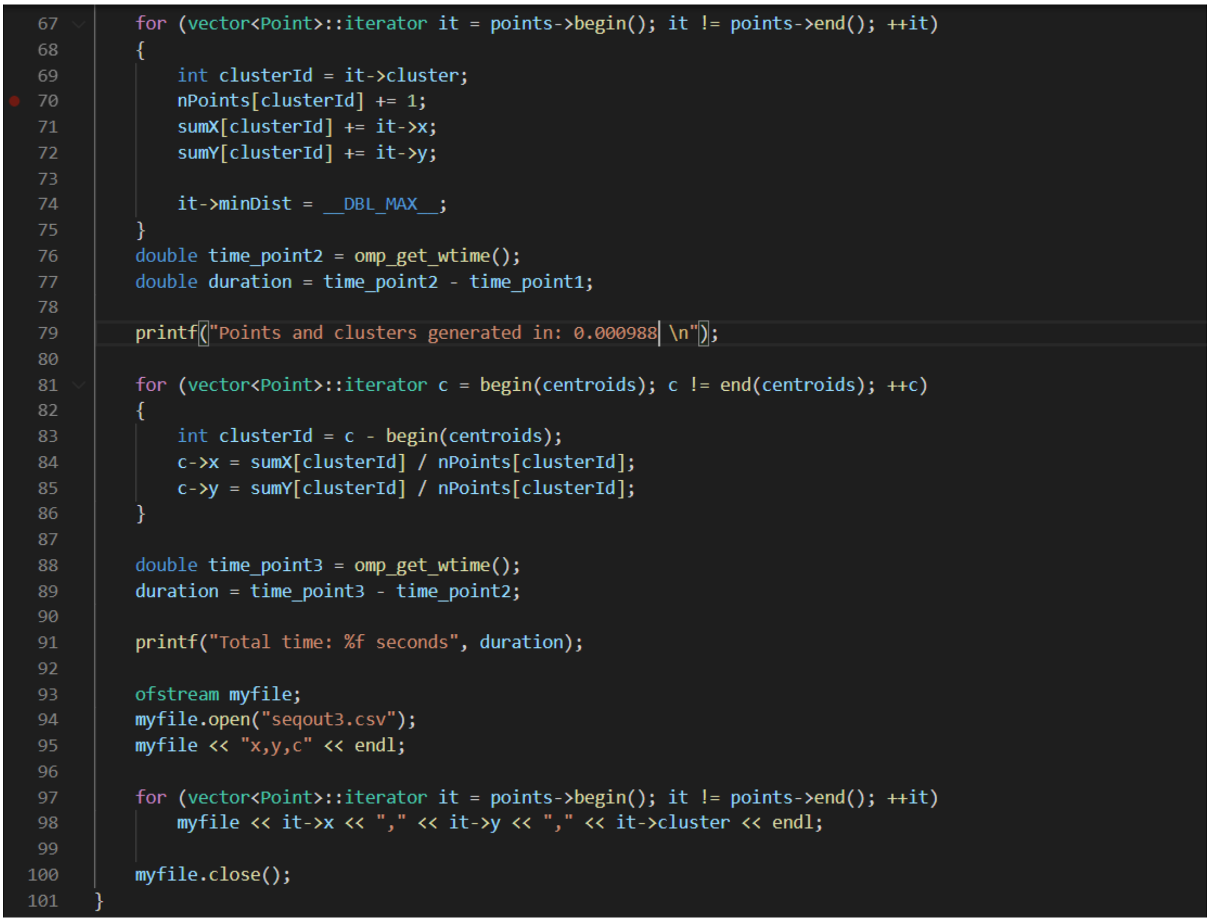


PC_Lab7.cpp > kMeansClustering(vector<Point>*)

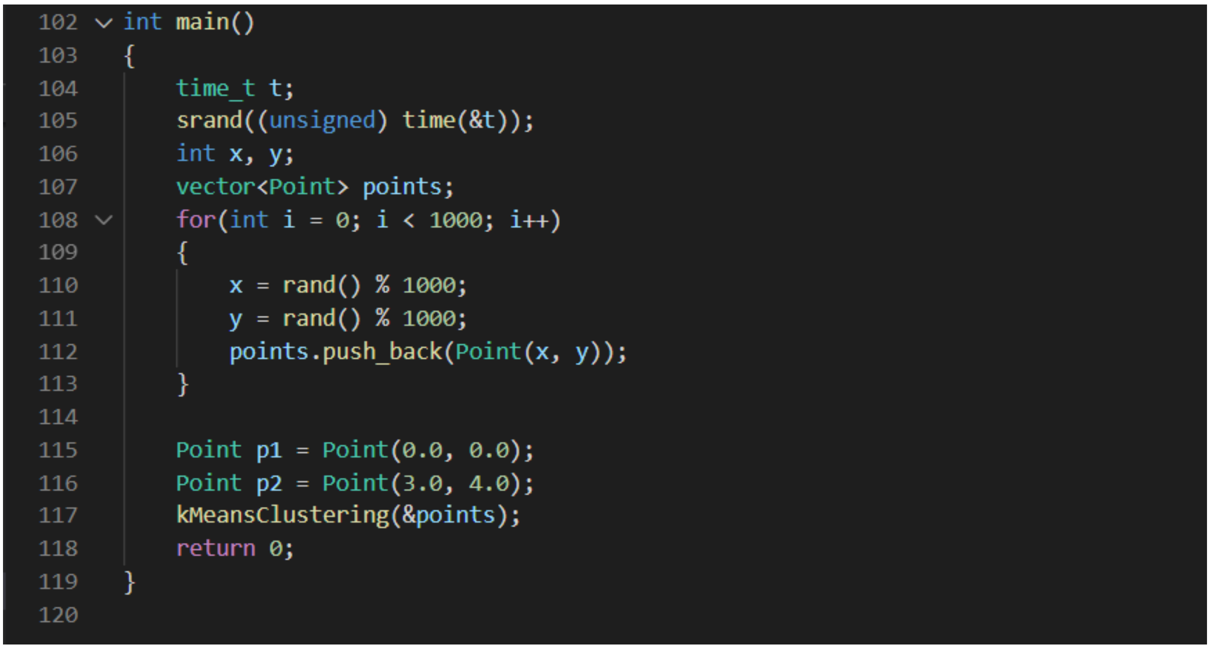
```cpp
#include <iostream>
#include <stdlib.h>
#include <cstdlib>
#include <time.h>
#include <vector>
#include <fstream>
#include <omp.h>
#define k 4
using namespace std;

struct Point {
    double x, y;
    int cluster;
    double minDist;

    Point() :
        x(0.0),
        y(0.0),
        cluster(-1),
        minDist(__DBL_MAX__) {}

    Point(double x, double y) :
        x(x),
        y(y),
        cluster(-1),
        minDist(__DBL_MAX__) {}

    double distance(Point p) {
        return (p.x - x) * (p.x - x) + (p.y - y) * (p.y - y);
    }
};

```



```cpp
void kMeansClustering(vector<Point>* points)
{
    vector<Point> centroids;
    srand(time(0));
    double time_point1 = omp_get_wtime();
    for (int i = 0; i < k; ++i)
    {
        centroids.push_back(points->at(rand() % 1000));
        for (vector<Point>::iterator c = begin(centroids); c != end(centroids); ++c)
        {
            int clusterId = c - begin(centroids);
            for (vector<Point>::iterator it = points->begin(); it != points->end(); ++it)
            {
                Point p = *it;
                double dist = c->distance(p);
                if (dist < p.minDist)
                {
                    p.minDist = dist;
                    p.cluster = clusterId;
                }
                *it = p;
            }
        }
    }
    vector<int> nPoints;
    vector<double> sumX, sumY;

    for (int j = 0; j < k; ++j)
    {
        nPoints.push_back(0);
        sumX.push_back(0.0);
        sumY.push_back(0.0);
    }

