K-Means Algorithm: A comparison between sequential and parallel version

Lorenzo Gianassi
lorenzo.gianassi@tud.unifi.it

Abstract

This mid-term paper focuses on the study and implementation of the k-means algorithm both in sequential and parallel version. The implementation was done in C ++ and OpenMP for parallel part. At the end some analyzes and comparisons were performed to obtain the performances of the two versions on a machine with a 2-core Intel i7 processor. To evaluate the performance increase from the sequential to the parallel version speedup was measured : the results say using the parallel version we got one average speedup of 2, therefore a linear speedup, which is what we would expect with a 2 core machine.

1. Introduction

We introduce the fundamental concepts of the project by addressing the algorithm and the choice of values used.

1.1. Algorithm

K-means is a simple unsupervised learning algorithm that is used to solve clustering problems. It follows a simple procedure of classifying a given number of **points** into a number of **clusters**, defined by the letter *k* which is known and chosen a priori. The clusters are then positioned as points and all observations or data points are associated with the nearest cluster.After the first computation, the process starts over using the new adjustments until a desired result is reached.
By abstraction, points are represented in a 2D space and the coordinates (*x,y*) of each one are randomly generated and each cluster is represented by his centroid (with coordinates (x,y)). The algorithm is very simple and basically consists of a few steps:

1. N points are generated in space with random coordinates;

2. K centroids are generated which represent the K clusters. In the initialization step, the cluster will contain only its centroid;

3. For each point, the distance with all clusters is calculated, then, the point is assigned to the nearest cluster;

4. The characteristics of the centroid are updated, in particular its position within the cluster;

5. It is repeated from step 3 until the centroids remain stationary or until other stop criteria occur.

It is an algorithm that converges very quickly, although it is not guarantee to find the optimum overall. The quality of the final result status depends mainly on the number and position of initial clusters.

1.2. Values

The algorithm needs to specify some parameters and we will show the assumptions made for these. The number K of the clusters must be known and chosen a priori as we said in the precedent paragraph; alternatively it is possible to adopt more complex solutions to choose the best *k* given the points. We decide to choose multiple values of *k* and show the performances obtained. About the **clusters** we need to to talk about the *centroids*. The centroid can be a point of the space with coordinates defined by the average of the points belonging to that cluster. Furthermore, it is necessary to choose the initial centroids: there are various techniques, the one used in this program is to take K random points of space. Having randomly initialized the clusters, it is not said that by repeating the algorithm twice on the same points, the result will be the same.
About the method to calculate the distance between the points and the clusters: we have decided to apply the **euclidean distance**, since we re considering point in a 2D space.
Finally, we have decided to include another stop criteria, that is a maximum number of iterations after which the process will stop.

*We can see a simple representation of the **K-means** algorithm in the following Figure*

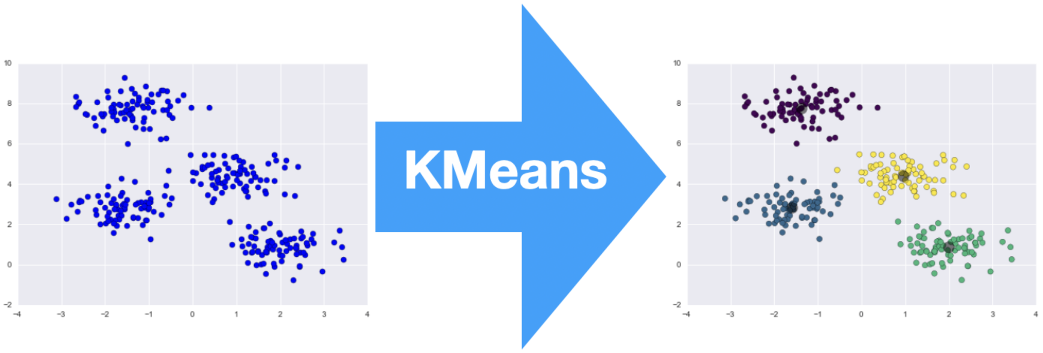

Figure 1: Example of **K-Means** Algorithm

2. Implementation

Let's now see the implementation of the algorithm: we will first analyze the sequential version and then we will move on to parallel version. As a programming language was used C ++ and then extended with OpenMP to parallelize some sections of the code.

