K-Means Clustering: comparison between a sequential and parallel implementation

Beatrice Paoli
beatrice.paoli@stud.unifi.it

Abstract

The goal of this paper is to present two implementations of the k-means clustering algorithm: a sequential version and a parallel one. Both are written in C++, with the parallel version obtained with the use of OpenMP. This paper also provides a performance analysis and comparison between the two versions in terms of execution times and speedup and how this values change with different inputs (dataset size and number of centers) and different numbers of threads.

1. The Algorithm

The K-means algorithm is a parametrized clustering technique that allows to partition a dataset of points or observations into K clusters. Points in the same cluster minimize the distance from the center of the cluster (called *prototype*).

Many metrics of distances can be used depending on the dataset to cluster, for this implementation the dataset is comprised of 2D points and distances are measured with the Euclidean distance.

The algorithm is composed by few steps, with two main phases alternated between each other: the *Assignment* phase and the *Update* phase.

Algorithm 1 K-Means Clustering

Require: K = number of clusters to create

```
Select K points as initial centroids of the clusters
while Centroids keep changing do
    for each point p do
        Assign p to the cluster with the closest centroid
    end for
    for each cluster c do
        Update the centroid of c
    end for
end while
return clusters
```

The algorithm converges after few steps to a local minimum, depending on the choice of the initial centroids. In the implementation presented in this paper, the starting centroids are chosen randomly from the dataset in input. Therefor, different runs of the algorithm on the same dataset and the same number of centroids can yield different results. Other initialization methods can be used to obtain the gloabl optimum more consistently.

2. Implementation

The details of the implementations of the function *kMeansClustering()* are presented in the following paragraphs.

2.1. Classes

- **Point**: this class is used to represent the points of the input dataset to cluster. It has three members: x and y for the coordinates, and *clusterId* for the id of the cluster to which the point has been assigned.

  The class also has two constructors and the method *dist()* to compute the Euclidean distance between two points.

- **Cluster**: this class is used to represent the clusters created by the algorithm. Each cluster has an *id* ranging from 0 to $K - 1$, a *Point* for the current mean or centroid of the cluster, two fields to compute the partial sums of all points for each coordinate, *tempSumX* and *tempSumY*, and an integer for the size of the cluster.

  Aside from the constructor, the class has two main methods:

  - *addPoint()*: is used during the Assignment phase; it adds a point to the cluster by increasing the *size* counter and by adding its coordinates to the partial sums.
  - *updateCentroid()*: is used during the Update phase; it computes the new mean of the cluster by using the sums of the coordinates computed

before and the size of the cluster. After the update, *tempSumX*, *tempSumY* and *size* are resetted to 0, ready for a new iteration of the algorithm.

2.2. kMeansClustering

The function *kMeansClustering()* accepts three parameters: an integer k for the number of clusters to output, a *std::vector* of type *Point* for the dataset, and an integer for the number of iterations of the algorithm to perform. The last parameter is optional and has a default value of 20.

The function starts by performing a random initialization of the clusters by choosing k points as initial centroids.

In a *for* cycle bounded by the number of iterations to perform, the function alternates the assignment and update steps. In the assignment step, for each point we find the cluster with the closest centroid and then update the point's *clusterId* and call the cluster's method *addPoint()* to update the partial sums of the future centroid.

The update step is limited to a simple *for* cycle over the clusters to call *updateCetroid()* to update all the centroids with the current points assignment.

2.3. OpenMP Parallelization

To achieve parallelization it was necessary to use only a few parallel directives.

The assignment step is embarrassingly parallel since the search and assignment of each point to a cluster is independent from one another. So a `#pragma omp for` statement was used on the points loop to use threads to parallelize the computation.

Algorithm 2 Parallel K-Means Clustering

Require: K = number of clusters to create

```
Select K points as initial centroids of the clusters
while Centroids keep changing do
    #pragma omp for
    for each point p do
        Assign p to the cluster with the closest centroid
    end for
    for each cluster c do
        Update the centroid of c
    end for
end while
return clusters
```

Since the workload for each thread is similar the default static scheduler is used.

