K-Means algorithm

Implementation of sequential and parallel version and speedup analysis

Alessandro Sestini - 6226094

15-01-2018

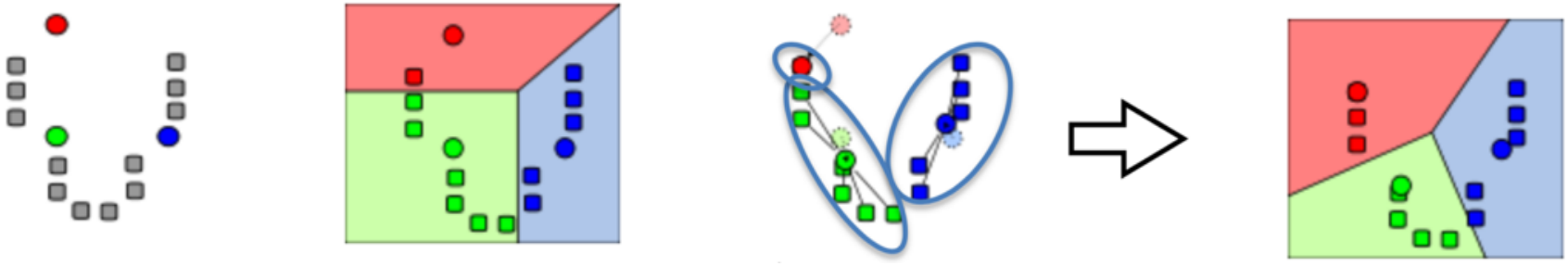
Introduction

K-Means algortihm

The task of k-means is to divide points within a space in **K groups** based on their characteristics.

It is based on few step:

1. Generate N points;
2. Generate K centroids that represent K clusters;
3. For each point, compute the distance between the point and all of the clusters and assign the point to the nearest cluster;
4. Update the centroid's characteristics, specially its coordinates, based on the new points inside the cluster;
5. Repeat from 3 until reaching a maximum number of iterations or until the clusters won't move;



K-means example

Parameters

- The number K of clusters must be **known**;
- We must define **space and centroids**;
- We must define **distance** (related to the space);
- We must define the **initial positions of clusters**;
- We must define the **maximum number of iterations**;

The algorithm **converges very quickly,** but it does not guarantee to find the **best solution**. If the clusters are chosen randomly, the solution might be different repeating the algorithm.

Implementation

Sequential version

It was chosen C++ for the sequential implementation and **OpenMP** for the parallel implementation.

2 classes are defined:

- **Point.h**
  - 2D points chosen randomly;
  - x_coord, y_coord, cluster_id;
  - setter() and getter() methods;
- **Cluster.h**
  - 2D points chosen randomly at first;
  - x_coord, y_coord, size, new_x_coord, new_y_coord;
  - setter() and getter() methods and others to update the coordinates of the centroids.

main_sequential

- In this program, points and clusters are 2D points **chosen randomly**;
- The first functions **initialize** clusters and points;
- After that, there is a while loop where are the **2 most important functions**:
  - compute_distance();
  - update_clusters();

compute_distance()

```
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]);
    }
}
```

update_clusters()

```
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;
}
```

Parallel version

- The program is mostly parallelizable in the **compute_distance()** function;
- We can parallelize the **outer for**;
- **OpenMP** was choosen in order to parallelize the for;
- Here's a table where we can see the time spent by the algorithm inside each funtion;

Function	*Seconds*
compute_distance(…)	1.075 s
update_clusters(…)	0.000001 s

Parallel compute_distance()

- It has been applied a parallel for to the **outer loop**;
- **Private variables** for each thread: min_distance and min_index;
- **Firstprivate variables** for each thread: points_size and clusters_size;
- **Shared variable**: vector of points and vector of clusters;
- Since the amount for computation is equal for each thread, it was chosen a **static scheduling**;

Parallel compute_distance()

```
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;
                }

            }
            points[i].set_cluster_id(min_index);
#pragma omp critical
            clusters[min_index].add_point(points[i]);
        }
    }
}
```

Parallel compute_distance()

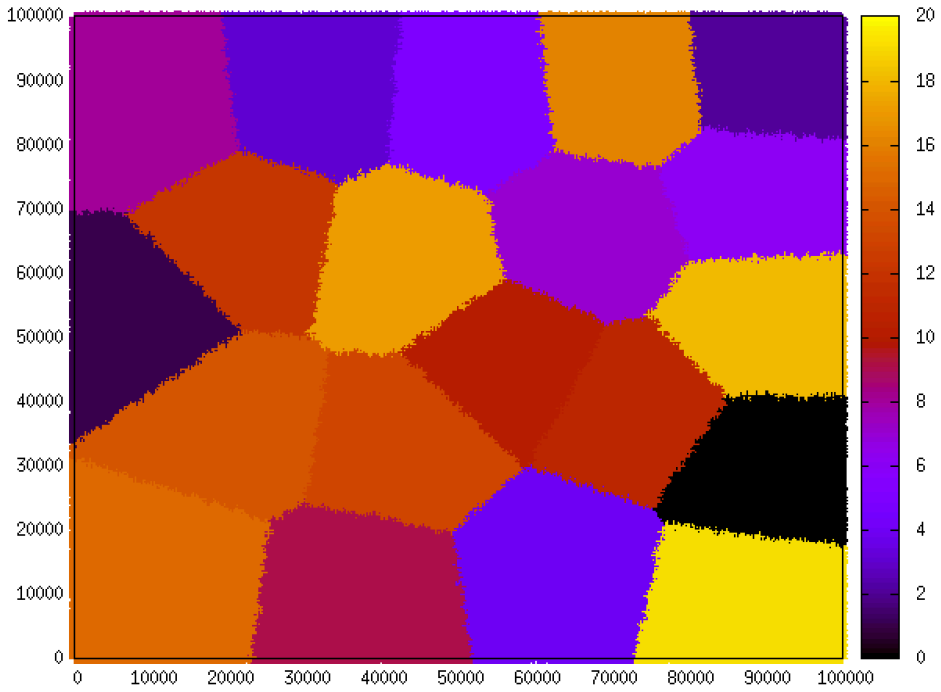
```
void compute_distance(vector<Point> &points, vector<Cluster> &clusters){

    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;
                }

            }
            points[i].set_cluster_id(min_index);
#pragma omp critical
            clusters[min_index].add_point(points[i]);
        }
    }
}
```

#pragma omp atomic!

Experiments and results

- The experiments were made by **varying the number of clusters and points**;
- It has been compared the **completion time** of the sequential algorithm versus the parallel version;
- The number of iterations was kept fixed at 20;
- The experiments were made on a **2 core** machine;
- As we can see from the table below, the general **speedup reached is 2**;

Points	Clusters	Total sequantial time	Iteration sequential time	Total parallel time	Iteration parallel time
100000	10	3.57 s	0.18 s	2.74 s	0.13 s
100000	20	5.68 s	0.28 s	4.04 s	0.20 s
500000	10	12.75 s	0.64 s	6.40 s	0.37 s
500000	20	22.40 s	1.12 s	12.13 s	0.61 s
1000000	10	22.74 s	1.13 s	12.13 s	0.61 s
1000000	20	40.66 s	2.03 s	21.40 s	1.20 s



Result example with gnuplot