2.1. Classes

We introduce the simplest components of the program which are the classes that represent the points and the clusters we need for the implementation of the Algorithm.

1. **Cluster**: Cluster-type objects are composite with the parameters *coord_x* and *coord_y* (*they are considered as points*), plus an int attribute *N_points* indicating the number of points belonging to that cluster, necessary to update the characteristics of the centroid. The methods in this class are the *getter()* and *setter()* functions of the attributes, plus *add_point()*. The Methods used for step 4 to update the coordinates of the centroid and verify if that particular point is moved or not compared to the previous position, are *delete_values()* and *update_values()*. Furthermore, to do so, we have used *tot_coord_x* and *tot_coord_y* as attributes that represents the sum of the coord_x and coord_y of the points belonging to the cluster. This will help us to update the coordinates of the centroids at each step as average of the points of each cluster.

2. **Point**: Point-type objects are composed by 3 attributes. They are double *coord_x* and *coord_y*, that represent the value of the x and y coordinates of the point in space, and int *id_cluster*, that stands for the id of the cluster to which that point belongs. In the initialization step, all points will be assigned to cluster 0. Finally, the functions in this class are simply the getter() and setter() methods of the attributes as saw in the *cluster* class.

2.2. Sequential and Parallel Main

We now show the core of the project that implements and reproduces the algorithm. We have two *Main* that represent sequential and parallel execution. The first represents the classical execution of the algorithm while the second represents the parallelized execution through the use of the **OpenMP** API.
Let's start with the ***sequential implementation***. Let's leave out the trivial parts of initialization of points and clusters and focus on the two fundamental functions of the program: *find_distance()* and *update_clusters()*. These functions contain most of the complexity of the algorithm. The points and the clusters are saved inside a vector object of the *stl* library to be able to handle vectors of points and clusters, this allows us to iterate through the various points and clusters. In *find_distance()* we iterate on the vector of points and for each we calculate the distance between the coordinates of the point and the ones of cetroids of each cluster, so we need to do a double for loop cycle (iterate on the points and on the clusters). In the end we assign to the point the cluster that resulted more close so we need, also, to call the method *update_clusters()* to update the coordinates of each centroid. We have also introduced another stop criterion in the method. a Boolean value that will be true if even only one cluster has changed its centroid coordinates.
These two methods have for loops but substantially differentiate the number of elements they iterate on. In *find_distance()* we iterate on all points we have in the space which can be also one million, instead in *update_clusters()* we iterate on clusters which can be at most a few thousands.
We can move on to talk about the **parallel version**, that differs from the sequential for few steps. The sequential version of the problem is very parallelizable in the first *find_distance()* function. This because it is both the center of the algorithm, and because the operations that are performed for each point are independent from each others.

It is therefore possible to take any point, calculate the distances between it and all the clusters and assign them the most close in a parallel way; in practice, just parallelize the outer for loop. The for lop in the *update_clusters()* method is done on few elements (the clusters) so it would be irrelevant to parallelize it, instead the for loop in *find_distance()*, if parallelized, it will produce the greatest increase in speedup. We will show this result in the paragraph dedicated to tests. The problem is therefore enclosed in a for loop, and therefore the most intuitive solution is to use **OpenMP** which, for its features, it is great for parallelizing this type of loop.
A ***pragma parallel for*** has been applied to the outer loop. As regard the variable, we set as **private** variables *min_dist* and *min_index*, which will be initialized for each point. Furthermore, the *pts_size* variables have been set as **first-private** so that each thread has its own private copy of the variables initialized to the value given by the main thread before entering the parallel region. The array of points and clusters are set to **shared**.
Since multiple threads can access the same cluster, the section in which we add the point to the cluster must be defined as critical, while the operations performed on the points are totally independent of each other and therefore no synchronization mechanism is required. Finally, as far as **scheduling** is concerned, we decided to set to static. This is because the amount of computation of each thread had to be equal.