However, each thread accesses the shared clusters to call *addPoint()* and update the internal cluster variable. Without any synchronization mechanism, this introduces a race condition on the algorithm that leads to incorrect results.

The method *addPoint()* only performs three sums over the variables *tempSumX*, *tempSumY* and *size*, so three `#pragma omp atomic` were used to ensure the correct update of the variables and avoid the race condition.

The update step, as stated before, is limited to a simple *for* loop over the clusters to update each centroid by performing only two divisions. This step is also embarrassingly parallel but the computations performed inside the loop are far too simple and the number of clusters is much smaller compared to the number of points. For these reasons it would be counter-productive to paralleliza this step, as the overhead of thread management would decrease the performance of the algorithm.

3. Performance Analysis

In order to compare the performance of a sequential algorithm with its parallel version we can use the concept of *speedup*.

Speedup is measured as the ratio between the execution time of the sequential algorithm and the parallel one.

$$S = \frac{t_s}{t_p}$$

Ideally, we should look for *perfect speedup* or even *linear speedup*, meaning S should be equal or similar to the number of processors used to perform the parallel algorithm.

The experiments were performed on a Intel Core i7-1165G7 with 8 logical cores, with 2D input datasets generated with *scikit-learn* in Python.

Since the algorithm has the hyperparameter K for the number of clusters, different experiments were performed with varying K and fixed N (number of points), and viceversa.

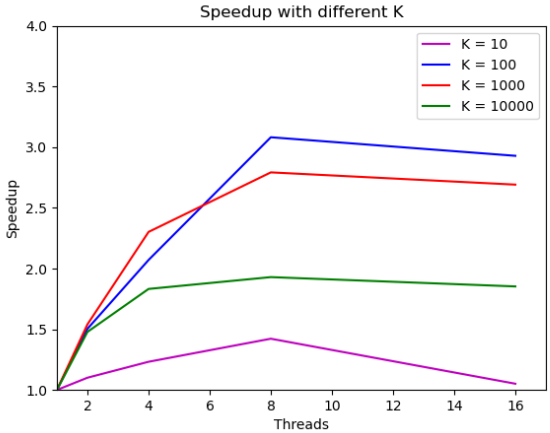

Figure 1. Speedup K Means Clustering con N = 100000.

Threads	Execution Time	Speedup
1	12.106	1
2	7.860	1.540
4	5.259	2.302
8	4.335	2.792
16	4.498	2.691

Table 1. Execution Times for K = 1000 and N = 100000.

In Figure 1 we have the different speedups for different numbers of threads used and different values of K. We can see how the speedup increases up to 8 threads as expected, but in a sub-linear way. The results also depend on the value of K chosen.

For small K, the speedup are significantly worse and deteriorate almost to 1 for a high number of threads. Since the work of each thread consists mainly on computing all the distances between a point and all centroids to find the closest cluster, so for small numbers of K the computation of each thread can become too little and the gain in speedup is lost due to thread management.

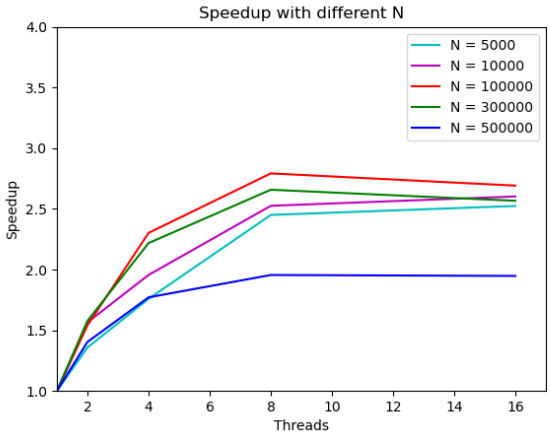

Figure 2. Speedup K Means Clustering con K = 1000.

In Figure 2 we have the speedup trends for different sizes N of datasets. As before, the values increase up to 8 in a sub-linear way.