Lab 1. K-Means Parallelization in Python

Victoria Anguix López, Lara Monteserín Placer, María Ferrero Medina

October 2023

1 Introduction

The K-Means algorithm is a clustering method used to separate the observations from a dataset into different clusters. The objective of this report is to document the parallelization of the K-Means algorithm in Python applied to a randomly generated dataset and determine which is the best alternative to parallelize the algorithm in terms of the execution time. The reason behind implementing parallelization is reducing the execution time of the algorithm and improving its performance compared to the serial implementation.

To perform this analysis, a computers' dataset will be used. The dataset has nine different variables regarding the computers characteristics: price, speed, hd, ram, speed, cores, cd, laptop and trend. This dataset will be artificially generated, so different sizes can and will be chosen to test and compare the performance.

Once the algorithm is working, the maximum average price for a cluster and the corresponding cluster will be determined. In addition, a two dimensions scatter plot will be represented showing the price and speed characteristics. Also, a heat map will be represented with the centroids of the clusters and the variables in the dataset.

For this report two parallelization approaches were considered: multiprocessing and threading. Both of them are already implemented in Python libraries and will be compared with the serial approach.

2 K-Means Algorithm

The K-Means algorithm is used for clustering problems, in which the original dataset with n observations wants to be splitted into k different clusters (k< n). Each data point will belong to the cluster with the nearest mean. The objective is to minimize the variance within each cluster and maximize the variance between clusters.

For K-Means algorithm, the choice of the number of clusters, k, is crucial. Depending on how the points are distributed along the dataset, the optimum number of clusters may vary. This is

the reason why it is interesting to plot the Elbow Graph, that represents the values of the Within-Cluster Sum of Squares(WCSS), that is to say, the variance, for different values of k [1]. It is required to apply the K-Means algorithm for all the k values considered, this is the part where the parallelization will be applied. Each k value will be a separated task that can be performed independently from the rest .

Once the values of different WCSS with their respective k are known, the Elbow Graph can be plotted. From this point on, the program will be serial again. The serial part of the code will plot the heatmap, the two dimensions scatter plot, and retrieve the maximum average price for a cluster.

3 Methodology

The main objective of this report is to determine which is the best alternative to parallelize the K-Means algorithm and to verify that parallelization indeed reduces the execution time. To decide the most suitable procedure, the speedup will be calculated for different sizes of the datasets and for both, the multiprocessing and the threading implementations. To begin with, for developing and testing, small datasets were considered (5,000 rows) for the program to run faster. Once the program was running, to compare the performances, four datasets of different sizes have been used (from 5,000 to 5,000,000 rows).

To be able to compare the procedures, three different scripts have been created:

- *Computer-serial.py*: it implements the serial version of K-Means algorithm from scratch, without using machine learning libraries. It will be useful to compare the execution time of the parallelized algorithm and to be able to compute the speedup.
- *Computer-mp.py*: it makes use of multiprocessing library in Python to parallelize the previous version.
- *Computer-th.py*: it makes use of threading library in Python to parallelize the serial version.

The three scripts perform the same operations: implementing the K-Means algorithm, representing the elbow plot to decide the most adequate value for K, calculating the cluster with the highest price, plotting of the first two dimensions, speed and price, colored by its clusters, and finally, plotting a heat map using the values of the clusters centroids. Following on from this, we will looking in more detail at how each script operates.

3.1 Computer-serial.py

First, the serial implementation of the algorithm. To organize the different actions, the code was divided into different functions:

- *initialize_centroids*: centroids are the center points in each cluster. For the algorithm to start running, a first approximation of the centroids is made using k random points from the dataset as centroids.

- *k_means*: it iterates using the centroids from the previous iteration and changing the position of the centroids to reduce the distance between the points and the centroids. If this process reduces the above mentioned distance, then the centroids are updated. This is the iterative part and it is repeated until the algorithm converges, this is when the previous centroids and the actualized centroids are equal.

- *calculate_wcss*: this function calculates the WCSS. This is the sum of the squared difference between each point and the centroid of the cluster, equivalently, the variance within each cluster.

- *plot_elbow*: given the values of k and the values of the WCSS, this function plots the elbow plot, representing both magnitudes.

- *calculate_average_price*: a specific function was created to calculate the average price for one particular cluster, adding all the prices and dividing by the number of elements in the cluster.