We now show the most important sections of code that concern the parallelization step.

```
bool update_clusters(vector<Cluster>&cls){
    bool  iterate = false;
    for (int i = 0; i < cls.size(); ++i){
        iterate = cls[i].update_values();
        cls[i].delete_values();
    }
    return iterate;
}
```

This is *update_clusters()*, the method that takes care of updating the cluster values.

```
void find_distance(vector<Point>&pts,vector<Cluster>&cls){
    unsigned long pts_size = pts.size();
    unsigned long cls_size = cls.size();

    double min_dist;
    int min_index;
#pragma omp parallel private(min_dist,min_index)
#firstprivate(pts_size, cls_size)
#shared(pts,cls)
    {
#pragma omp for schedule(static,1000)
        for (int i = 0; i <pts_size ; ++i) {
            Point &currrent_point = pts[i];
            min_dist = euclidean_dist(currrent_point, cls[0]);
            min_index = 0;
            for (int j = 0; j < cls_size; ++j) {
               Cluster &current_cluster = cls[j];
               double dist =
               euclidean_dist(currrent_point, current_cluster);
// check the value of the distance if is inferior, update
            if (dist < min_dist) {
                    min_dist = dist;
                    min_index = j;
                }
            }
// set the cluster_id of the point with minor distance
            pts[i].set_id(min_index);
            // add th point to the found cluster
#pragma omp critical
            cls[min_index].add_point(pts[i]);
        }
    }

}
```

This is *find_distance()*, the method that deals with calculating the value of the Euclidean distance for each point, by iterating over the vector of the points and clusters.

3. Experiments and Results

In this section we are going to show the experiments made and the results obtained from them. In the various experiments we will modify the parameters to see how they affect program execution times. The goal of these experiments will be to obtain a considerable speed up in execution times

3.1. Experiments

The experiments we did have the main goal of showing the speedup achieved by parallelization using OpenMP. Furthermore, by varying the parameters it is possible to evaluate the changes in the program execution times.

Sequential Algorithm			
Points	Clusters	Total Time	Iteration Time
100k	10	2.21 sec	0.110 sec
100k	20	2.81 sec	0.140 sec
250k	10	3.96 sec	0.195 sec
250k	20	6.75 sec	0.337 sec
500k	10	7.25 sec	0.362 sec
500k	20	13.37 sec	0.668 sec
1mln	10	14.50 sec	0.725 sec
1mln	20	28.29 sec	1.414 sec

Table 1: Result of the Sequential Computation

Parallel Algorithm			
Points	Clusters	Total Time	Iteration Time
100k	10	1.58 sec	0.079 sec
100k	20	1.79 sec	0.089 sec
250k	10	2.14 sec	0.107 sec
250k	20	3.81 sec	0.190 sec
500k	10	4.20 sec	0.212 sec
500k	20	6.78 sec	0.339 sec
1mln	10	8.45 sec	0.423 sec
1mln	20	13.26 sec	0.663 sec

Table 2: Result of the Parallel Computation

We therefore decided to show the execution times as the number of **points** belonging to the space increases, starting from 100k points up to 1 million. As regards the **clusters** we have decided to show the results for 10 and 20 clusters. To complete the analysis, the number of clusters were increased by keeping the number of points fixed (1 million), subsequently analyzed a possible speedup on the method that updates the cluster centroids (*update_clusters()*) and finally the number of *threads* was changed choosing a fixed number of points and clusters. The values obtained in the tests are an average of the tests performed in the various combinations of parameters to obtain more stable results.

3.2. Results

Let us now look at the results obtained from all the experiments for both the sequential and the parallel solution. We show both the total execution times of the program and the average time to execute one iteration of the algorithm. The number of Iterations is set to 20 but can obviously be changed. As far as the **sequential** solution is concerned, we can notice that as the number of points increases, the execution times obviously increase as well the as the times for each iteration. Although the execution times of the algorithm increase, the possibility of parallelizing the algorithm also increases. So we can better appreciate the effects of parallelization with a high number of points as in the case of one million. Instead, using a lower number of points, the gap between the sequential version and the parallelized version is less visible and appreciable. For a few points it is not visible as we have to consider the time of creation of CPU threads, so the time gain given by the parallelization is partially nullified. This statement is shown by the first lines of the two tables above (*Table 1 and Table 2*) where the gap between the two processes is smaller. Indeed, in the following lines, the **speedup** achieved has a constant trend as we have a completion time halved compared to the sequential version. This is due to the fact that the program is tested on a two-core computer. To better appreciate the previous results, tests were carried out by increasing the number of clusters, keeping the number of points stable at 1 million. The number of clusters used follows this trend: 100 - 250 - 500 - 1000 - 5000. The values shown in the *table 3* confirm the previous results. Obviously the execution times increase but, particular, we can notice that the speed up achieved remains approximately equal to 2 for all executions. Having considerably increased the number of clusters, it might be interesting to parallelize the method that deals with updating the values associated with each of them, **update_clusters()**. To do this, the for loop that runs through all the clusters, updating the respective centroid, has been parallelized. Furthermore, the variable **iterate** has been set to **lastprivate**, which represents a *boolean* value that is used as a stop criterion in case the coordinates of the centorids no longer vary during the predifined number of iteration, and the other variable set to *shared*. The implementation of the sequential method can be seen on the previous page. But the order of magnitude of the execution time of this method is very small, around 0.0001. This means that the parallelization performed on this method does not lead to a real speedup in performance. Furthermore, as previously mentioned if the number of points on which we execute the parallelized for loop is small, the time of creation of CPU threads leads to a loss of the time gain given by parallelization. To finish the study of the experiments we are going to work on the number of threads used in the for loop of the *find_distance()* method. Since the machine that was used has a 2-core processor that works on 4 threads, tests were performed by varying the number of the latter.