```

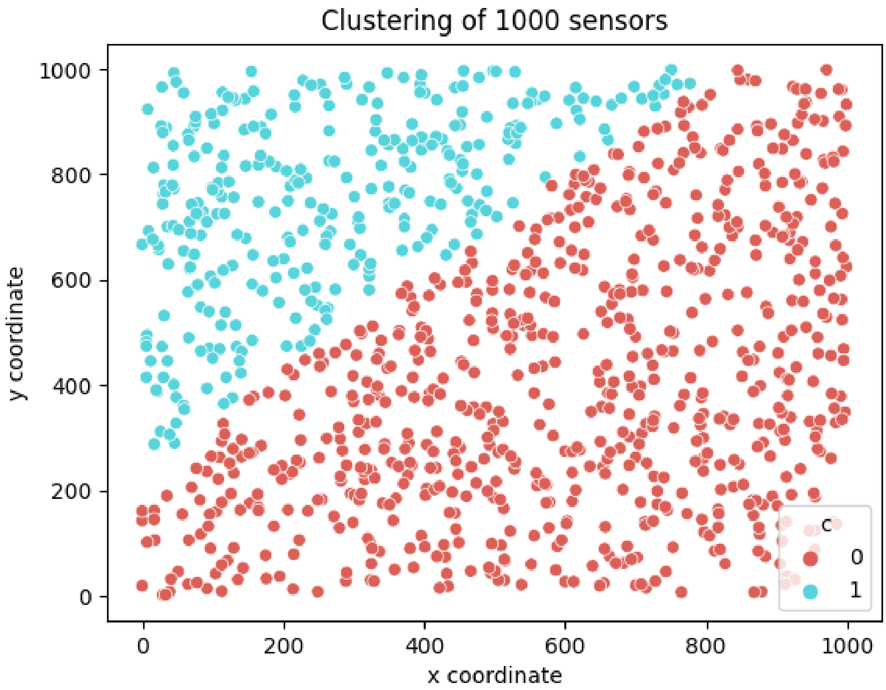


```cpp
    for (vector<Point>::iterator it = points->begin(); it != points->end(); ++it)
    {
        int clusterId = it->cluster;
        nPoints[clusterId] += 1;
        sumX[clusterId] += it->x;
        sumY[clusterId] += it->y;

        it->minDist = __DBL_MAX__;
    }
    double time_point2 = omp_get_wtime();
    double duration = time_point2 - time_point1;

    printf("Points and clusters generated in: 0.000988 \n");

    for (vector<Point>::iterator c = begin(centroids); c != end(centroids); ++c)
    {
        int clusterId = c - begin(centroids);
        c->x = sumX[clusterId] / nPoints[clusterId];
        c->y = sumY[clusterId] / nPoints[clusterId];
    }

    double time_point3 = omp_get_wtime();
    duration = time_point3 - time_point2;

    printf("Total time: %f seconds", duration);

    ofstream myfile;
    myfile.open("seqout3.csv");
    myfile << "x,y,c" << endl;

    for (vector<Point>::iterator it = points->begin(); it != points->end(); ++it)
        myfile << it->x << "," << it->y << "," << it->cluster << endl;

    myfile.close();
}
```

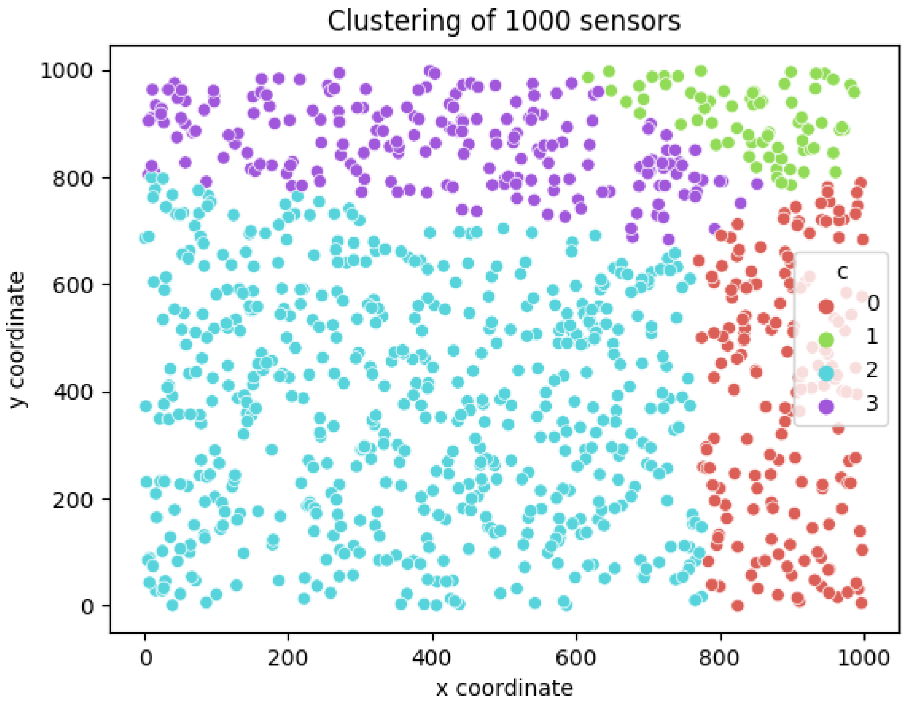


```
int main()
{
    time_t t;
    srand((unsigned) time(&t));
    int x, y;
    vector<Point> points;
    for(int i = 0; i < 1000; i++)
    {
        x = rand() % 1000;
        y = rand() % 1000;
        points.push_back(Point(x, y));
    }

    Point p1 = Point(0.0, 0.0);
    Point p2 = Point(3.0, 4.0);
    kMeansClustering(&points);
    return 0;
}
```

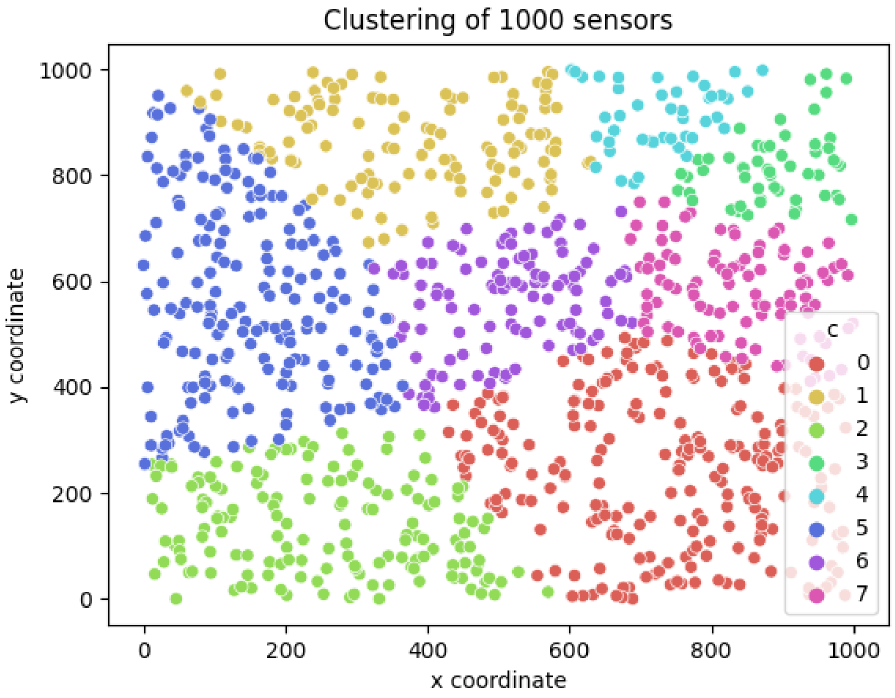
Graphs:



K = 2

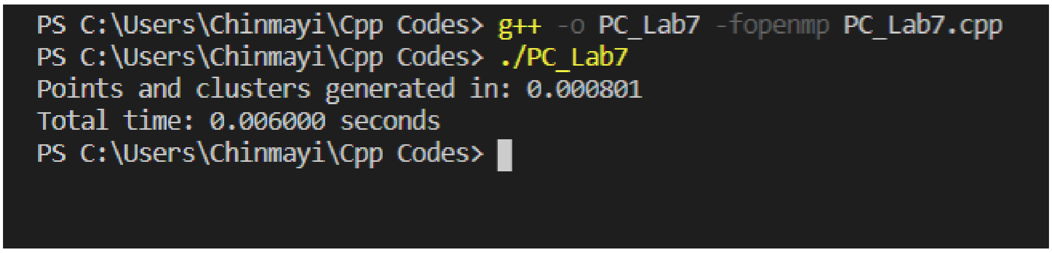

Clustering of 1000 sensors
y coordinate
x coordinate
0
200
400
600
800
1000
0
200
400
600
800
1000
c
0
1
2
3

K = 4


Clustering of 1000 sensors
y coordinate
x coordinate
0
200
400
600
800
1000
0
200
400
600
800
1000
c
0
1
2
3
4
5
6
7

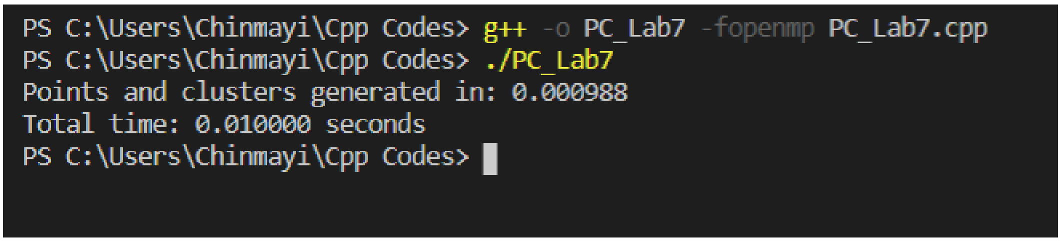
K = 8

Output:



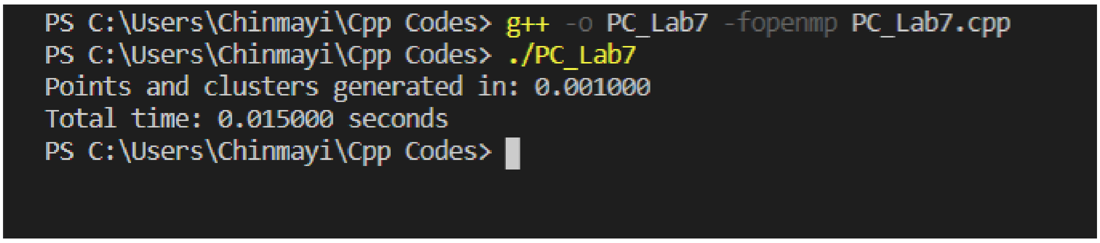
```
PS C:\Users\Chinmayi\Cpp Codes> g++ -o PC_Lab7 -fopenmp PC_Lab7.cpp
PS C:\Users\Chinmayi\Cpp Codes> ./PC_Lab7
Points and clusters generated in: 0.000801
Total time: 0.006000 seconds
PS C:\Users\Chinmayi\Cpp Codes> 
```

K =2



```
PS C:\Users\Chinmayi\Cpp Codes> g++ -o PC_Lab7 -fopenmp PC_Lab7.cpp
PS C:\Users\Chinmayi\Cpp Codes> ./PC_Lab7
Points and clusters generated in: 0.000988
Total time: 0.010000 seconds
PS C:\Users\Chinmayi\Cpp Codes> 
```

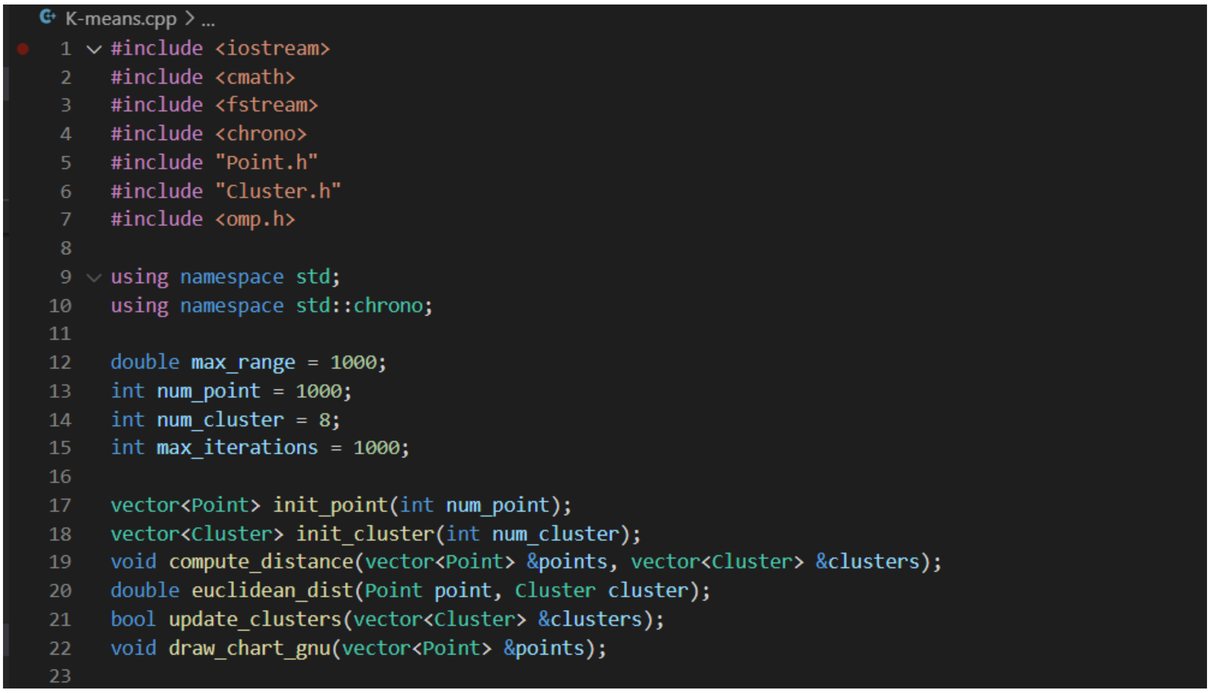
K = 4



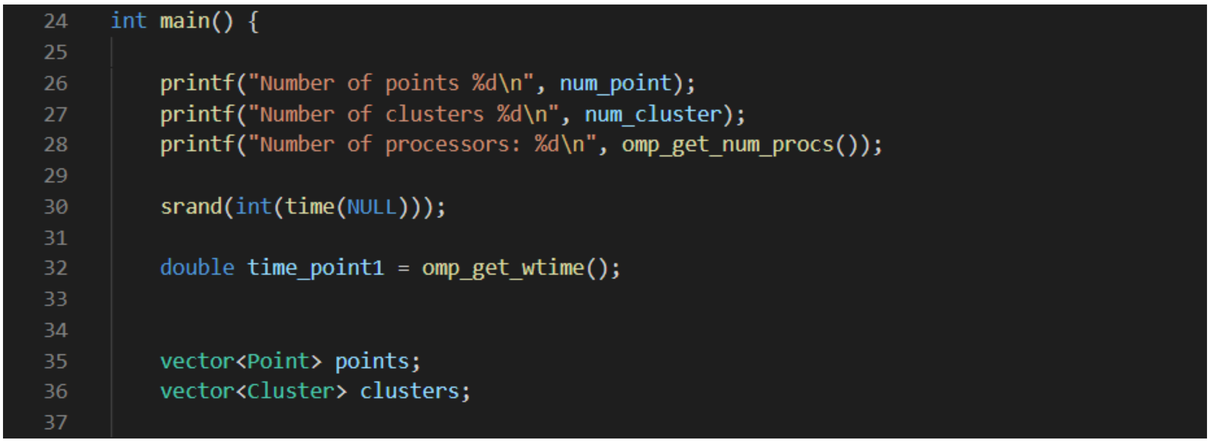
```
PS C:\Users\Chinmayi\Cpp Codes> g++ -o PC_Lab7 -fopenmp PC_Lab7.cpp
PS C:\Users\Chinmayi\Cpp Codes> ./PC_Lab7
Points and clusters generated in: 0.001000
Total time: 0.015000 seconds
PS C:\Users\Chinmayi\Cpp Codes> 
```

K = 8

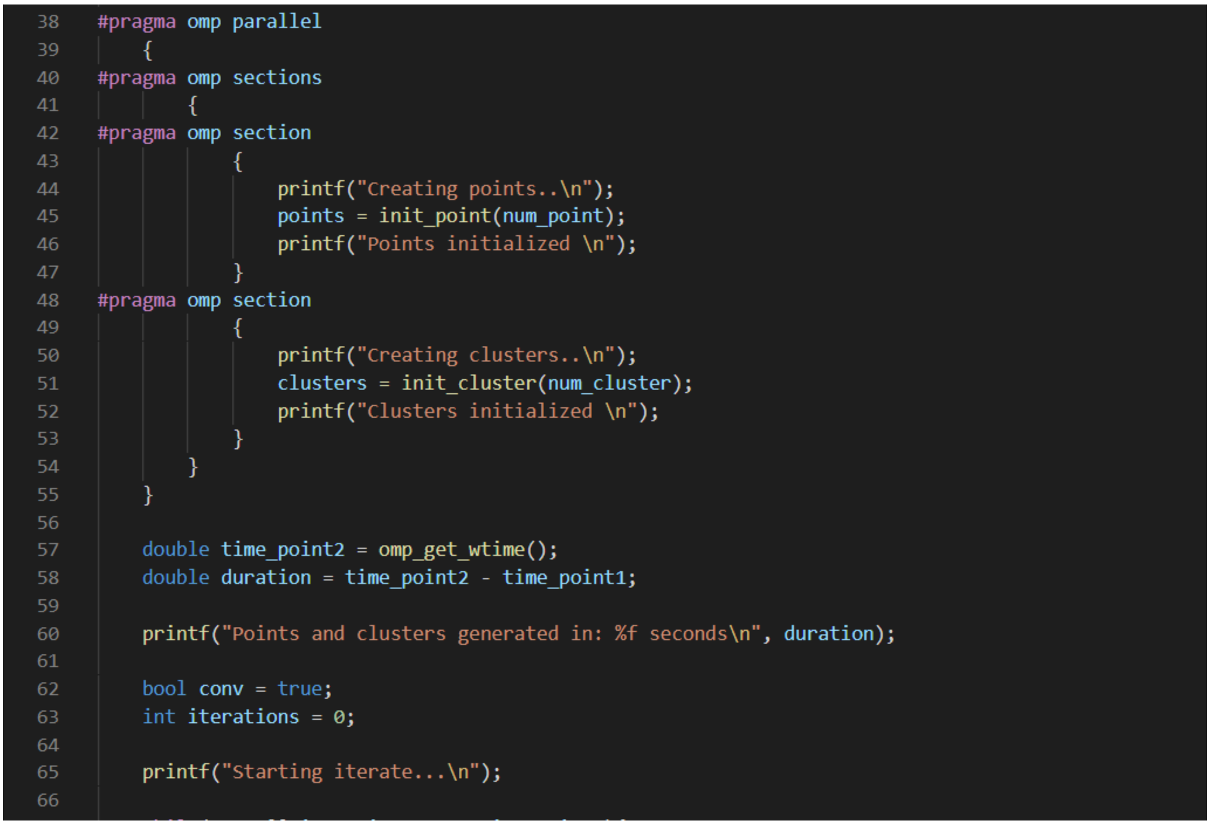
Parallel K-means Clustering:



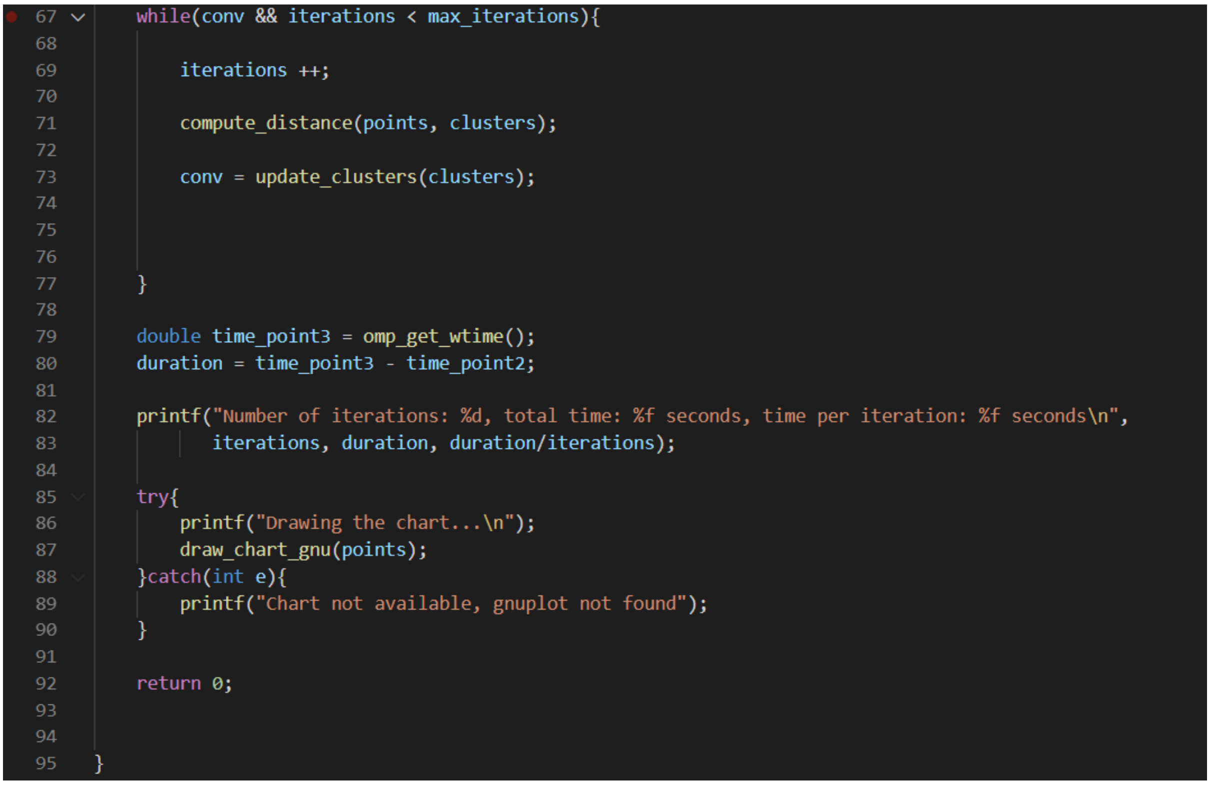
```cpp
#include <iostream>
#include <cmath>
#include <fstream>
#include <chrono>
#include "Point.h"
#include "Cluster.h"
#include <omp.h>

using namespace std;
using namespace std::chrono;

double max_range = 1000;
int num_point = 1000;
int num_cluster = 8;
int max_iterations = 1000;

vector<Point> init_point(int num_point);
vector<Cluster> init_cluster(int num_cluster);
void compute_distance(vector<Point> &points, vector<Cluster> &clusters);
double euclidean_dist(Point point, Cluster cluster);
bool update_clusters(vector<Cluster> &clusters);
void draw_chart_gnu(vector<Point> &points);

```



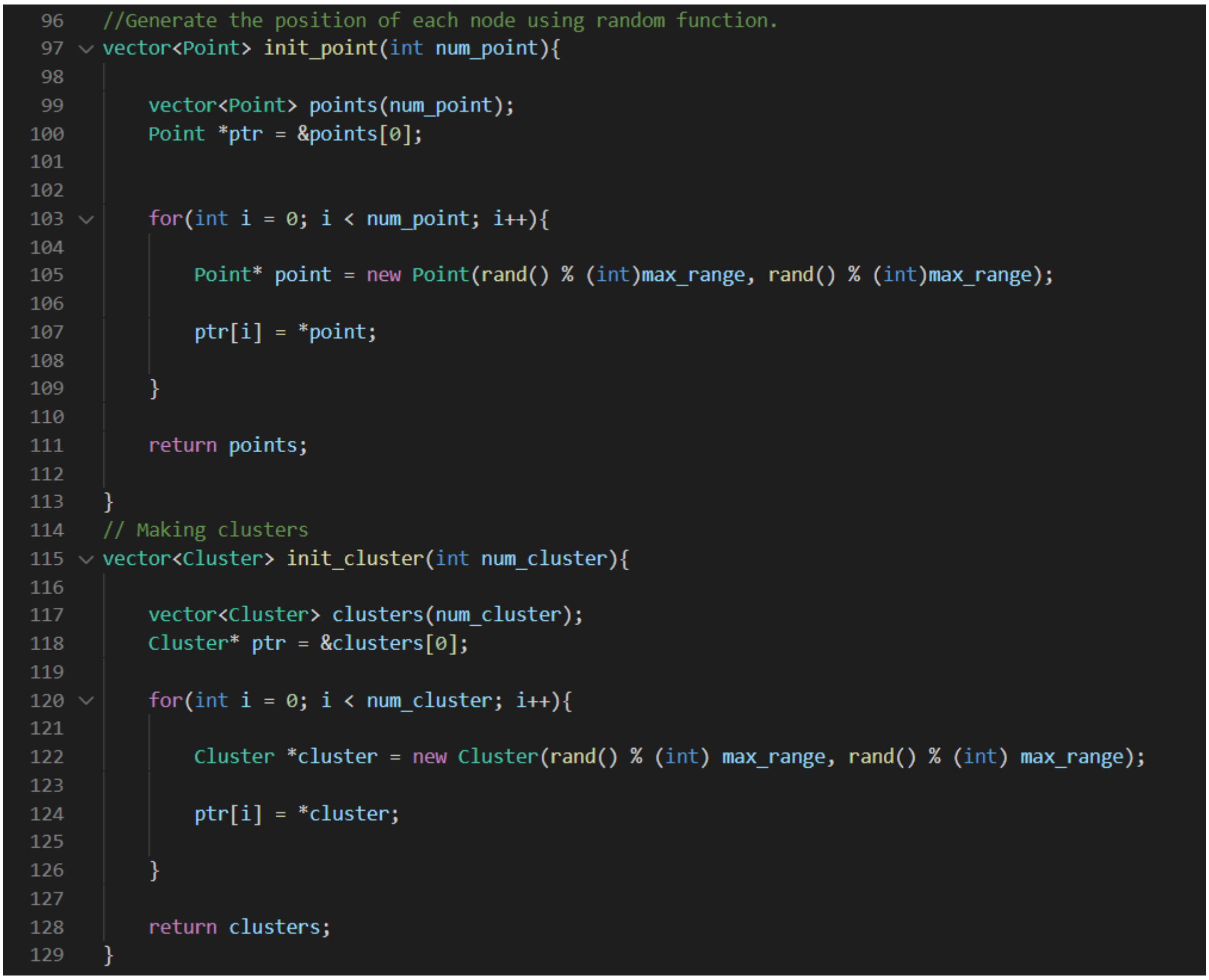
```
int main() {

    printf("Number of points %d\n", num_point);
    printf("Number of clusters %d\n", num_cluster);
    printf("Number of processors: %d\n", omp_get_num_procs());

    srand(int(time(NULL)));

    double time_point1 = omp_get_wtime();


    vector<Point> points;
    vector<Cluster> clusters;

```



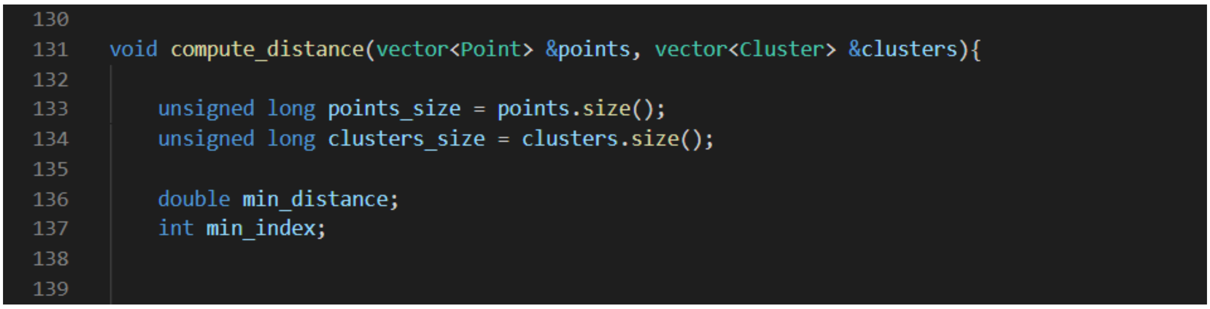
```
#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            {
                printf("Creating points..\n");
                points = init_point(num_point);
                printf("Points initialized \n");
            }
#pragma omp section
            {
                printf("Creating clusters..\n");
                clusters = init_cluster(num_cluster);
                printf("Clusters initialized \n");
            }
        }
    }

    double time_point2 = omp_get_wtime();
    double duration = time_point2 - time_point1;

    printf("Points and clusters generated in: %f seconds\n", duration);

    bool conv = true;
    int iterations = 0;

    printf("Starting iterate...\n");

