Parallelized k-means algorithm for data clustering

Juan Luis Jurado Esteve, Julia Gómez Concejo,
& Javier Rodríguez Valverde

Master in Big Data Analytics

Statistics for Data Analysis

Universidad Carlos III de Madrid

October 24, 2023

1 Serial code

In order to load the data, we first load it as a data frame using `pandas`, which is useful to read `.csv` documents. We select the nine last columns, excluding the `id` column, and replace the cualitative values with symbolic numbers. Finally, we convert this data into an array, which is more useful in numerical analysis.

```
# import packages
from scipy.spatial.distance import cdist
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import seaborn as sns
# read and edit the csv file
df=pd.read_csv('computers.csv', delimiter = ',')
new_df=df.iloc[:,1:]
names = new_df.columns.tolist()
new_df.replace({'cd': {'no': 0, 'yes': 1}, 'laptop': {'no': 0, 'yes':
    1}}, inplace=True)
datos=np.array(new_df)
```

Afterwards, we will define our main `kmeans` function, with the number of clusters, `k`, and the data, `X`, as arguments. We also set a fixed number of iterations and a seed, in order to generate the random numbers in the same order and be able to compare the different codes properly. In order to select the centroids of the clusters, we have chosen to select a random row of the data. This is the optimal way of selecting the centroids, since there will always be one point belonging to that cluster (the centroid itself), and choosing a row is faster than generating a centroid which is close enough to the data.

```
# define the main function
def kmeans(k,X,max_iterations=200):
  it = 1
  # set seed
  np.random.seed(42)
  # randomly choose rows of the data as first guess centroids
  centroids = X[np.random.choice(X.shape[0], k, replace=False), :]
```

We start a loop which divides the data into clusters and relocates the centroids in each iteration. Firstly, we calculate the distance of each data point to all the centroids and select, for each data point, the minimum distance to a centroid and the column-index of such distance, which gives us the cluster to which that data point belongs.

```
  # divide the data for some iterations or until convergence
  for _ in range(max_iterations):
    cluster_indices=[]
    cluster_centers =[]
    # divide the data into clusters based on the distance
    # from the data to the centroids
    distances=cdist(X,centroids)
    belong_to_cluster=np.argmin(distances,axis=1)
    dist_min= distances[np.arange(len(distances)), belong_to_cluster]
```

We then divide the indeces of the data points into lists, where each one corresponds to a cluster.

```
    # create a list of lists that shows the indeces of the rows
    # of data set into each cluster
    for i in range(k):
      cluster_indices.append(np.argwhere(belong_to_cluster==i))
```

Using the indices in each list, we compute the cluster center by computing the mean of all the data points corresponding to a cluster.

```
    # find the cluster center
    for indices in cluster_indices:
      cluster_centers.append(np.mean(X[indices], axis=0)[0])
      cluster_centers=np.array(cluster_centers)
```

Finally, we have set a tolerance conditions, which stops the process if the chosen centroids are close enough to the cluster centers. However, if they are not close enough yet, we relocate the centroids as the center of the clusters and repeat the loop for the next iteration. When it finishes, the main function returns the labels of the data, which show to which cluster they belong, the list that divides the indeces of the data into lists, the final centroids, the WCSS and the number of iterations that the loop has taken before finishing.

```
    if np.max(centroids-cluster_centers)<0.001:
      break
    else:
      centroids=cluster_centers
      it += 1
    # return the division of data into clusters
    return belong_to_cluster, cluster_indices, centroids,
     np.sum(dist_min**2), it
```

We then create a function which computes the average price (which is the first column of the data) of each cluster using the list of lists that includes the indeces of the data points that belong to each cluster.

```
## Set function for average price calculation
def price_avg(X,c_i,k):
  prices = []
  for data_point in c_i[k]:
    prices.append(X[data_point,0])
  return np.mean(prices)
```

Finally, we create a list of all of the results for a number of clusters in the range of [1,12], to represent the elbow graph using the Within Cluster Sum of Squares (WCSS) and see what is the optimal number of cluster for the division. We use the command `time.time()` to start measuring the time that the code takes to compute the results that we use to represent the elbow graph, and we use the same command again when it finishes, taking the elapsed time as the difference between those two measures.