Increasing Number of Clusters		
Clusters	Sequential	Parallel
100	119.089 sec	64.684 sec
250	321.60 sec	163.245 sec
500	627.510 sec	323.199 sec
1000	1154.647 sec	639.892 sec
5000	5778.009 sec	3118.259 sec

Table 3: Results obtained with increasing number of **Clusters** and fixed number of Points.

The range of number of threads used corresponds to: 2-3-4-5-6-8. Previous tests have been performed with the default number of threads being 4. In *Figure 2* it is reported the trend of the speedup value, represented as the ratio between sequential time and parallel time. In this case the experiments were performed on 1 million points and 500 clusters as the number of threads increases. This combination of parameters was the best compromise between execution times and enough number of clusters to observe the speedup. The values obtained attest that using less than 4 threads leads to a lower speedup value. While using a value higher than 4 you can see that the speedup stabilizes around the value of 2 (slightly less). So the value that leads to better speedup results is 4 threads.

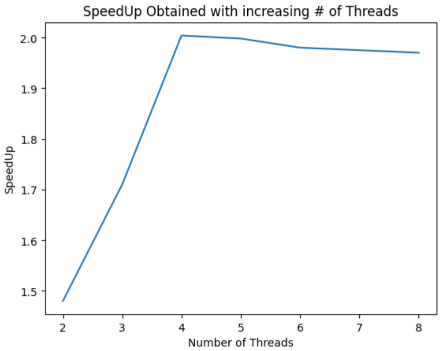

Figure 2: Speedup obtained running the process on 1 million points and 500 clusters as the number of threads increases.

3.3. Graphic Plot

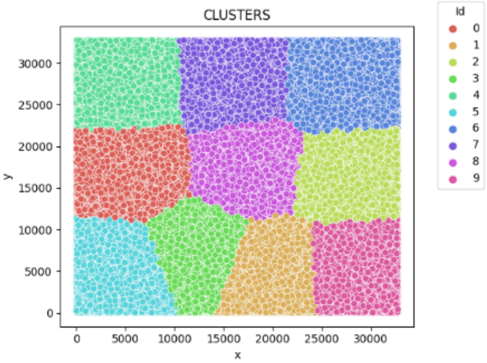

Figure 3: Plot of the K-Means Algorithm with 10 clusters

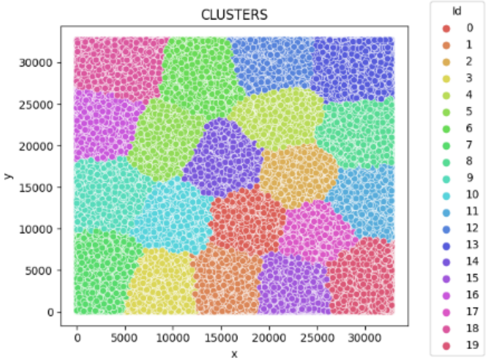

Figure 4: Plot of the K-Means Algorithm with 20 clusters

Plots of graphs were also performed which represent the results obtained by the algorithm with the use of 10 and 20 clusters respectively. We used a Python script external to the program. The *matplotlib* and *seaborn* library was used to draw the graph together with *pandas* to manage the csv file generated by the plot method implemented in the main of the program. In Figure 4 we can see graphically the results of the K-Means algorithm for 10 clusters and 20 clusters.