Comparative analysis between sequencial and parallel version of the K-means algorithm in C++

Francesco Bongini
bongini.francesco@gmail.com

Abstract

For physical limitations, in the recent decades computers have increased the number of cores rather than increasing the speed of each one. To get the best beneficts of the computational power, we need to build new versions of existing algorithms. This paper presents the results of a comperative analysis between sequencial and parallel version of the K-means clustering. The parallel code is implemented using OpenMP in C++. My results indicate that the parallel implementation has benefits regarding its speed and the amount of data processed in the same quantity of time.

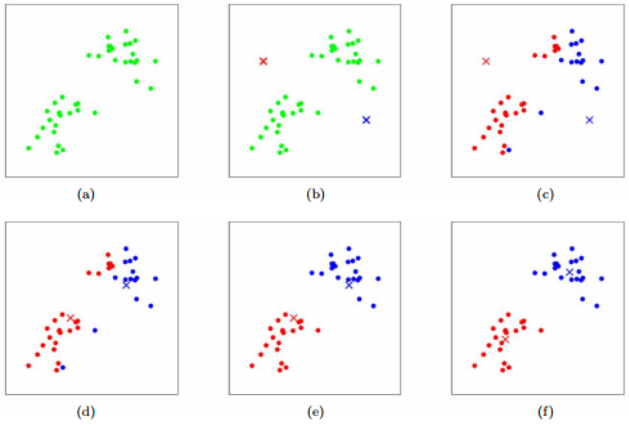

Figure 1. K-means algorithm in different iterations

1. Introduction

K-means is a simple unsupervised learning algorithm used to solve clustering problems. Definited the parameter *K* i.e. the numer of clusters, the algorithm chooses randomly *K* points (called centroids). Each point of the dataset is assigned to the cluster with the closest centroid. After all points are assigned, the positions of the *k* centroids are recalculated with the mean of the cluster. The algorithm repeates the recalculates until the centroids don't change anymore. Here the pseudocode of the K-means clustering:

Algorithm 1 K-means

1: Select K points as the initial centroids.
2: repeat
3: Form K clusters by assigning all points to the closest centroids.
4: Recompute the centroid of each cluster with the mean of the points in each cluster.
5: Until the centroids don't change.

Given two points p and q in n dimensions, their euclidean distance is defined as:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

2. Plotting clusters

After have calculated the clusters with K-means algorithm, we plot them. Python provides interesting libraries to plot our clusters in 3D.
The dataset is a mixture of 3 dimensions points from Gaussian distributions with mean 0, 3 and 6 and standard deviation equal to 0.2 for each one. Given the standard deviation very small, the algorithm converges in barely 4 iterations. Here the plots in 2 and 3 dimensions:

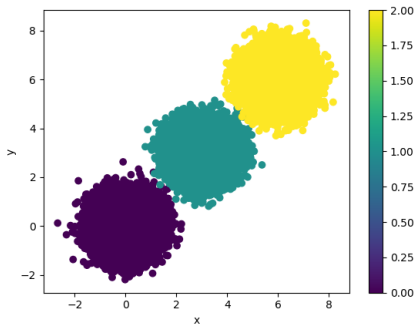

Figure 2. Plot clusters in 2 dimensions

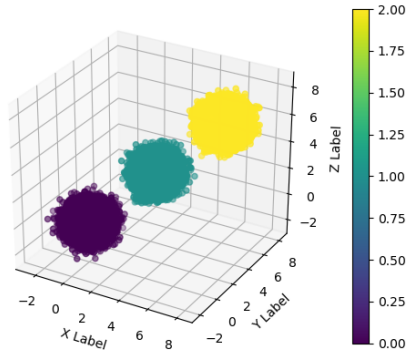

Figure 3. Plot clusters in 3 dimensions

3. Sequential version

Sequential version of the K-means algorithm consists to calculate and update the new centroids until they differ from the old centroids less than the ϵ value. Here is the pseudocode of the sequential version:

Algorithm 2 Sequential version

```
while centroids change do
    for each point do
        point.setCluster()
    end for
    upgradeCentroids();
end while
```

The choice of the ϵ value may affect the speed and the precision of the algorithm. The points are definited as PointND object. It consists in vector of floats i.e. the coordinates and the attribute Cluster. Cluster is an integer-type that indicates the class to which each point corresponds. During each iteration of the K-means the points don't change, but may occur to the Cluster. An other important data structure is the vector of PointND, used to define the list of centroids. Each point of the dataset is visited sequentially to calculate the distance from the K centroids.