```



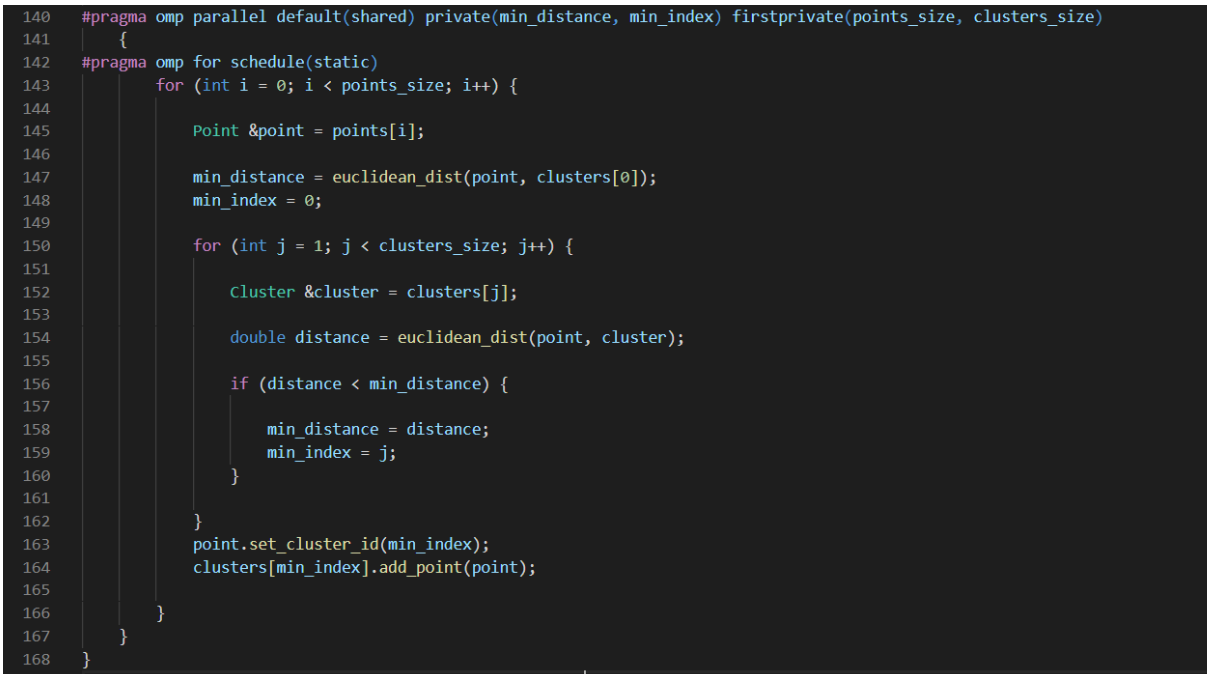
```cpp
    while(conv && iterations < max_iterations){

        iterations ++;

        compute_distance(points, clusters);

        conv = update_clusters(clusters);



    }

    double time_point3 = omp_get_wtime();
    duration = time_point3 - time_point2;

    printf("Number of iterations: %d, total time: %f seconds, time per iteration: %f seconds\n",
           iterations, duration, duration/iterations);

    try{
        printf("Drawing the chart...\n");
        draw_chart_gnu(points);
    }catch(int e){
        printf("Chart not available, gnuplot not found");
    }

    return 0;


}
```



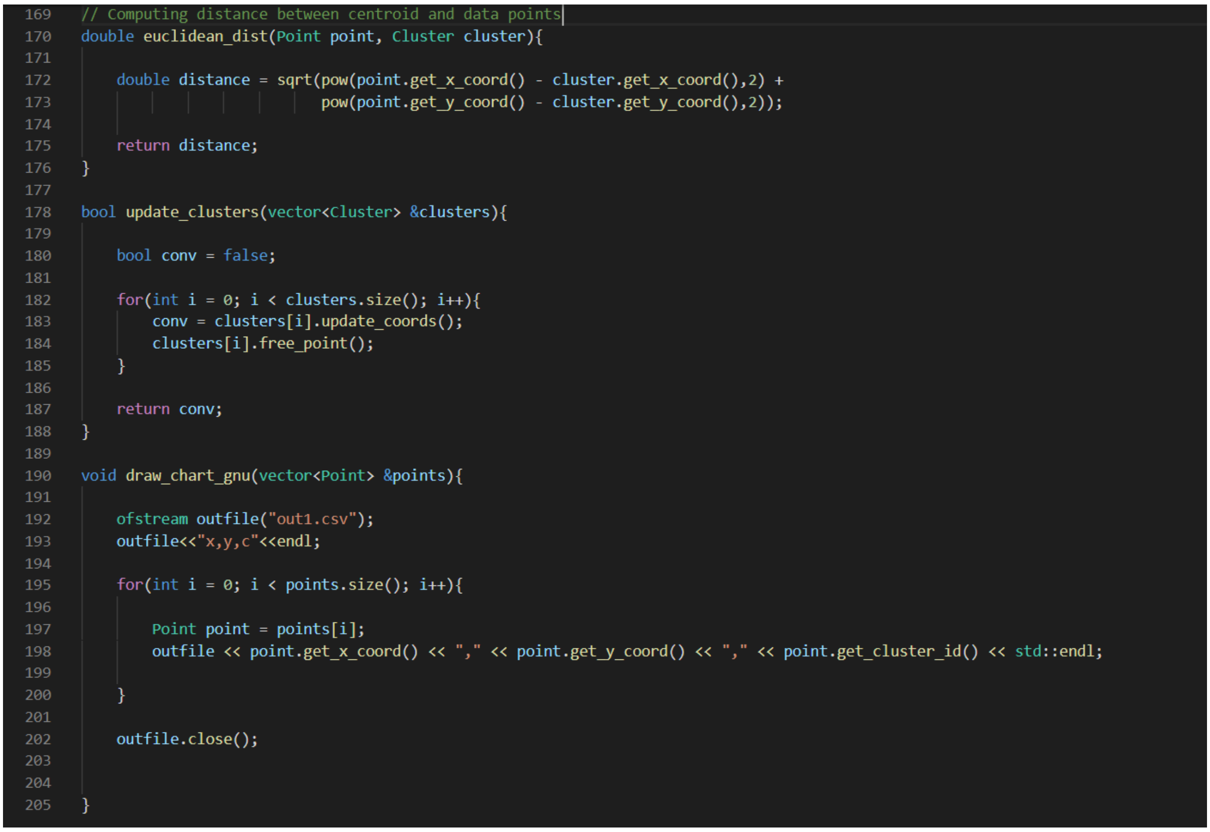
```cpp
//Generate the position of each node using random function.
vector<Point> init_point(int num_point){

    vector<Point> points(num_point);
    Point *ptr = &points[0];


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

        Point* point = new Point(rand() % (int)max_range, rand() % (int)max_range);

        ptr[i] = *point;

    }

    return points;

}
// Making clusters
vector<Cluster> init_cluster(int num_cluster){

    vector<Cluster> clusters(num_cluster);
    Cluster* ptr = &clusters[0];

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

        Cluster *cluster = new Cluster(rand() % (int) max_range, rand() % (int) max_range);

        ptr[i] = *cluster;

    }

    return clusters;
}
```



