Parallelizing the K-means algorithm with OpenMP

Francesco Bongini

September 2019

Introduction

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 benefits of the computational power, we need to build new versions of existing algorithms.

In particular, programs that process big amount of data need to be parallelized. K-means clustering is one of them.

Clustering

Clustering is a Machine Learning technique that consixts to group similar points together.
In theory, data points that are in the same group should have similar features.

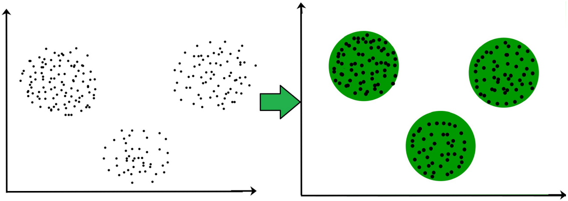

Figure 1: From data points to clusters

K-means

K-means is a simple unsupervised learning algorithm used to solve clustering problems.
The parameter K represents the number of groups in the data.
The algorithm uses the euclidean distance

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

Other distances can be used.

How it works

1. The algorithm chooses randomly K points, called **centroids**.
2. Each data point is assigned to the cluster with the **closest** centroid
3. Recompute the centroids with the mean.
4. Repete untill centroids don't change.

How it works

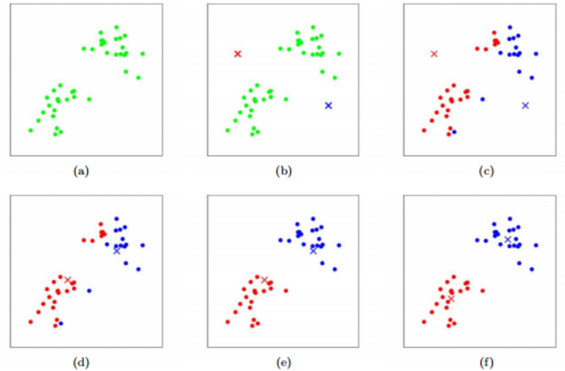

Figure 2: From data points to 2 clusters

Pros and cons

Pros:

- Simple
- Easy to implement
- Time complexity
- Accuracy

Cons:

- No-optimal set of clusters
- Lacks consistency
- Sensitivity to scale

Sequential version

Algorithm 1 Sequential version

```
While centroids change do
    For each point do
        point.setCluster()
    EndFor
    updateCentroids()
EndWhile
```

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.

Parallel version

Algorithm 2 Parallel version

```
While centroids change do
    #pragma omp parallel for
    For each point do
        point.setCluster()
    EndFor
    updateCentroids()
EndWhile
```

OpenMP allows to parallelize the execution 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.

Parallel version

Note that "updateCentroids()" isn't parallelized. That is because 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.

Speed-up

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

Analysis

The speed-up analysis consists to compare sequential and parallel version through speed-up index.

In the first example, I used my laptop dualcore, in the others the t2.2xlarge Amazon Ec2 Instance with 8 vCPUs.

500k data points

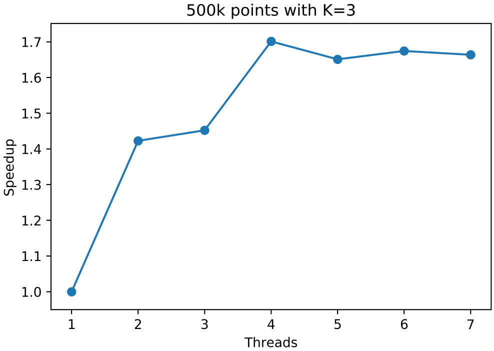

Figure 3: Speed-up using my laptop and setting ϵ=0.001

2M data points

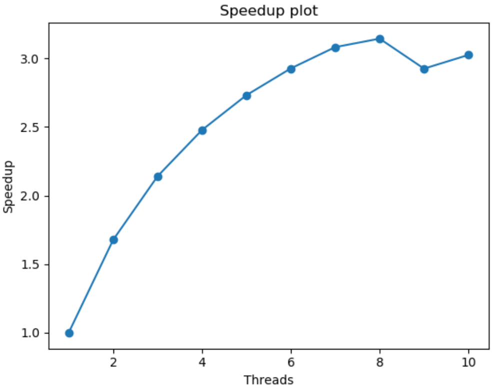

Figure 4: t2.2xlarge Amazon Ec2 Spot Instance with 8 vCPUs

A further parallelization

It's possibile to parallelize the second part of the algorithm, if any new centroid is calculated with a different thread. In this way mutual exclusion is guaranteed.

