NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL
DEPARTMENT OF INFORMATION TECHNOLOGY
IT 301 Parallel Computing LAB 7

P Akshara - 181IT132

1.Consider random deployment of sensor nodes in field to sense the environment. The nodes are deployed randomly and the position of each sensor node is sent to centralised server. The server would like to cluster these nodes. Use K-means algorithm to cluster the nodes. Write an OpenMP program to cluster the sensor nodes and compare the result with sequential and parallel approach.

Helper files:
Header file for Point Class

```
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;
};
```

Header file for Cluster Class

```
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;
}

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;
};
```

Sequential

```
#include <iostream>
#include <cmath>
#include <fstream>
```

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

using namespace std;
using namespace std::chrono;

double max_range = 100000;
int num_point = 10000;
int num_cluster;
int max_iterations = 30;

// Initializing nodes
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;
}

// Initializing 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;
}

// Euclidean distance computation
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;
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;
}
}
points[i].set_cluster_id(min_index);
clusters[min_index].add_point(points[i]);
}
}

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;
}

// For each cluster, update the coords. If only one cluster moves, conv will be TRUE
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;
}

// Plot with gnuplot
void draw_chart_gnu(vector<Point> &points)
{
ofstream outfile("plotter.txt");
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();
system("gnuplot -p -e \"plot 'plotter.txt' using 1:2:3 with points palette notitle\"");
remove("plotter.txt");
}

int main()
{
srand(time(NULL));
printf("Enter no. of clusters needed..\n");
scanf("%d", &num_cluster);
printf("Number of nodes %d\n", num_point);
printf("Number of clusters %d\n", num_cluster);

double time_point1 = omp_get_wtime();

printf("Starting random initialization..\n");
vector<Point> points = init_point(num_point);
printf("Nodes initialized \n");
vector<Cluster> clusters = init_cluster(num_cluster);
printf("Clusters initialized \n");

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

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

bool conv = true;
int iterations = 0;
```

```
printf("Starting K-means..\n");

// Stopping conditions: Max iterations achieved or clusters don't move
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 to converge: %d, total time: %f seconds, time per iteration: %f seconds\n",
iterations, duration, duration / iterations);
try
{
printf("Plotting nodes...\n");
draw_chart_gnu(points);
}
catch (int e)
{
printf("Chart not available, gnuplot not found");
}
return 0;
}
```

Parallelized

```
#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 = 100000;
int num_point = 10000;
int num_cluster;
int max_iterations = 30;

// Initializing nodes
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;
}
```

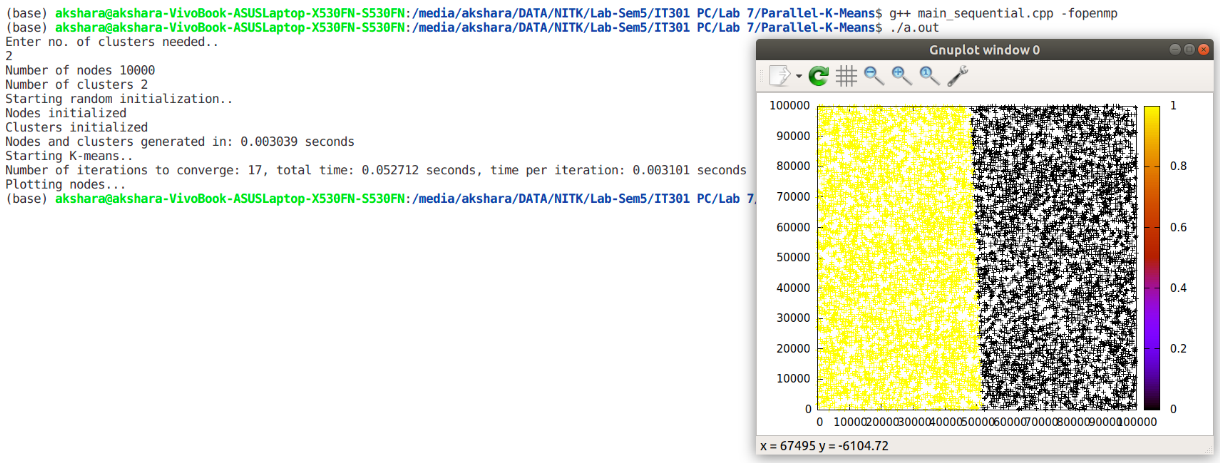
```
// Initializing Clusters
vector<Cluster> init_cluster(int num_cluster)
{
vector<Cluster> clusters(num_cluster);
Cluster *ptr = &clusters[0];
for (int i = 0; i < num_cluster; i++)
{
Cluster *cluster = new Cluster(rand() % (int)max_range, rand() % (int)max_range);
ptr[i] = *cluster;
}
return clusters;
}