4. Parallel version

OpenMP is a standard Application Programming Interface of C, C++ and FORTRAN for a shared memory parallel programming. It allows to parallel the exacution of K-means in a very simple way. In fact, after definited the number of $threads$, the command *#pragma omp parallel for* allows to each $thread$ to compute an indipendent part of the data. This is possible because each iteration is indipendent from the other one. Given that the steps are the same for each data point, static schedule is assigned by default. Here is the pseudocode of the parallel version:

Algorithm 3 Parallel version

```
while centroids change do
    #pragma omp parallel for
    for each point do
        point.setCluster()
    end for
    upgradeCentroids();
end while
```

Note that "upgradeCentroids()" isn't parallelized. If we parallelize that with *Pragma OpenMP*, different data points might be added to the sum of them to calculate the centroid at the same time, leading to Write-After-Write (WAW) hazard. It happens when different threads save their results to the same variable causing the loss of results of the previous threads.

5. Speedup

Speedup is an index that measures the relative performance of two systems processing the same problem. It is defined as:

$$S_p = t_s/t_p$$

where P is the number of processors, t_s is the completion time of the sequential algorithm and t_p is the completion time of the parallel algorithm. Here is some Speedup types:

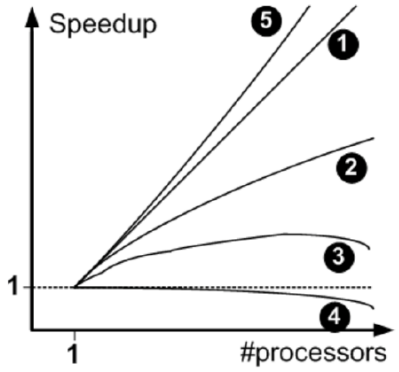

Figure 4. Speedup types

Types:
1. Linear speedup
2. Sub-linear speedup
3. Speedup with an optimal number of processors
4. No speedup
5. Super-linear Speedup

In order to analyze the speedup for this problem, different datasets are considerated. The first example shows the results from the hardware of my laptop. In the other examples I used an istance from Amazon AWS.

5.1. 500k points

Dataset: 500k points in 3 dimensions from Gaussian distribution with mean 0 and standard deviation 1.
Parameters: ϵ=0.001 and K=3.
Processor: Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz, 2401 Mhz, 2 Core(s), 4 Logical Processor(s).
number iterations: 121
Results
1 thread: 186.833 seconds
2 threads: 131.318 seconds
3 threads: 128.658 seconds
4 threads: 109.824 seconds
5 threads: 113.157 seconds
6 threads: 111.582 seconds
7 threads: 112.294 seconds
Increasing the number of threads the overhead increases, so we need to find the optimal number of threads. In particular, 4 threads is the optimal choice.

5.2. 2M points

Dataset: 2 millions points in 4 dimensions from Gaussian distributions with mean 1 and standard deviation 1.
Parameters: ϵ=0.001 and K=3.
Processor: t2.2xlarge Linux/UNIX Amazon Ec2 Spot Instance with 8 vCPUs.
number iterations: 229
Results:
1 thread: 929.932 seconds
2 threads: 554.635 seconds
3 threads: 434.368 seconds
4 threads: 375.278 seconds
5 threads: 340.358 seconds
6 threads: 317.651 seconds
7 threads: 301.631 seconds
8 threads: 295.699 seconds
9 threads: 317.819 seconds
10 threads: 307.321 seconds