- *plot_2D*: this function prints a scatter plot of the given points by colors depending on the cluster each of them belongs to, and also the centroids for each cluster. As the dataset has nine dimensions, only two can be considered for the scatter plot and these are: speed and price.

- *plot_heatmap*: this function uses the *heatmap* function from the seaborn library [2], specialized in statistics and data visualization and plots the heatmap for the k clusters and the 9 dimensions.

3.2 Computer-mp.py

To apply parallelization with the *multiprocessing* library [3], the main task will be split into several processes. These processes can run independently in separated locations of the memory.

Before representing the Elbow Graph, the values for WCSS and k must be determined. The computation of the different values for WCSS for different k values was the parallelized part of the code. First of all, to achieve parallelization with multiprocessing a Pool object has to be created, for the processes to be located there. Then the processes are created in the parallelized part using the apply function, so one process is created for each k value. Finally, it is required to finish the parallelization and this is achieved first closing the pool, this is preventing any more tasks from being submitted to the pool. Also, to guarantee that all the processes in the pool have finished, it

is necessary to join the processes after closing the pool for the program to wait until all of them are completed.

Furthermore, to parallelize also the computation of the distances, the function *cdist* [4] from the scipy library was used, inside the function *k_means*. This aims to achieve better performance than the serial computation.

3.3 Computer-th.py

Alternatively, threading divides the main process in several threads. Unlike processes, threads share the same space in memory, and they can not be executed at the same time. Multithreading refers to the concurrent execution of more than one sequential thread of instructions, and it has been implemented with the Python library *threading* [5].

Similar to the case of multiprocessing, with threading the parallelization was applied while calculating the WCSS for different k values before representing the Elbow Graph. Additionally, more performance was achieved by changing the function *k_means* on the moment of calculating the distances. The function *cdist* [4] from the scipy library was used, which performs a parallelize version of the distance calculation.

In order to perform a multithreading version of the code, two facts must be have in mind. First, we must enclose the K-Means implementation into a target function that will be used for each thread initialization. This function is called *start_k_means*, as for the multiprocessing case, however in the case of threads we have a difference. The second fact to have in mind is that, as threads can not be executed at the same time, they need of a threading lock for synchronizing the results, that in case of K-Means and elbow plot, it is of utmost importance. Then, the target function saves the result differently than in multiprocessing case.

Finally, the main branch of the program execute it equally to the serial version, having all the plots at the end for not stopping the execution, and measuring the time for the K-Means implementation.

4 Results

4.1 K-Means results

4.1.1 Choice of K: Elbow Graph

The goal of the K-Means algorithm is to minimize the WCSS. The more clusters considered, the smaller the WCSS will be. There is a critical value for the number of clusters, kin which the WCSS is not reduced significantly for larger values of K. This is shown graphically as a change in the slope, an elbow. For choosing the optimum value for the number of clusters, K, the Elbow Graph representation has been studied. This graph represents the WCSS for different numbers

of clusters. The value of k depends on the distribution of the dataset observations. Then, for different datasets, different values of k should be chosen and in each case the Elbow Graph should be examined. For example, for the small dataset (of 5,000 rows) the optimum value for k is 4 as shown in Image 1. Another example is the large dataset (with 5,000,000 rows) that reaches the elbow with k equal to 3.

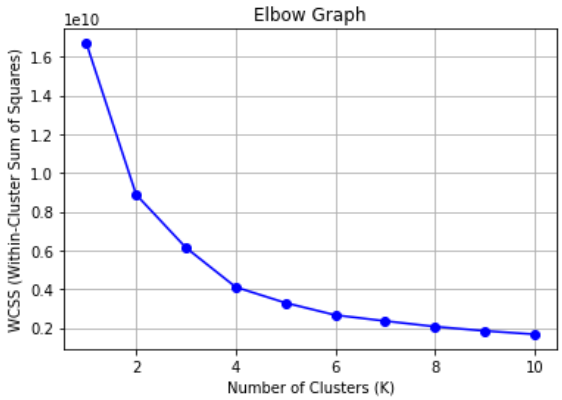

Figure 1: Elbow Graph for the small dataset (5,000 rows)