Algorithm 3 Parallel version

```
While centroids change do
    #pragma omp parallel for
    For each point do
        point.setCluster()
    EndFor
    #pragma omp parallel for
    For each K do
        updateCentroids()
    EndFor
EndWhile
```

A further parallelization

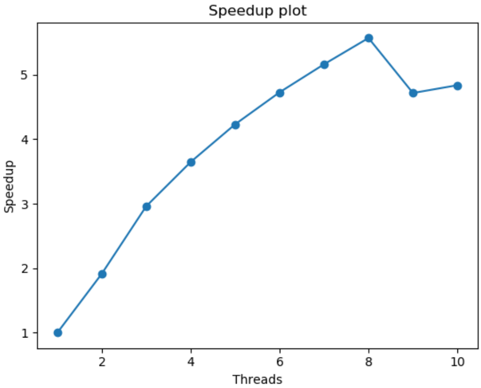

Figure 5: Speedup OpenMP

Same example but parallelizing the second part with K=3.

200k data points with different K

In the first part of this example we set 8 threads, while on the second part the number is equal to K.

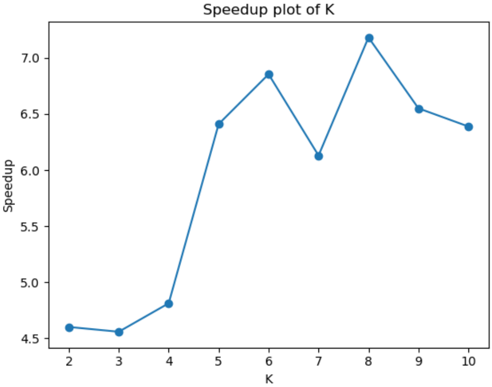

Figure 6: Speed-up with ϵ=0.01

200k data points with different K

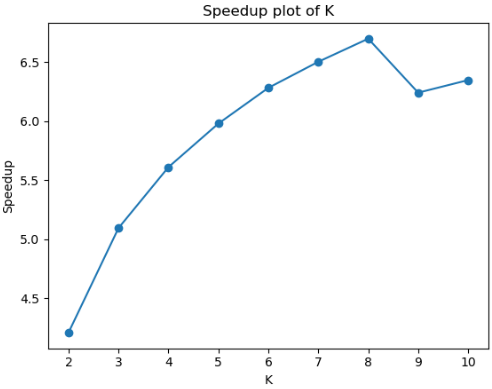

Figure 7: Speed-up with 200 iterations

Inefficient parellization

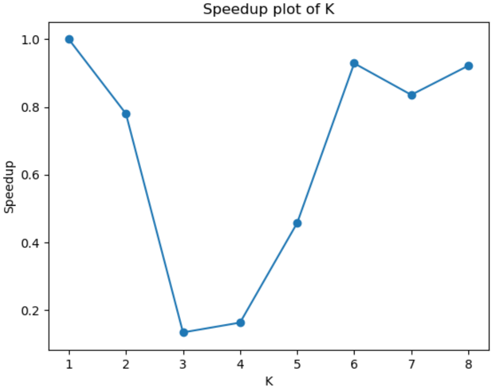

Figure 8: speed-up with 10 data points

Conclusions

OpenMP has proven to be a great candidate for the parallelization of the K-means clustering. In fact, in some cases we reached 7 of speed-up using 8 cores. Future development could affect other clustering methods like DB-SCAN and Mean Shift.