Parallel K-means clustering using Python

Fernando Chirici
E-mail address
fernando.chirici@stud.unifi.it

Abstract

In this paper will be presented and analyzed the K-means clustering algorithm. Following a small introduction about it's characteristics, it will be reported a implementation that allow to test the algorithm both in a sequential mode and in parallel on multiple cores using the multiprocessing library. The python implementation will enlight the limits of this programming language to operate in such a way and show, throughth the use of the multiprocessing library, how to relieve those limitations.

1. Introduction to K-means

The K-means algorithm, also called *Lloyd algorithm*, is a **clustering algorithm** , i.e. is part of the family of the techniques that aim to partition an input set in one or more output sets named ***clusters***. Specifically, K-means is a parametric clustering algorithm very popular because of it's simplicity.

Starting with random centroids, the algorithm iteratively assign every point of the feature space to one and only one cluster using as a discriminant the minimum euclidean distance from each centroid. At the following steps the centroids are updated and the assignment is repeated until the algorithm doesn't converge or the prefixed maximum number of iteration is reached. The number of cluster obtained in output is a configuration parameter that need to be set before the algorithm is launched.

The K-means algorithm has a linear computational complexity:

$$C = O(nKt) \tag{1}$$

where **n** is the number of input points, **K** is the number of output clusters and **t** is the number of iterations that the algorithm needs to converge (this value, as already said can be limited above). It's easy to see from it's linearity that K-means lack of scalability. The bottle neck is the time spent for every euclidean distance calculated from and to each point and the following assignment to a specific cluster. This limitation can be partially overcome using parallelization. Searching for the minimum euclidean distance is in fact a completely independent task for each point, so it is possible to run multiple instance of this particular trivial task at the same time.
In the next section will be presented an implementation of K-means and a set of experiments using both a sequential execution, where every instruction is executed only after the last one has been computed, and a parallel one, characterized by a subdivision of the work on multiple cores of the CPU.

The experiments on the K-means clustering algorithm have been done working in a 2-dimensional space. This choice has been take for reasons of computational limitation of the device that would have forced to limit the number of points to be analyzed and overshadowed the performance of the parallelization.

2. Python implementation

To develop the Python implementation and testing the performance of the K-means algorithm has been used Pycharm IDE on a Macbook Pro 2017 with a 3,1 GHz Intel Core i7 quad-core hyperthreading to 8 logical cores.
Starting from the pseudo-codes below a high level python implementation has been implemented both for a sequential and a parallelized execution. In both cases has also been tested the effects of multi-threading on the performance.

For the serial algorithm (Algorithm 1), we only have in input the input-set P of points and the number of desired output clusters K that is the only configuration parameter needed. With that given, K centroid are randomly placed in the featured space. Each point in the feature space is then assigned to the cluster that results as the one with the centroid with the lowest euclidean distance to the point itself. In the last step, the centroids are updated following the new topology. This procedure is repeated until convergence or until the maximum number iteration has been reached.

In the parallel version of this algorithm (Algorithm 2), the difference lies in the subdivision between cores of the work of finding for each point the closest centroid on the featured space. The inputs for this algorithm are the input-set P, the number of output clusters K and the number of cores to work on C. As visible in the pseudo-code below, this operation is executed by firstly dividing the data points in *p* subgroups, and only after working on the euclidean calculations, the subgroups are re-aggregated in one single set in order to be able to update the centroids at the following step.

Algorithm 1: Serial K-means algorithm

```
Data: P, K
Place the centroids c_1^0, c_2^0, .....c_k^0 randomly;
while is not reached convergence or maximum iterations do
    for each data point x_i do
        find the nearest centroid(c_1^i, c_2^i..c_k^i);
        assign the point to that cluster;
    end
    for each cluster j = 1..k do
        Update centroids c_i^{i+1} = mean of all points
          assigned to the cluster c_i^i;
    end
end
End;
```