The optimal number of clusters for dividing the data is defined as that for which the elbow graph stabilizes, that is, the minimum of the third derivative of the function WCSS(k), k being the number of clusters, with a small margin of error. In addition, for the optimal number of clusters, the cluster with the greatest average price is computed.

```
# start chronometer
resultados = {}
WCSS=[]
K = np.arange(1,13)
start_time = time.time()
for k in K:
  # run the function for k clusters and fit the data
  resultados[f"resultados{k}"] =kmeans(k,datos)
  WCSS.append(resultados[f"resultados{k}"][3])
## Compute the optimal k using the minimum of WCSS 3rd derivative
first_derivative = np.gradient(WCSS)
second_derivative = np.gradient(first_derivative)
third_derivative = np.gradient(second_derivative)
optimal_k = np.argmin(third_derivative)+int(len(K)/5)
print("Optimal number of clusters:", optimal_k+1)
## Get the cluster division for the optimal k
results = resultados[f"resultados{optimal_k+1}"]
print('Convergence for ', optimal_k+1, ' clusters achieved in ',
   results[4], ' iterations')
```

```
## Find the cluster with the highest average price
price_avgs = [price_avg(datos,results[1],k) for k in
    range(optimal_k+1)]
print(price_avgs)
print('The cluster with the highest average price is cluster number ',
    np.argmax(price_avgs)+1)
# stop chronometer
end_time = time.time()
# compute and print elapsed time
elapsed_time = end_time - start_time
print("Tiempo transcurrido:", elapsed_time, "segundos")
```

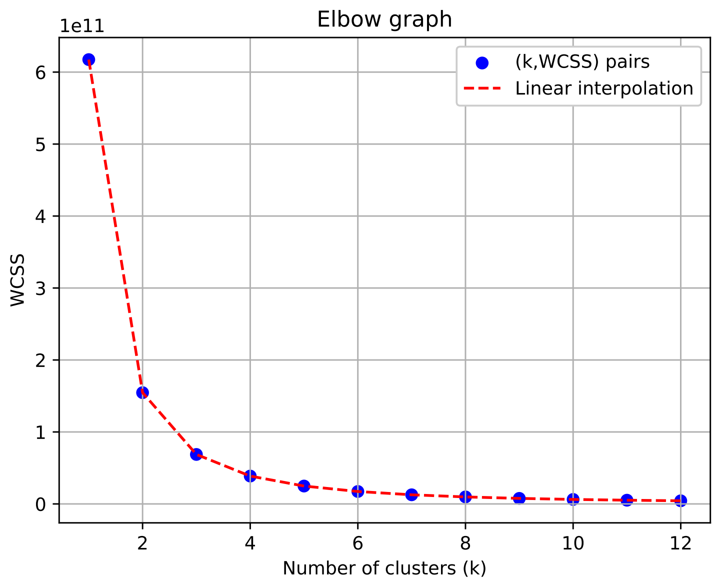

Figure 1: Elbow Graph for 500 000 data.

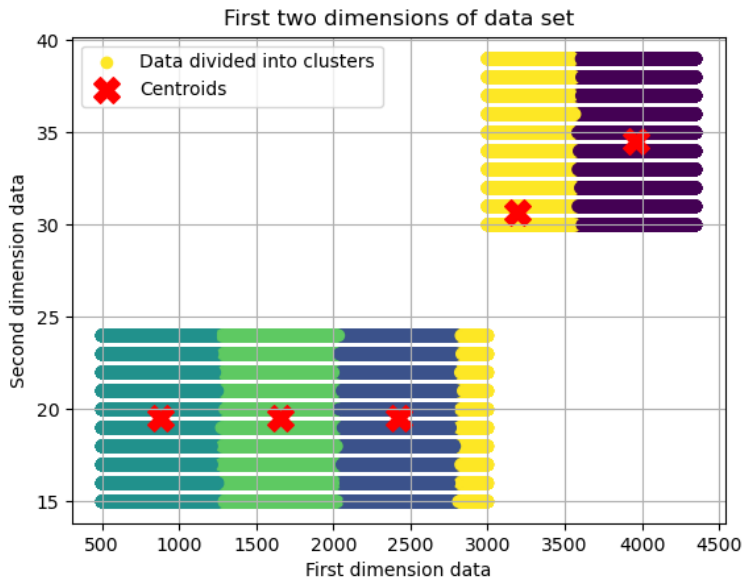

Figure 2: Plot of the first 2 dimensions of the 500 000 data divided into the optimal number of clusters.