// Euclidean distance computation
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);
}
}
}

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;
}

// A parallel for chosen for each cluster with lastprivate=conv
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;
}
```

```
// Plot with gnuplot
void draw_chart_gnu(vector<Point> &points)
{
ofstream outfile("plotter.txt");
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();
system("gnuplot -p -e \"plot 'plotter.txt' using 1:2:3 with points palette notitle\"");
remove("plotter.txt");
}

int main()
{
printf("Enter no. of clusters needed..\n");
scanf("%d", &num_cluster);
printf("Number of nodes %d\n", num_point);
printf("Number of clusters %d\n", num_cluster);

srand(int(time(NULL)));

double time_point1 = omp_get_wtime();
printf("Starting random initialization..\n");

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

// Parallel initialization of Cluster and Point
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
{
points = init_point(num_point);
printf("Points initialized \n");
}
#pragma omp section
{
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 K-means...\n");

// The algorithm stops when iterations > max_iteration or when the clusters didn't move
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 to converge: %d, total time: %f seconds, time per iteration: %f seconds\n",
iterations, duration, duration / iterations);

try
{
printf("Plotting nodes...\n");
draw_chart_gnu(points);
}
catch (int e)
{
printf("Chart not available, gnuplot not found");
}
return 0;
}
```

Outputs

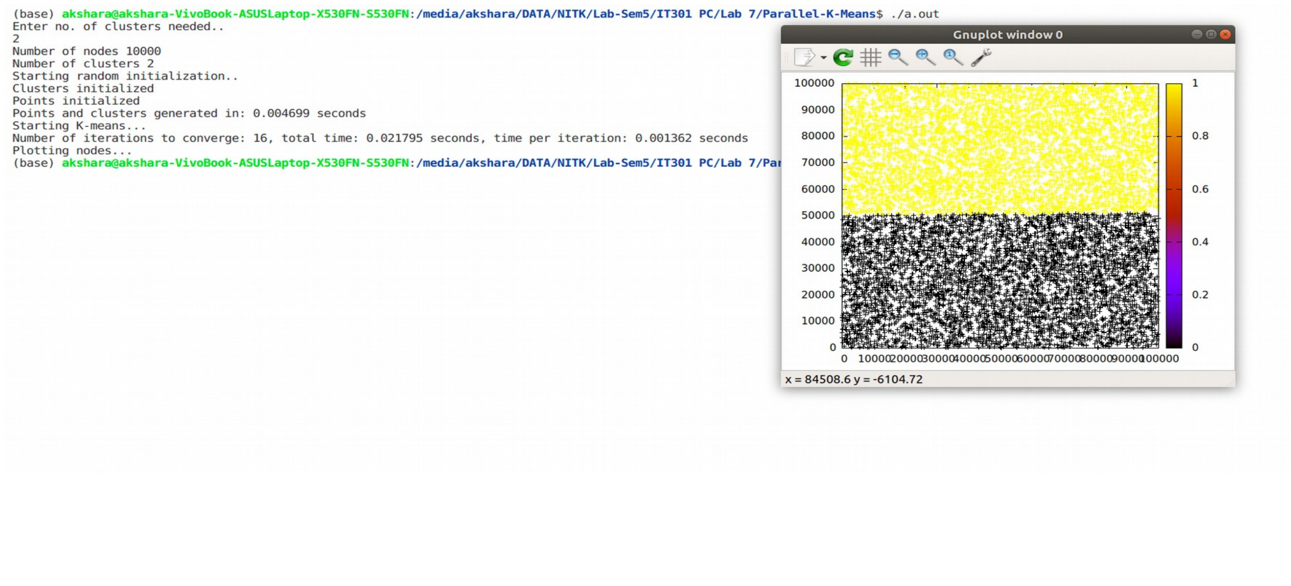
2 clusters

a) Sequential