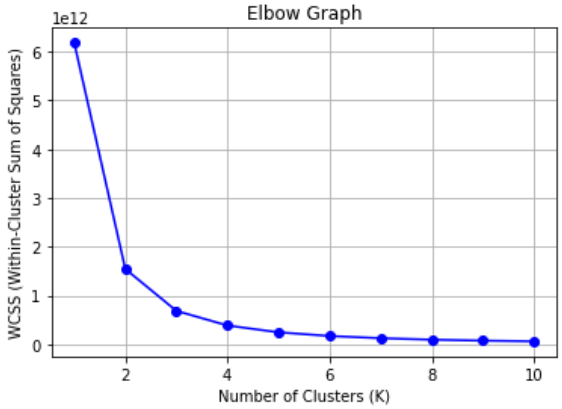

Figure 2: Elbow Graph for the large dataset (5,000,000 rows)

4.1.2 Maximum price

For each cluster the average price was calculated and the maximum for all the clusters was retrieved. The same outcome was achieved with the three scripts. This confirms that the three procedures retrieve the same results although through different computational structures.

For the large dataset, the cluster that retrieves the maximum price is the cluster labeled as cluster 1 and the average price of that cluster is 2428.47.

4.1.3 2-Dimensions Scatter Plot

The scatter plot shows the distribution of the points in the dataset using only two variables as descriptors, in this case speed and price. The points are coloured depending on the cluster they belong to and the centroids are indicated with a red cross.

Figure 3: Scatter plot showing price and speed for the large dataset (5,000,000 rows)

4.1.4 Heatmap

Heatmap is a representation of the normalized deviation of the average cluster value for one parameter compared to the average value for the whole dataset. This allows us to analyze which variables are more significant in the cluster assignment and if the model is accurate or if there are two or more clusters that could be considered as the same cluster. According to Figure 4 the number of clusters selected seems consistent, for most variables the colors are different for each cluster, which means they are different. Heatmap is a useful tool and in this case it confirms that the number of clusters is adequate.

4.2 Parallelization results

4.2.1 Performance, Execution Time and Speedup

The methods will be compared using the execution time T in seconds, the speedup S and the performance P. Execution time is the measure of the time it takes for the program to be executed, it is directly measured in the code. Performance is defined as the the inverse of the execution time, and therefore, measured as inverse of seconds.

$$P = \frac{1}{T}$$

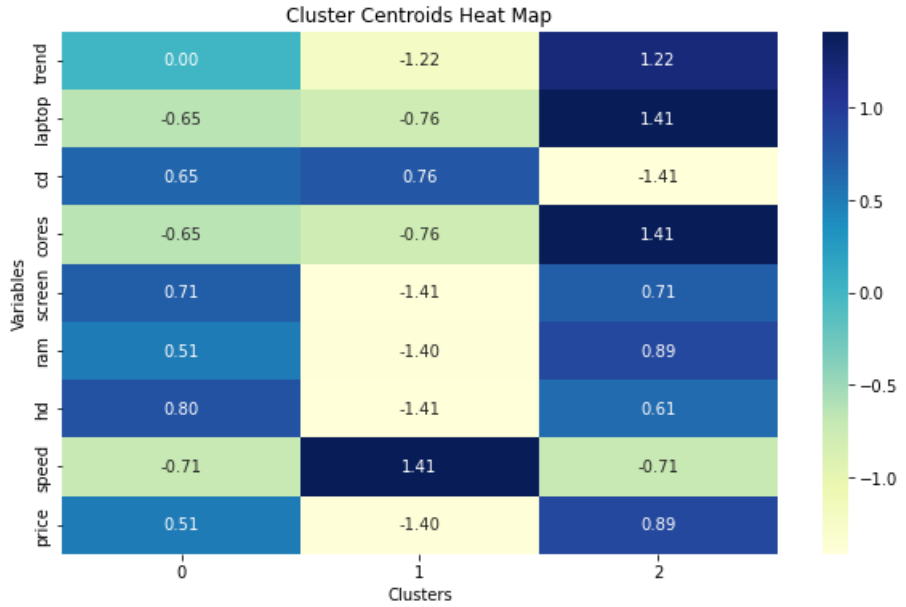

Figure 4: Heat Map representation for the large dataset (5,000,000 rows)

The speedup is a measure of how fast the parallelized program is compared to the serial implementation. It is defined as the ratio between the time for the serial computation by the time for the parallelized computation.