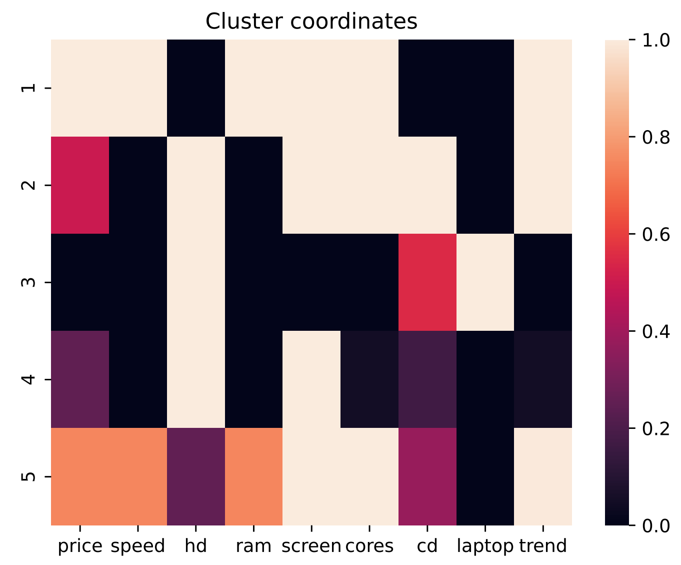

Figure 3: Heat map for all the dimensions of the 500 000 data.

Once checked that the code runs as expected with 500 000 data points, we tested for 5 000 000, which is going to be our final result.

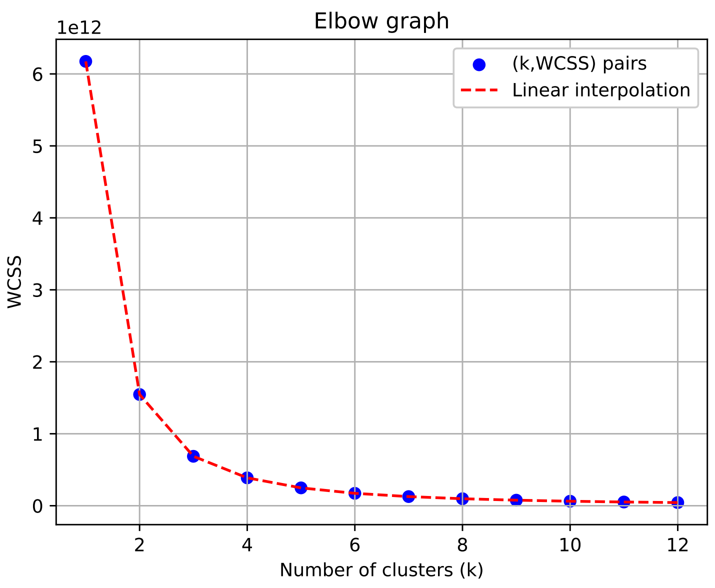

Figure 4: Elbow Graph for 5 000 000 data.

From the figure, it is clear that the WCSS gets smaller as we increase the number of clusters. However, the increase between 11 and 12 cluster is negligible, therefore we try to balance minimizing the WCSS and simplifying the problem as much as we can. Thus, the optimal number of clusters obtained is 5, when the elbow has passed and the slope stabilizes.