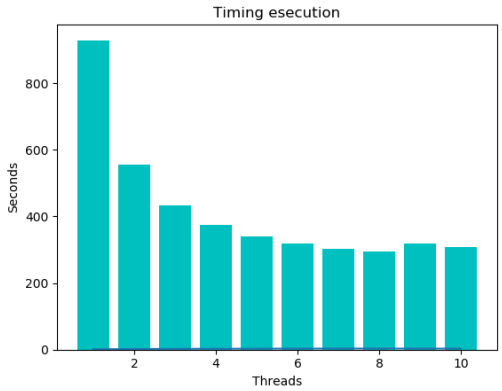

Figure 5. Bar chart

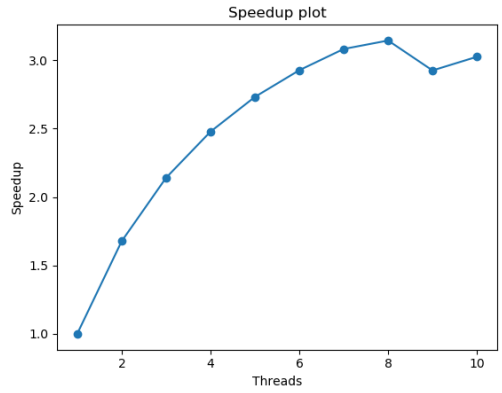

Figure 6. Speedup OpenMP

6. Further parallelization

A further parallelization of the problem is possible. In fact, after calculating the distance between points and centroids, it's possible to parallelize the external loop of the calculation of the new centroids. In this way, each thread calculate the new centroid of a different cluster. For istance, thread 0 computes the cluster 0, thread 1 computes the cluster 1 etc.. . It is important therefore that K is big enough,

to use all cores availables. Given that the steps are the same for each cluster. There are slight differences between static and dynamic schedule. Static schedule is assigned by default.
Here is the pseudocode of the new parallel version:

Algorithm 4 Parallel version

```
while centroids change do
    #pragma omp parallel for
    for each point do
        point.setCluster()
    end for
    #pragma omp parallel for
    for each K do
        upgradeCentroid();
    end for
end while
```

Now we consider the same example of the subsection 5.2. with the further parallelization. Even though all cores are not used given that $K = 3$, we expect an improvement of the performances. Same istance of amazon EC2 is used with 8 vCPUs.

6.1. 2M points with further parallelization

Dataset: 2 millions points in 4 dimensions from Gaussian distributions with mean 1 and standard deviation 1.
Parameters: ϵ=0.001 and K=3.
Processor: t2.2xlarge Linux/UNIX Amazon Ec2 Spot Instance with 8 vCPUs.
number iterations: 229
Results:
1 thread: 929.932 seconds
2 threads: 485.726 seconds
3 threads: 314.031 seconds
4 threads: 254.987 seconds
5 threads: 219.757 seconds
6 threads: 196.695 seconds
7 threads: 180.139 seconds
8 threads: 166.975 seconds
9 threads: 197.157 seconds
10 threads: 192.232 seconds

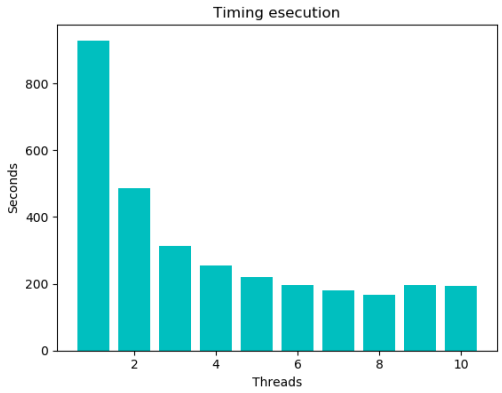

Figure 7. Bar chart

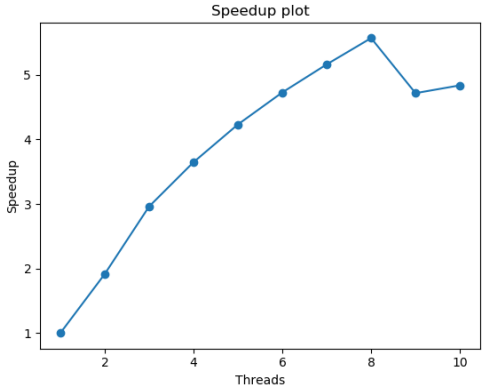

Figure 8. Speedup OpenMP

We can note a big improvement. With 8 threads the new speedup is 5.57 against the old one with 3.14. In this case the speedup trend is linear until thread 8, in fact with 9 and 10 threads it leads in a overhead.

6.2. K-means performances changing K

The biggest disadvantage of the second parallelization is that the number of threads used depends on the value of K. As we can see in the example below, it's also hard to evaluate the effective speedup performance when we change the value of K.
The used dataset refers to 200k points from normal distribution with mean 0 and standard deviation 1. In this example K changes but ϵ is fixed to 0.01. In the first parallelization we set 8 threads i.e. the number of vCPUs on the t2.2xlarge istance, while in the second the number of threads is exactly K.