Algorithm 2: Parallel K-means algorithm

```
Data: P, K, C
Place the centroids c_1^0, c_2^0, .....c_k^0 randomly;
Randomly shuffle the data points and partition them
  into p subgroups;
while is not reached convergence or maximum iterations do
    for each subgroup p_i do
        find the nearest centroid(c_1^i, c_2^i..c_k^i);
        assign the point to that cluster;
    end
    Aggregate the results from each subgroup p to
      form K clusters;
    for each cluster j = 1..k do
        Update centroids c_i^{i+1} = mean of all points
          assigned to the cluster c_i^i;
    end
end
End;
```

2.1. Experiments

The experiments on the K-means clustering algorithm have been constructed to gradually upscale the number of points to be clustered and notice the differences between the various performances. Every execution has been run using the *timeit* library, set up in order to run for every experiment 50 different execution and return the average time to complete it. Note that timeit library is set to use a *performance timer* usually used for benchmarks. The times that will be reported are to be interpreted in time relative

to CPU counters and have no connection with the *"real world time"*.

In Figure 1 is shown an execution of the algorithm on different input sized going from 500 to 35000 2D points, in order to obtain 3 output cluster (K). It is very interesting to notice that

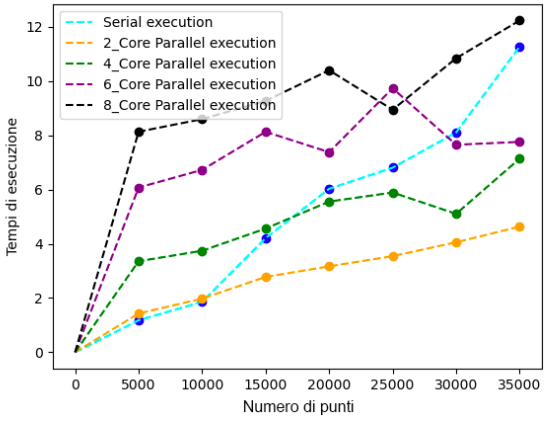

Figure 1. K-means run from 0 to 35k - K=3.

35k points is not yet enough to get from the parallelization over multiple cores the desired positive effect. In fact the cyan line, representing the serial execution of the algorithm is not even the worst among the other, instead that's the 8 core run. This is due to concurrency and core access issues, and the technical time needed in order to do so. Increasing the number of points, this delays no longer weigh so much on the execution. This is shown in Figure 2.
In Table 1 are also reported the speedup gained from the parallelization over multiple cores.

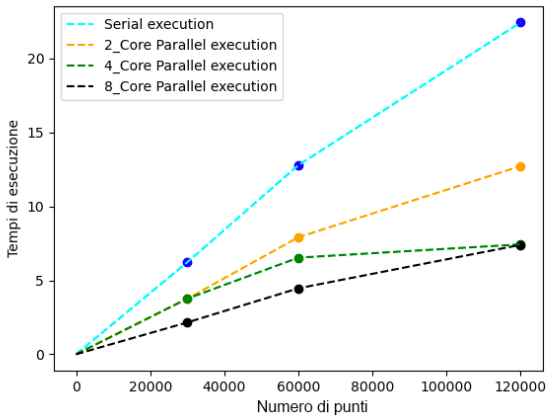

Figure 2. K-means run from 0 to 120k - K=3.

Mode	Points	Time	Speedup
Sequential	30000	6.192	—
2 Core	30000	4.298	1.441
4 Core	30000	4.266	1.451
8 Core	**30000**	**1.980**	**3.127**
Sequential	60000	13.434	—
2 Core	60000	7.223	1.860
4 Core	60000	5.991	2.242
8 Core	**60000**	**4.021**	**3.340**
Sequential	120000	23.919	—
2 Core	120000	12.942	1.848
4 Core	120000	6.373	3.753
8 Core	**120000**	**6.225**	**3.842**

Table 1: K-Means parallel speedup - K=3

Clustering over big amount of data clearly shows a gap between serial and parallel execution. The sequential execution takes an enormous amount of time, almost six times the 8 and 4 core executions, to run the clustering algorithm on 120k points. Still, the limits of K-means and Python multiprocessing performance needs to be noted. Even if the parallel executions clearly outperforms the serial one, those still needed a lot of hours to terminate every run. The linear nature of the complexity of the algorithm means a linear strong increase on the number of operations needed in order to compute K-means, to which Python is not able to cope with. In fact, even though Python is an interpreted language, it first gets compiled into byte code.