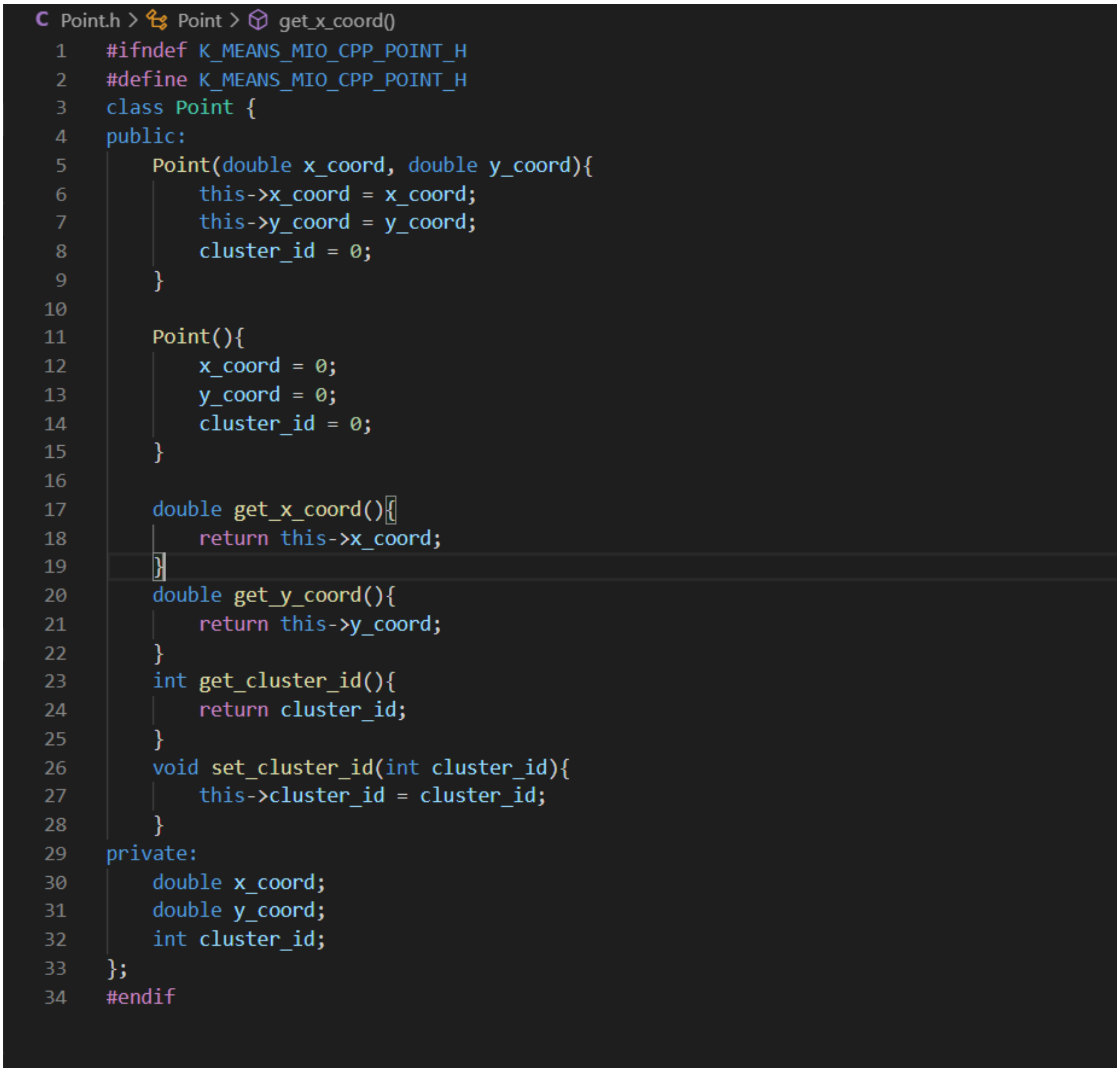
```
void compute_distance(vector<Point> &points, vector<Cluster> &clusters){

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

    double min_distance;
    int min_index;

```



```
#pragma omp parallel default(shared) private(min_distance, min_index) firstprivate(points_size, clusters_size)
    {
#pragma omp for schedule(static)
        for (int i = 0; i < points_size; i++) {

            Point &point = points[i];

            min_distance = euclidean_dist(point, clusters[0]);
            min_index = 0;

            for (int j = 1; j < clusters_size; j++) {

                Cluster &cluster = clusters[j];

                double distance = euclidean_dist(point, cluster);

                if (distance < min_distance) {

                    min_distance = distance;
                    min_index = j;
                }

            }
            point.set_cluster_id(min_index);
            clusters[min_index].add_point(point);

        }
    }
}
```

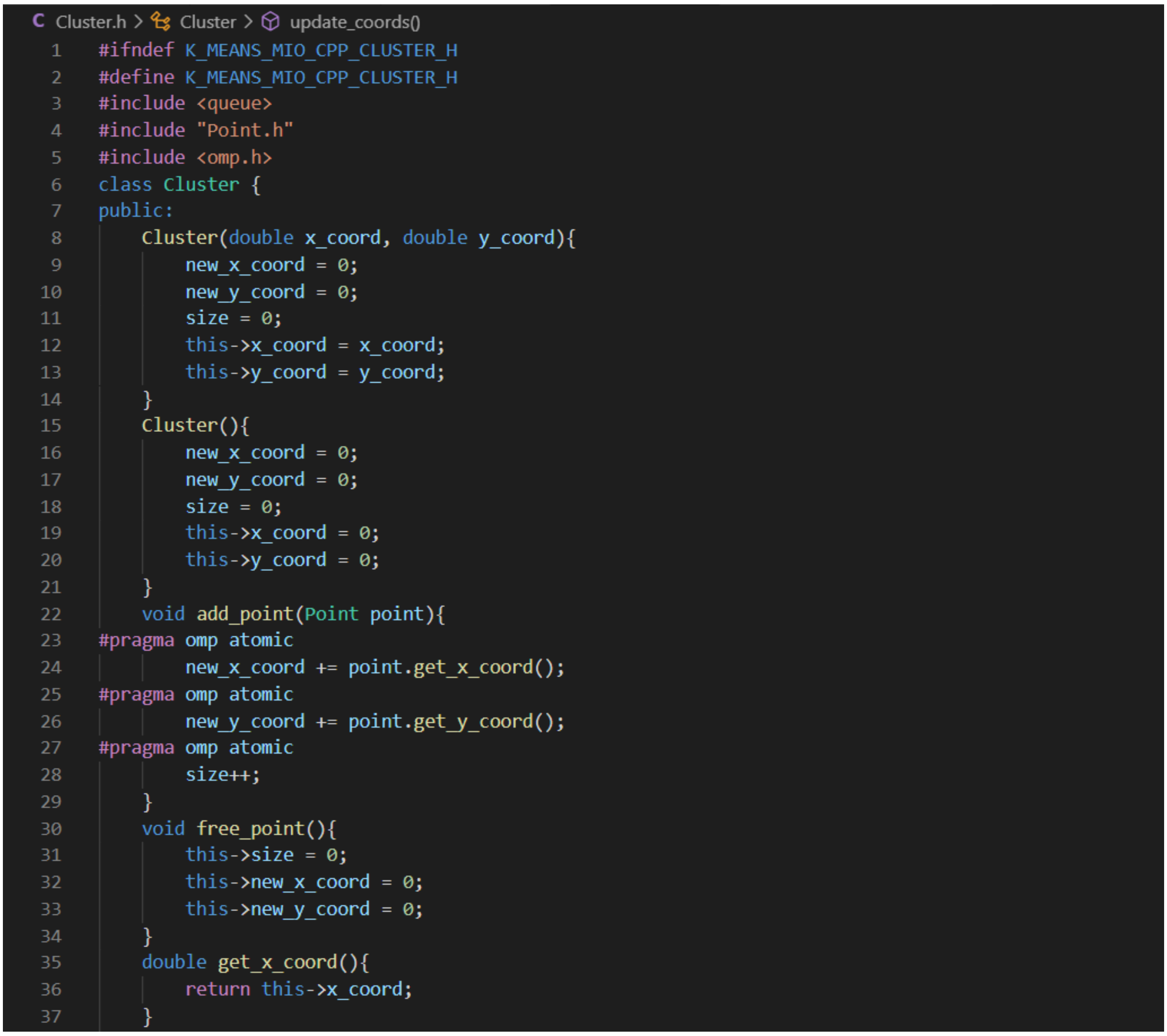


```
// Computing distance between centroid and data points
double euclidean_dist(Point point, Cluster cluster){

    double distance = sqrt(pow(point.get_x_coord() - cluster.get_x_coord(),2) +
                           pow(point.get_y_coord() - cluster.get_y_coord(),2));

    return distance;
}

bool update_clusters(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;
}

void draw_chart_gnu(vector<Point> &points){

    ofstream outfile("out1.csv");
    outfile<<"x,y,c"<<endl;

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

        Point point = points[i];
        outfile << point.get_x_coord() << "," << point.get_y_coord() << "," << point.get_cluster_id() << std::endl;

    }

    outfile.close();


}
```