K	Sequential time	Parallel time	Speedup
2	68.62	14.91	4.60
3	29.58	6.49	4.56
4	28.38	5.89	4.81
5	98.03	15.29	6.41
6	135.06	19.70	6.85
7	50.65	8.26	6.13
8	131.08	18.25	7.18
9	130.24	19.88	6.55
10	89.14	13.95	6.39

In fact it may happen that bigger K converge with fewer iterations, leading to the overhead and afecting the speedup performance.

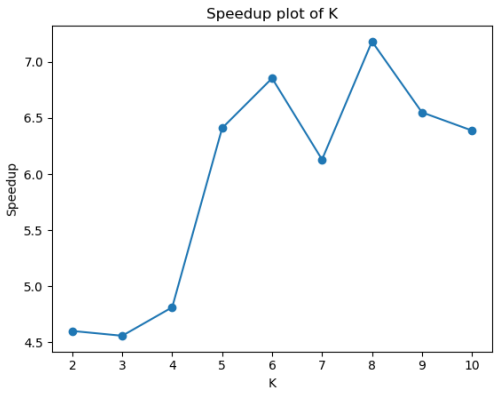

Figure 9. Speedup

In this example, K=3 and K=7 have worst performance than some their lower values.
In order to evaluate correctly the speedup performances, we sightly change the algorithm. We remove the ϵ parameter and we iterate the algorithm a fixed number of times. Here is the pseudocode:

Algorithm 5 Parallel version

```
while count≤200 do
    #pragma omp parallel for
    for each point do
        point.setCluster()
    end for
    #pragma omp parallel for
    for each K do
        upgradeCentroid()
    end for
    count++
end while
```

Here the new speedup analysis with 200 iterations of the same dataset with 200k points:

K	Sequential time	Parallel time	Speedup
2	47.13	11.19	4.21
3	63.63	12.49	5.10
4	77.96	13.89	5.61
5	92.94	15.54	5.98
6	104.94	16.70	6.28
7	120.33	18.49	6.50
8	133.82	19.97	6.70
9	149.23	23.90	6.24
10	161.50	25.44	6.35

and here is the speedup plot:

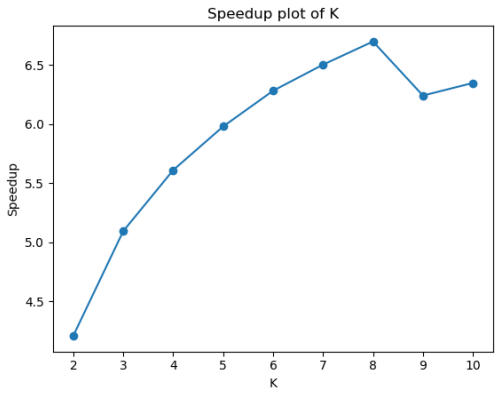

Figure 10. Speedup

6.3. Inefficient parallelization

There are few cases in which the parallelization is not convenient. In these cases, we should use the sequential version. Let's consider a dataset with barely 10 observations. As we can see below, it isn't convenient to parallelize the problem in which it leads to the overhead.

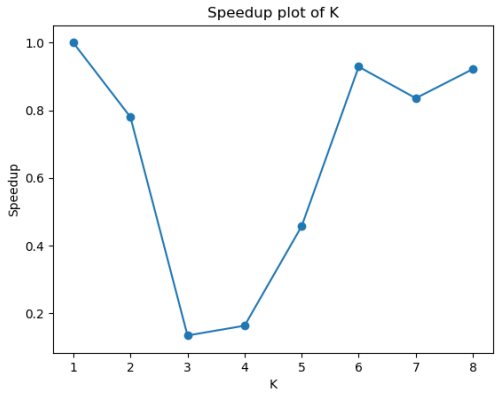

Figure 11. Speedup

The plot shows that there isn't speedup and the sequential version is even more fast than parallel version.

6.4. Conclusions

In this paper we compared sequential and parallel version of the K-means algorithm with OpenMP. We used different example to show the performances. Most of the time the parallel version is preferable rather than the sequential version. In fact, in some cases we reached over 7 of speedup using 8 cores. OpenMP has proven to be a great candidate for the parallelization of this problem. Future developments could affect other types of clustering methods like DB-SCAN and Mean Shift.