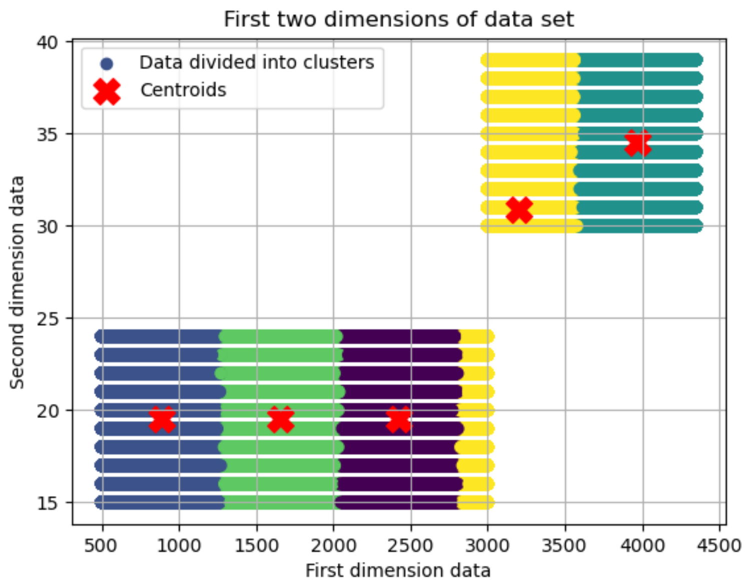

Figure 5: Plot of the first 2 dimensions of the 5 000 000 data divided into the optimal number of clusters.

In this plot we have projected the data points onto the first two dimensions: price and speed. We can see that the centroids are moreless where we expect them to minimize the distance within points in the cluster, as predicted by the WCSS.

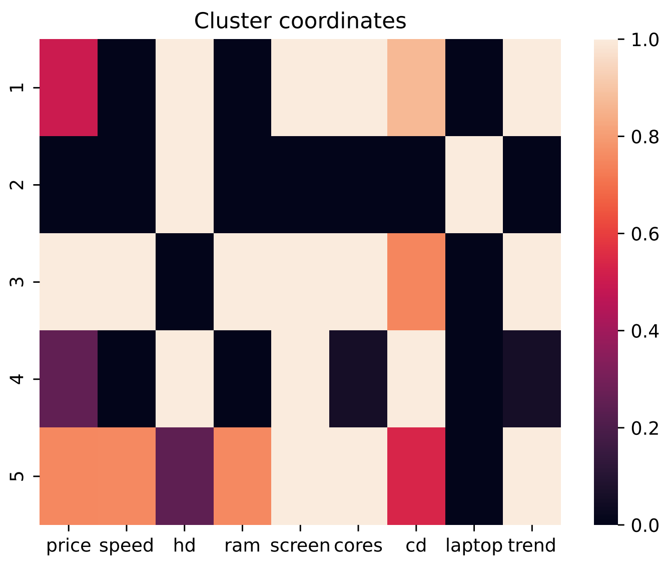

Figure 6: Heat map for all the dimensions of the 5 000 000 data.

Finally, the heat map shows the different distribution of the coordinates for the centroids of each cluster. We have normalized the centroids' coordinates because each dimension had a different order of magnitude, difficulting the visualization of the data.

$$C' = \frac{C - min(C)}{max(C) - min(C)}$$

Where C is a matrix where each row are the coordinates of a centroid. This guarantees that the data is normalized between 0 and 1.

2 Process parallelization

In this section we will discuss how we carried out the parallelization of the code above using the python package multiprocessing and its Pool class.

When paralellizing the serial code using processes it is necessary to take into account the time needed to create the necessary processes as well as calling these processes with functions such as pool.apply or pool.map.

For this reason, our tests have determined that introducing parallelization inside the kmeans function does infact not speed up the run time. The reason is that these processes need to be called in each iteration of the main loop (at least) and this takes much more time than the actual computation itself.

Taking this into account, we found that the only computation for which process parallelization actually gains time is for the computation of the WCSS for several number of clusters k and the elbow graph. This is because the serial program is already very optimized thanks to the different python libraries used.