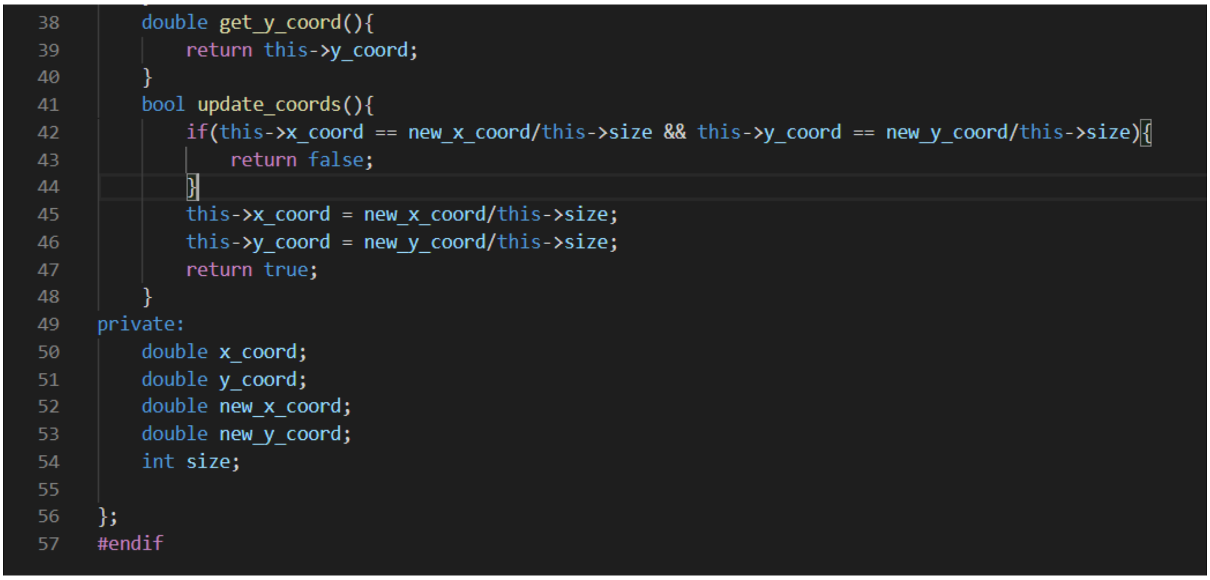


```cpp
#ifndef K_MEANS_MIO_CPP_POINT_H
#define K_MEANS_MIO_CPP_POINT_H
class Point {
public:
    Point(double x_coord, double y_coord){
        this->x_coord = x_coord;
        this->y_coord = y_coord;
        cluster_id = 0;
    }

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

    double get_x_coord(){
        return this->x_coord;
    }
    double get_y_coord(){
        return this->y_coord;
    }
    int get_cluster_id(){
        return cluster_id;
    }
    void set_cluster_id(int cluster_id){
        this->cluster_id = cluster_id;
    }
private:
    double x_coord;
    double y_coord;
    int cluster_id;
};
#endif
```

Point.h



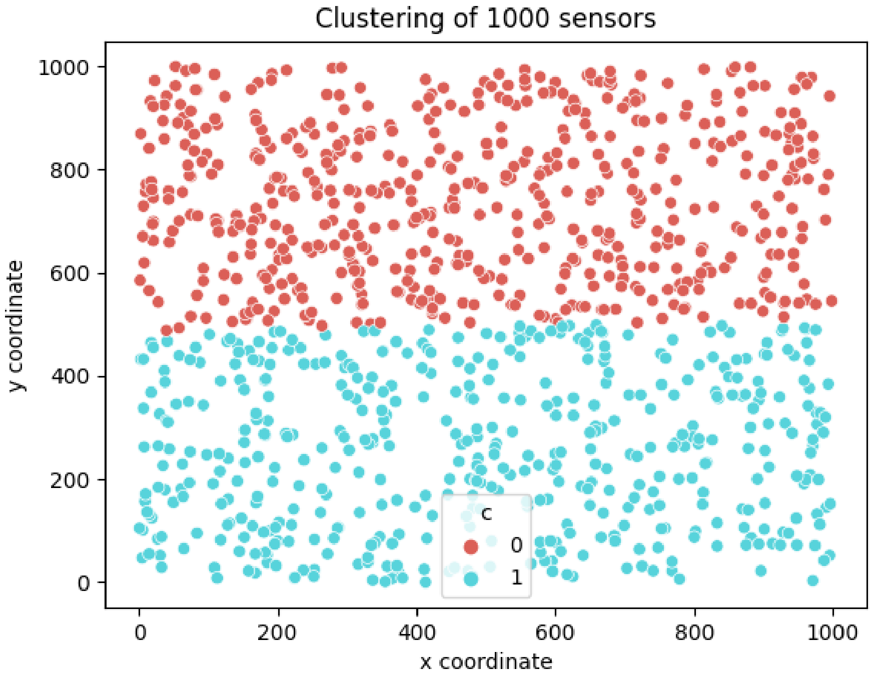
```cpp
#ifndef K_MEANS_MIO_CPP_CLUSTER_H
#define K_MEANS_MIO_CPP_CLUSTER_H
#include <queue>
#include "Point.h"
#include <omp.h>
class Cluster {
public:
    Cluster(double x_coord, double y_coord){
        new_x_coord = 0;
        new_y_coord = 0;
        size = 0;
        this->x_coord = x_coord;
        this->y_coord = y_coord;
    }
    Cluster(){
        new_x_coord = 0;
        new_y_coord = 0;
        size = 0;
        this->x_coord = 0;
        this->y_coord = 0;
    }
    void add_point(Point point){
#pragma omp atomic
        new_x_coord += point.get_x_coord();
#pragma omp atomic
        new_y_coord += point.get_y_coord();
#pragma omp atomic
        size++;
    }
    void free_point(){
        this->size = 0;
        this->new_x_coord = 0;
        this->new_y_coord = 0;
    }
    double get_x_coord(){
        return this->x_coord;
    }
```



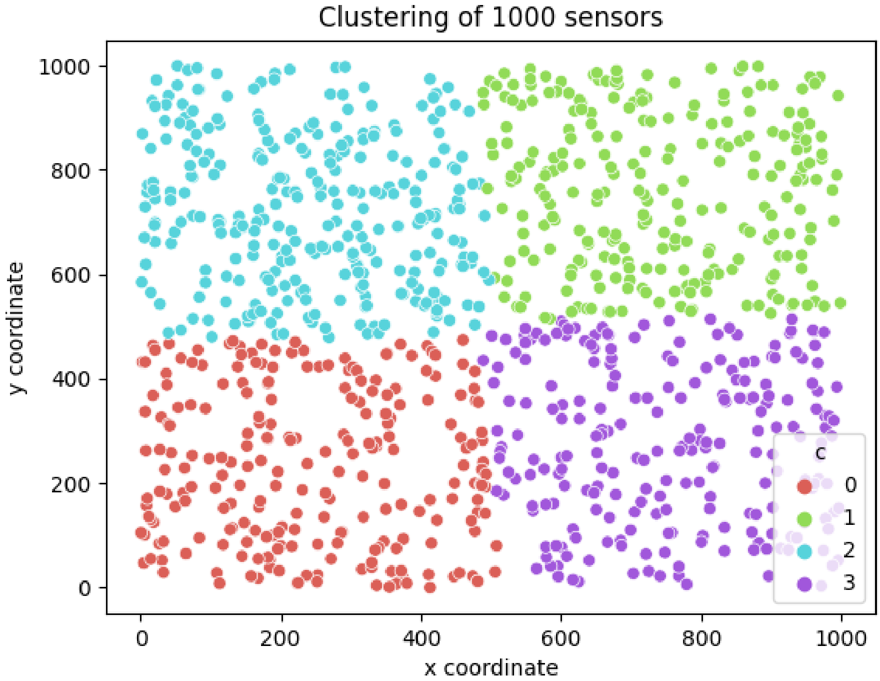
```cpp
    double get_y_coord(){
        return this->y_coord;
    }
    bool update_coords(){
        if(this->x_coord == new_x_coord/this->size && this->y_coord == new_y_coord/this->size){
            return false;
        }
        this->x_coord = new_x_coord/this->size;
        this->y_coord = new_y_coord/this->size;
        return true;
    }
private:
    double x_coord;
    double y_coord;
    double new_x_coord;
    double new_y_coord;
    int size;

};
#endif
```

Cluster.h

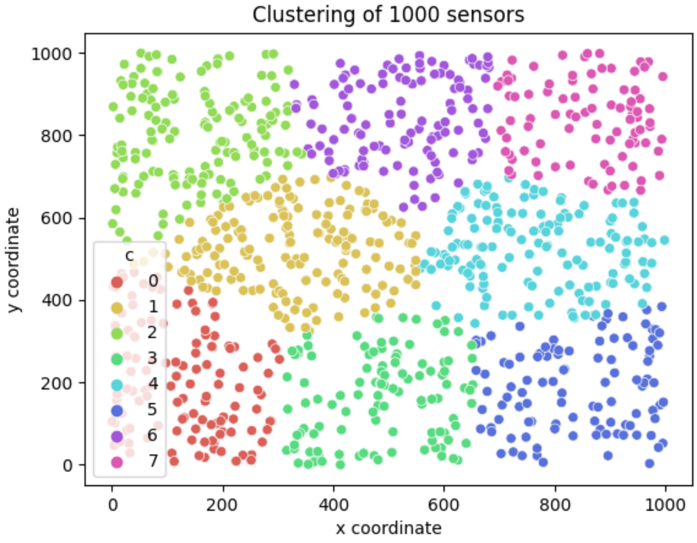
Graphs:


Clustering of 1000 sensors
y coordinate
x coordinate
0
200
400
600
800
1000
0
200
400
600
800
1000
c
0
1