```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Parallel-K-Means$ g++ main_sequential.cpp -fopenmp
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Parallel-K-Means$ ./a.out
Enter no. of clusters needed..
2
Number of nodes 10000
Number of clusters 2
Starting random initialization..
Nodes initialized
Clusters initialized
Nodes and clusters generated in: 0.003039 seconds
Starting K-means..
Number of iterations to converge: 17, total time: 0.052712 seconds, time per iteration: 0.003101 seconds
Plotting nodes...
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7
```

b) Parallel

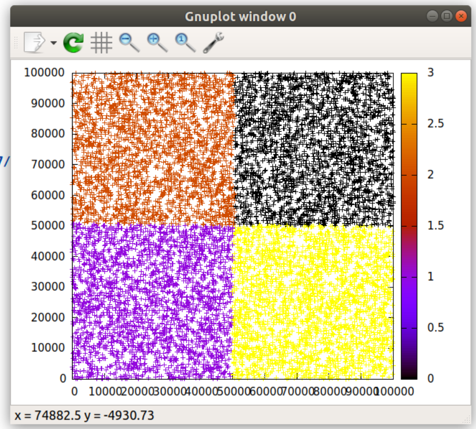


```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Parallel-K-Means$ ./a.out
Enter no. of clusters needed..
2
Number of nodes 10000
Number of clusters 2
Starting random initialization..
Clusters initialized
Points initialized
Points and clusters generated in: 0.004699 seconds
Starting K-means...
Number of iterations to converge: 16, total time: 0.021795 seconds, time per iteration: 0.001362 seconds
Plotting nodes...
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Par
```

4 clusters

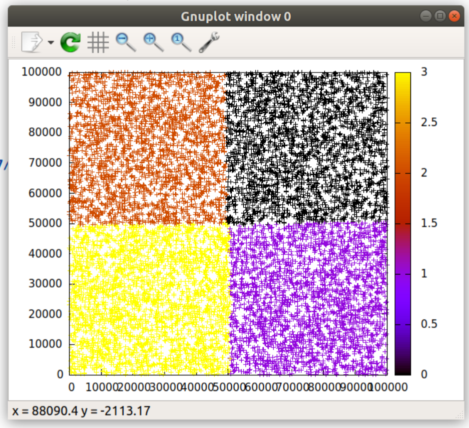
a) Sequential