This byte code is then interpreted and executed by the Python Virtual Machine. The act of compilation and following execution drastically slows down Python code and doesn't make it a good candidate to develop this kind of algorithms.

Increasing the number of output cluster also increase the number of operations so it's natural to expect worst performance for every kind of run. In the experiment results shown in Figure 3, every data have been left the same but the number of output cluster set to K = 50. As expected the

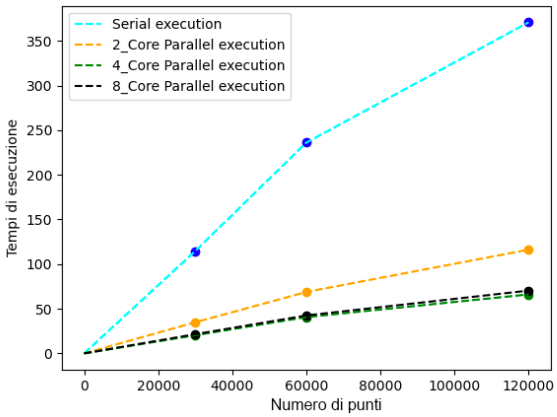

Figure 3. K-means run from 0 to 120k - K=50.

total time of execution for every run drastically increase and the sequential execution moves even further away from the other one. An interesting fact to be notice is the similarity between the 8 and 4 core executions. In fact, in the last example the 4 core actually slightly outperform the other. This issue can be addressed to a lack of capacity from Python multiprocessing to succeed on parallelize on every of the 8 logical cores and only really using the 4 physical ones. The same issue is observable in Figure 4 and seems to be more present when facing a big amount of work. For example, notice that in Figure 2, elaborating only 3 output cluster, the same issue only appeared on the elaboration of the 120k input set. In this last experiment the algorithm is again set up in order to generate 3 output cluster on the same amounts of increasing data but instead of multiprocessing,

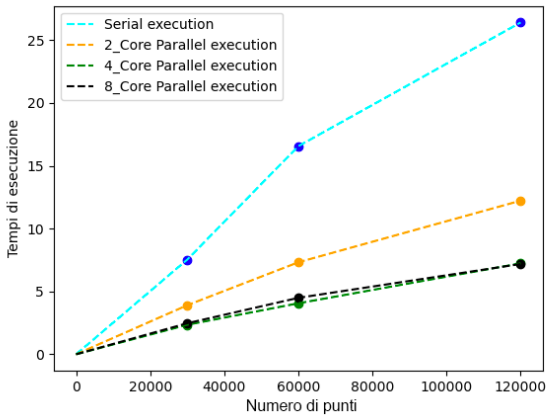

Figure 4. K-means threading run from 0 to 120k - K=3 - Max Workers=100.

a multithreading approach is performed using the library *ThreadPoolExecutor*. In this scenario almost no limit has been set on the number of workers (set to max 100).
Comparing with Figure 2, it's visible the speedup of the 2 cores execution and the slight improvement of the 4 and 8 cores runs.

3. Conclusions

In these experiments has been possible to observe clearly the speedup gained by running multiple process in parallel to compute due distances calculation of K-means clustering. Starting from around 35k points, the sequential run became the worst case run and from there on it stays like that. The biggest difference between the parallel execution stands between the 2 cores and the other runs. This is both because the first one simply uses less cores then the others in order to compute the algorithm, and also because of the problem discovered about Python's multiprocessing library not being able to compute efficiently using the 4 remaining hyper-threaded logical cores of the CPU, but only working the 4 physical ones. At last, switching to a multithreading scenario it's been possible to notice a significant speedup for the 2 cores execution and a slight improvement for the 4 and 8 cores runs. Specifically embedding the multiprocessing pool directory allowed the script to gain up to 6.225 speedup.

References

[1] H.-H. Bock. Clustering methods: a history of k-means algorithms. *Selected contributions in data analysis and classification*, pages 161–172, 2007.

[2] wikipedia$_2$022. $K-means clustering.wikipedia$, 2022.