K = 2

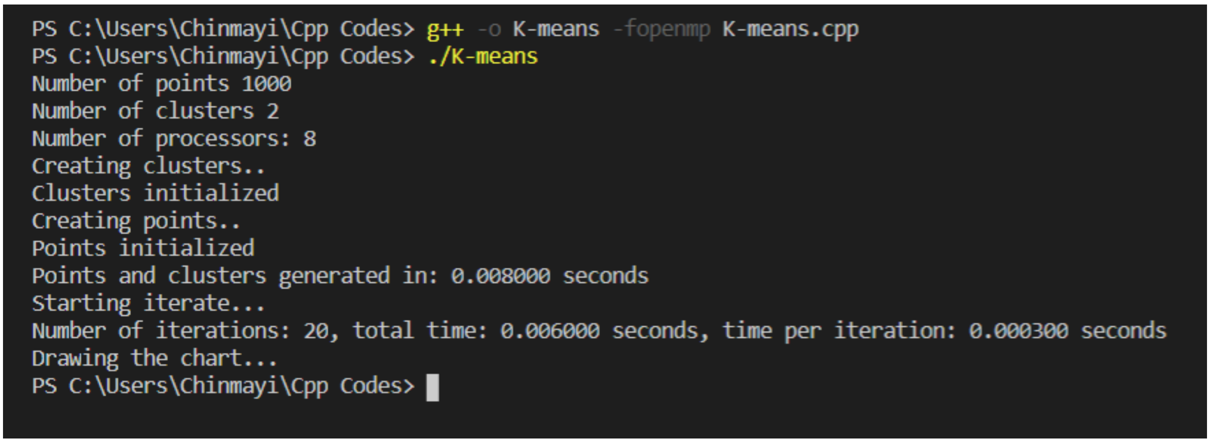

Clustering of 1000 sensors
y coordinate
x coordinate
0
200
400
600
800
1000
0
200
400
600
800
1000
c
0
1
2
3

K = 4



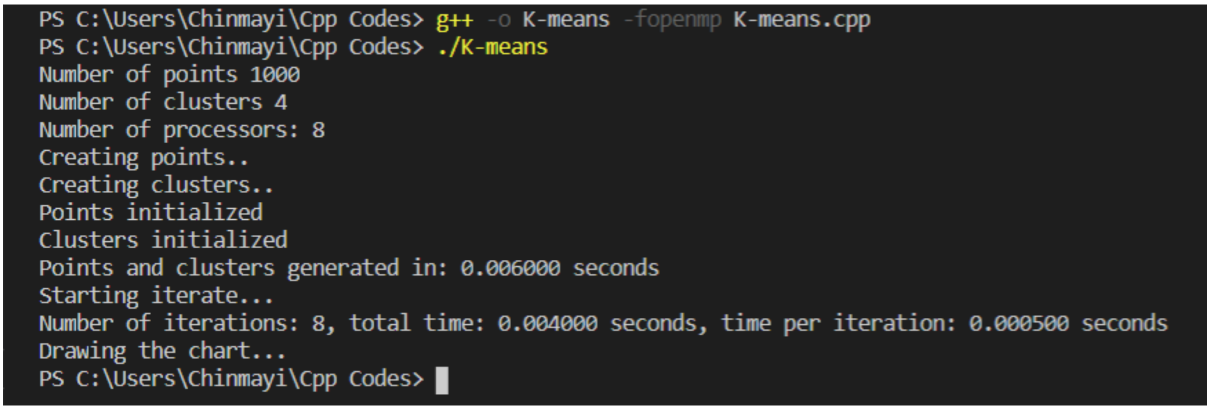
K = 8

Outputs :



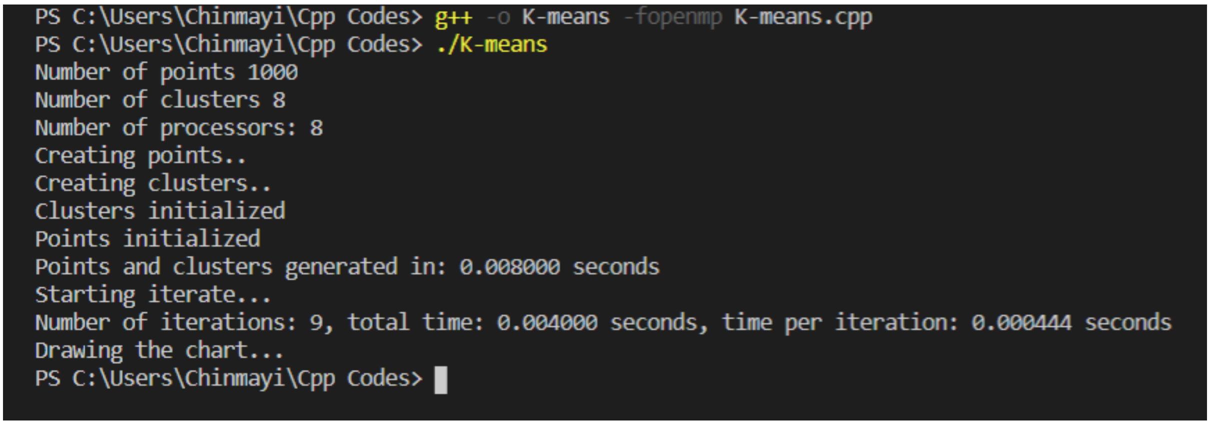
```
PS C:\Users\Chinmayi\Cpp Codes> g++ -o K-means -fopenmp K-means.cpp
PS C:\Users\Chinmayi\Cpp Codes> ./K-means
Number of points 1000
Number of clusters 2
Number of processors: 8
Creating clusters..
Clusters initialized
Creating points..
Points initialized
Points and clusters generated in: 0.008000 seconds
Starting iterate...
Number of iterations: 20, total time: 0.006000 seconds, time per iteration: 0.000300 seconds
Drawing the chart...
PS C:\Users\Chinmayi\Cpp Codes>
```

K = 2



```
PS C:\Users\Chinmayi\Cpp Codes> g++ -o K-means -fopenmp K-means.cpp
PS C:\Users\Chinmayi\Cpp Codes> ./K-means
Number of points 1000
Number of clusters 4
Number of processors: 8
Creating points..
Creating clusters..
Points initialized
Clusters initialized
Points and clusters generated in: 0.006000 seconds
Starting iterate...
Number of iterations: 8, total time: 0.004000 seconds, time per iteration: 0.000500 seconds
Drawing the chart...
PS C:\Users\Chinmayi\Cpp Codes>
```

K = 4



```
PS C:\Users\Chinmayi\Cpp Codes> g++ -o K-means -fopenmp K-means.cpp
PS C:\Users\Chinmayi\Cpp Codes> ./K-means
Number of points 1000
Number of clusters 8
Number of processors: 8
Creating points..
Creating clusters..
Clusters initialized
Points initialized
Points and clusters generated in: 0.008000 seconds
Starting iterate...
Number of iterations: 9, total time: 0.004000 seconds, time per iteration: 0.000444 seconds
Drawing the chart...
PS C:\Users\Chinmayi\Cpp Codes>
```

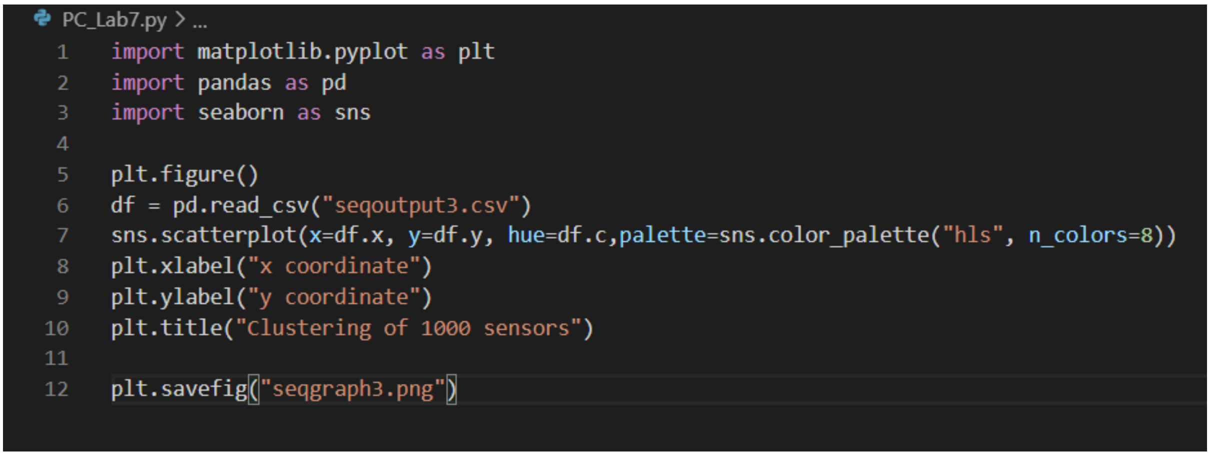
K =8

Different sections have been used to allot the section to threads without encountering race conditions.

The two for loops have been declared as two sections.

Schedule(static) has been used to compute distances to allot fixed number of chunks to the threads. This ensures that each thread gets some amount of task and they can work efficiently.

For greater number of clustering, parallel execution takes lesser time than sequential execution.



```python
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

plt.figure()
df = pd.read_csv("seqoutput3.csv")
sns.scatterplot(x=df.x, y=df.y, hue=df.c,palette=sns.color_palette("hls", n_colors=8))
plt.xlabel("x coordinate")
plt.ylabel("y coordinate")
plt.title("Clustering of 1000 sensors")

plt.savefig("seqgraph3.png")
```

For plotting the graph