Given that the tasks we want to parallelize are homogeneous (iterations of kmeans with different k), the best way to parallelize it with multiprocessing is using the Pool class rather than the Process class. We have tried several functions to try and get the best time possible:

1. Asyncronous functions end up taking more time because we need to retrieve the results for the optimal k, which takes additional computation since we loose the order.
2. Since the iterating function has two arguments, we were left with the Pool.apply and Pool.starmap functions. Our tests have determined that the latter is almost 4 times faster than the former.

```python
if __name__ == "__main__":

    pool = mp.Pool(4)

    ## Run kmeans for k clusters
    test_for_k = pool.starmap(kmeans,[(k,data) for k in K])
    for k in (K-1):
        WCSS[k] = test_for_k[k][3]

    pool.close()
```

Given that the used CPU had 4 physical cores, our tests predictably returned that the optimal number of processes was 4.

Now we will present the results of this code and compare them with the others, in order to show that parallelization does not affect the results in any way, only the performance (as shows by the speedup).

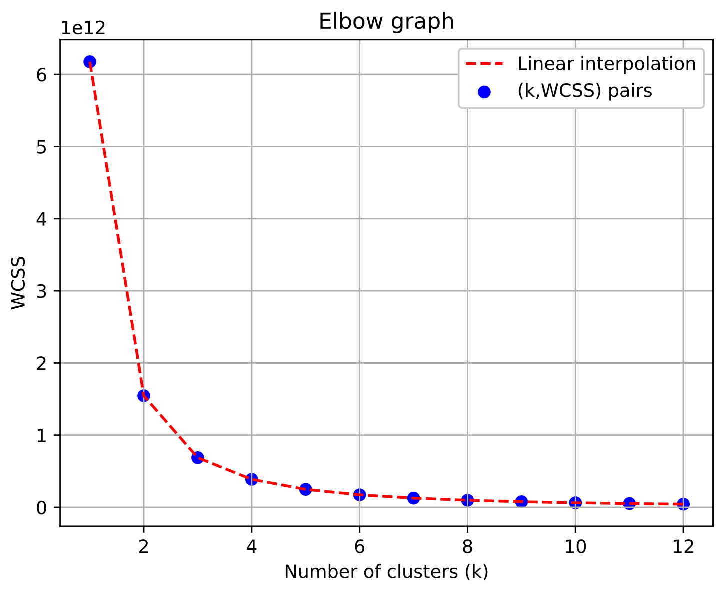

Figure 7: Elbow graph for 5 000 000 data with multiprocessing.

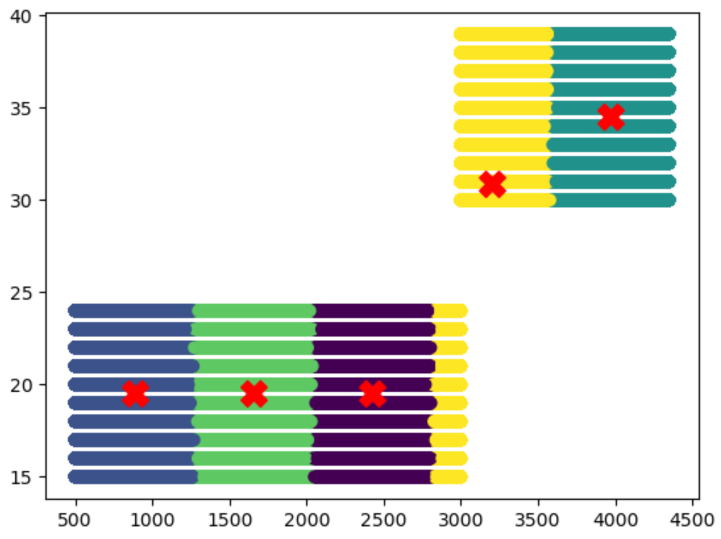

Figure 8: Plot of the first 2 dimensions of the 5 000 000 data divided into the optimal number of clusters with multiprocessing.