```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Parallel-K-Means$ ./a.out
Enter no. of clusters needed..
4
Number of nodes 10000
Number of clusters 4
Starting random initialization..
Nodes initialized
Clusters initialized
Nodes and clusters generated in: 0.003050 seconds
Starting K-means..
Number of iterations to converge: 12, total time: 0.053969 seconds, time per iteration: 0.004497 seconds
Plotting nodes...
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/
```



b) Parallel

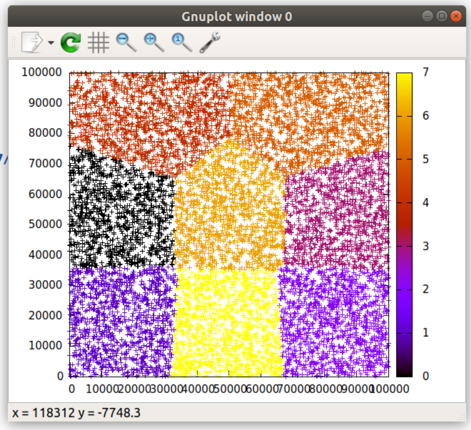
```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Parallel-K-Means$ ./a.out
Enter no. of clusters needed..
4
Number of nodes 10000
Number of clusters 4
Starting random initialization..
Clusters initialized
Points initialized
Points and clusters generated in: 0.005264 seconds
Starting K-means...
Number of iterations to converge: 14, total time: 0.018036 seconds, time per iteration: 0.001288 seconds
Plotting nodes...
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/
```



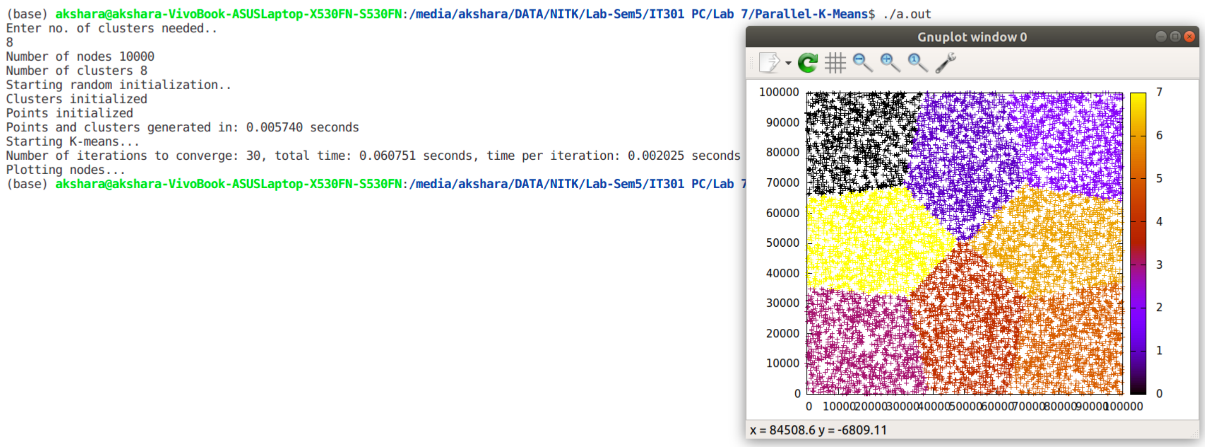
8 Clusters

a) Sequential

```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Parallel-K-Means$ ./a.out
Enter no. of clusters needed..
8
Number of nodes 10000
Number of clusters 8
Starting random initialization..
Nodes initialized
Clusters initialized
Nodes and clusters generated in: 0.003903 seconds
Starting K-means..
Number of iterations to converge: 30, total time: 0.123582 seconds, time per iteration: 0.004119 seconds
Plotting nodes...
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/
```



b) Parallel



```
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7/Parallel-K-Means$ ./a.out
Enter no. of clusters needed..
8
Number of nodes 10000
Number of clusters 8
Starting random initialization..
Clusters initialized
Points initialized
Points and clusters generated in: 0.005740 seconds
Starting K-means...
Number of iterations to converge: 30, total time: 0.060751 seconds, time per iteration: 0.002025 seconds
Plotting nodes...
(base) akshara@akshara-VivoBook-ASUSLaptop-X530FN-S530FN:/media/akshara/DATA/NITK/Lab-Sem5/IT301 PC/Lab 7
```

Analysis

Sl. No.	No. of clusters	Toal Sequential Time (s)	Total Parallel Time (s)
1	2	0.053	0.022
2	4	0.054	0.018
3	8	0.123	0.061

Table: Comparing Sequential and Parallel version

As seen in the above table, the parallel version is much faster than the sequential one for all cluster values taken.
The following parts of the sequential code have been parallelized which leads to the improvement:
1. Random initialization of the nodes and clusters by using sections directive.
2. Main computational improvement is through parallelizing the Euclidean distance computation in each iteration. Since the amount for computation is equal for each thread, static scheduling used.
3. Atomic directive for updating coordinates in iterations.