$$S = \frac{T_{serial}}{T_{parallel}}$$

Dataset Size	Serial	Multiprocessing	Threading
5,000	1.59	5.85	0.66
50,000	24.87	24.97	4.54
500,000	224.56	64.17	43.32
5,000,000	3084.76	795.06	678.37

Table 1: Total execution time in seconds for different dataset sizes

Results from Table 1 and Table 4 show that parallelization begins to be significantly useful for large datasets. In the case of multiprocessing, for the small dataset, the parallelized computation takes longer than the serial. As the dataset grows larger, multiprocessing presents higher speedup. On the other hand, threading is, for all the cases considered, faster than the serial computation but reaches a maximum speedup for the dataset size of 50,000 rows.

Dataset Size	Multiprocessing	Threading
5,000	5.76	0.61
50,000	24.29	4.09
500,000	61.47	37.69
5,000,000	769.08	620.33

Table 2: Execution time for the parallelized part in seconds for different dataset sizes

Dataset Size	Serial	Multiprocessing	Threading
5,000	0.629	0.171	1.51
50,000	0.0402	0.0400	0.220
500,000	0.00445	0.0156	0.0231
5,000,000	0.000324	0.00126	0.00147

Table 3: Performances for different dataset sizes

4.2.2 Amdahl's Law

Amdahl's Law is a formula that calculates the theoretical speedup for a given task when parallelization is applied. The formula considers the fraction of tasks that are parallelized p, the fraction that is not parallelizable $(1-p)$ and the number of cores that is going to be used to parallelize n.

$$S(n) = \frac{1}{1 - p + \frac{p}{n}}$$

Based on the previous results, the theoretical speedup can be calculated if the number of cores is known and the fraction of the code that will be parallelized is also known. The number of cores for all the parallelized tasks in this report is the maximum possible for the computer that was used, which is 4.

The fraction of code that is parallelized depends on the dataset size, for larger datasets, the calculation of the WCSS for different k values consumes more time than in smaller datasets. So, in the serial implementation of the code, two times were measured. On the one hand, the total execution time. On the other hand, the execution time of the loop that will be parallelized later. So p can be defined as the quocient between those two time measures.

$$p = \frac{T_{parallel}}{T_{total}}$$

If comparing Table 5 with Table 4 the differences between the theoretical and the real speedup are shown.

Dataset Size	Multiprocessing	Threading
5,000	0.27	2.41
50,000	0.99	5.48
500,000	3.50	5.18
5,000,000	3.88	4.55

Table 4: Speedup for different dataset sizes

Dataset Size	Total time	Loop time	p	$S_{theoretical}$
5,000	1.593	1.530	0.960	3.57
50,000	24.87	24.25	0.975	3.72
500,000	224.56	216.75	0.965	3.62
5,000,000	3084.76	3007.64	0.975	3.72

Table 5: Theoretical speedup for different dataset sizes

5 Conclusion

Parallelization can present advantages in terms of performance. This report shows that both the threading and the multiprocessing speed up the processes for large datasets. Contrary to that, for small datasets multiprocessing is slower than serial computation.

The low performance observed in multiprocessing can be produced because the creation of many small processes can take longer than computing them. If this happens, the overhead of creating a lot of processes can outweigh the benefits of parallelization. But, as the processes become more complicated (when enlarging the dataset), multiprocessing presents a larger speedup.

Regarding the theoretical values of speedup, it can be observed how for the larger datasets both methods have reached the maximum, even surpassing it in the case of the threading implementation. Therefore, threading could be the best alternative for all the cases considered. However, it is observed that when the dataset grows larger, the speedup with threading decreases and the speedup with multiprocessing improves. Having this in mind, for this particular problem the multiprocessing solution would be more scalable than threading, as for larger datasets (or more complex problems) multiprocessing continues increasing on speedup, but certainly for the datasets managed during this report threading is the best alternative due to higher values of speedup in all cases.

References

[1] Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. "The k-means Algorithm: A Comprehensive Survey and Performance Evaluation". en. In: *Electronics* 9.88 (Aug. 2020), p. 1295. ISSN: 2079-9292. DOI: 10.3390/electronics9081295.

[2] URL: https://seaborn.pydata.org/generated/seaborn.heatmap.html.

[3] URL: https://docs.python.org/3/library/multiprocessing.html.

[4] URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html.

[5] URL: https://docs.python.org/3/library/threading.html.