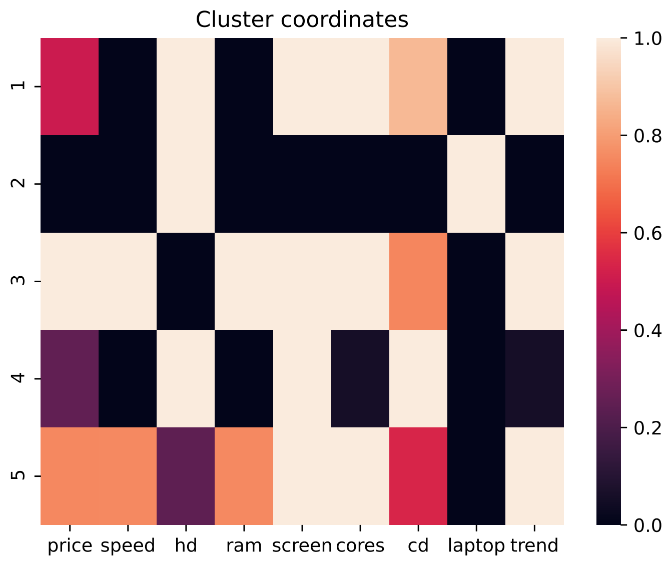

Figure 9: Heat map for all the dimensions of the 5 000 000 data with multiprocessing.

As we can see, the graphs are identical to the other cases.

3 Threads parallelization

In this section, we will implement threading to parallelize the serial code and analyze its performance in comparison to the serial version. We explored several methods of implementing threads, each yielding different results.

We experimented with three approaches: parallelizing the K-means function internally, parallelizing it externally, assigning different values of 'k' to each thread, and a combination of both. It is essential to consider that thread creation incurs some overhead, so parallelization may not always outperform the non-parallelized version.

Among the three methods we tested, the external parallelization approach yielded the best results. This is the one we implemented:

```python
threads2 = []
for k in K:
    thread2 = threading.Thread(target=kmeans, args=(k, data, 200))
    threads2.append(thread2)
    thread2.start()
for thread2 in threads2:
    thread2.join()
```

In this code, the primary concept is to assign one thread to handle the entire workload for a specific 'k' value. As in previous experiments within this lab, we varied the number of clusters to find the optimal configuration, which is a resource-intensive operation for the computer. Parallelization was expected to significantly improve performance.

The code begins by creating an empty array to store the threads. A 'for' loop iterates through all the 'k' values. The first step involves creating a thread, targeting the 'kmeans' function, and storing it. Subsequently, the threads are started, and, finally, they are joined together once they have completed their calculations. The 'join' operation ensures that all threads have finished, with each thread storing its results in an array.

The results obtained with the dataset of 5,000,000 data points will be presented in Figures 9, 10 and 11. These results, as observed, remain consistent. The primary goal of parallelization is to enhance the efficiency of the process without altering the outcomes.

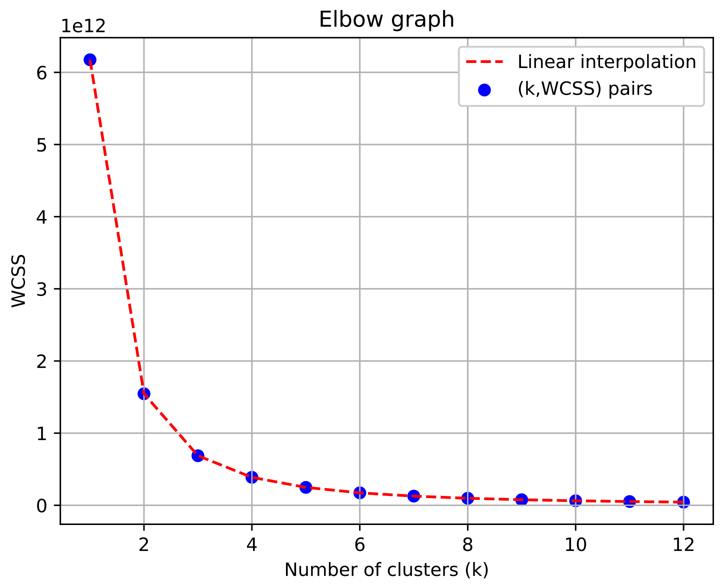

Figure 10: Elbow Graph for 5 000 000 data (Threading).

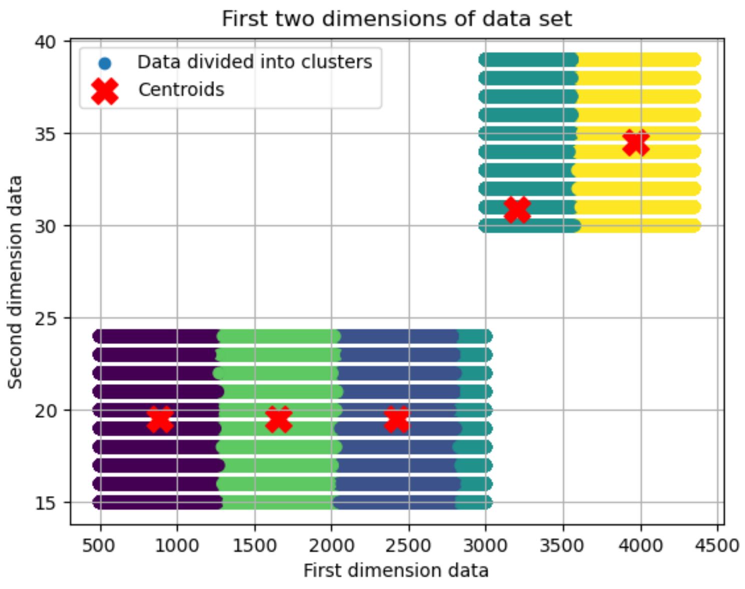

Figure 11: Plot of the first 2 dimensions of the 5 000 000 data divided into the optimal number of clusters (Threading).

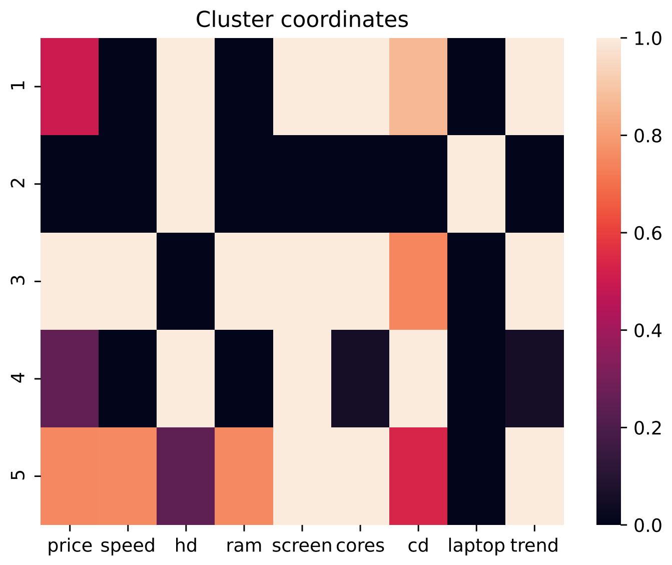

Figure 12: Heat map for all the dimensions of the 5 000 000 data (Threading).

As we can see, the graphs are identical to the other cases.

4 General Comparison

In this section we will compared the performance of the three different methods. As we have shown above, this was done by comparing the elapsed time between the call to the function for building the elbow graph and the computation of the average price (included). As expected, we can conclude from table 1 that both of the parallelizing methods improve the performance with respect to the serial code, with a greater improvement by using the threads method. However, it must be noticed that after increasing the number of data, the threads method reduces its improvement with respect to the serial code, while the multiprocessing method increases it. The threads method remains to be the best, but this leads us to believe that if the number of data is increased, the multiprocessing method may become a better option than the threads method.

Data	**Opt. clust.**	**Pricy clust.**	**Serial**	**Threads**	**Mult.**	
500000	5	1	16.07	54.02	37.75	**Performance (mHz)**
				3.36	2.35	**Speedup**
5000000	5	3	2.34	7.04	5.66	**Performance (mHz)**
				3.01	2.42	**Speedup**

Table 1: Column description from left to right: number of data, optimal number of clusters, number of the cluster with the greatest average price, performance of the serial code, performance of the threads code, performance of the multiprocessing code.

Where the speedup and performance were computed through the usual definition.

$$\text{Performance} = \frac{1}{\text{Time }(ms)}, \qquad \text{Speedup} = \frac{\text{Time }(seq)}{\text{Time }(par)} = \frac{\text{Performance }(par)}{\text{Performance }(seq)}.$$