PC-2018/19 Mid-term Project: Implementation of K-Means using Pthreads

Matteo Ghera
matteo.ghera@stud.unifi.it

Luisa Collodi
luisa.collodi@stud.unifi.it

Abstract

In this article we analyze and compare the performances of our sequential and parallel implementation of K-means on a set of points. We have used the C programming language and the Pthreads library for the parallel implementation.

1. Introduction

The K-means is a non-hierarchical partition algorithm for clustering a set of objects. The generic algorithm is described for example in [2] and it is shown below:

Algorithm 1 K-means

Select *K* points as the initial centroids.
repeat
  Form *K* clusters by assigning all points to the closest centroid.
  Recompute the centroid of each cluster.
until The centroids don't change.

We assumed that the objects are bidimensional points, the *K* initial centroids can be inserted by the user, the closeness between one bidimensional point and each centroid is evaluated using the euclidean distance, finally, new centroids are calculated as the mean of all the points of each cluster.

We used the Minkowski distance to measure the closeness between a point and a cluster:

$$d = (\sum_{i=1}^{n} |x_i - y_i|^p)^{\frac{1}{p}}$$

In particular, we used the Euclidean distance that is the Minkowski distance with $p = 2$.

Subsequently each centroid is computed as the mean of the point of the considered cluster. In fact it can be demonstrated that this choice minimizes the sum of squared error (SSE):

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} (c_i - i)^2$$

$$c_i = \frac{1}{m_i} \sum_{x \in C_i} x$$

where C_i is the i-th cluster, c_i is the centroid of the i-th cluster and m_i the number of points of the i-th cluster.

1.1. Language

We have used C programming language and the Pthreads library for the parallel implementation. An overview of P-thread functionality can be found in [1, 3].

2. Implementation

In this section we introduce the main ideas we followed writing the code.

2.1. Data representation

We have used two different structures to represent clusters and points.
A `Cluster` has the following attributes:

- *cx*: the sum of the abscissas of the points contained in the cluster
- *cy*: the sum of the ordinates of the points contained in the cluster

- *label*: a string which represents the name of the cluster
- *next*: a pointer to the next cluster

```
struct Clusters {
     double cx;
     double cy;
     char* label;
     struct Clusters* next;
};

typedef struct Clusters Clusters;
```

Instead a `Point` is so defined:

- *x*: abscissa of the point
- *y*: ordinate of the point
- *cluster*: pointer to the cluster it belongs to
- *next*: a pointer to the next point read in the .csv file

```
struct Points{
    double x;
    double y;
    struct Clusters *cluster;
    struct Points *next;
};

typedef struct Points Points;
```

We therefore maintain a pointer to the first cluster and to the first point read, respectively: `clusters` e `head`.

2.2. Reading Data

We assume that the dataset is contained in a .csv file, in which in each line are present abscissa and ordinate of a point, separated by commas. The program scans the file, so it reads the data and for each point it creates a new record (structure `Points`) to be included in the points list.

Similarly, it reads the centroids from files and it creates a new instance of `Cluster` for each centroid. It initializes the created structure with the coordinates just read and with the name of the cluster (also read from file).

Finally the results of the clustering are saved again in a .csv file.

2.3. Sequential implementation

The function that implements the K-means consists essentially of a while loop, in which at each iteration the clusters are updated and the centroids are recalculated.

Clusters are updated scrolling through the list of points and assigning them to their closest cluster. In particular, in the `MakeCluster()` function the entire list of points is scrolled with a pointer and for each point its distance to all the clusters is calculated. The point considered is then assigned to the cluster with the shortest distance, in practice the `cluster` field of the relative `Point` structure is updated with a pointer to the structure that represents the cluster previously identified.

```
while(endCluster==FALSE) {
  double distance=euclideanDistance
                  (nextCluster, nextRecord);
   if(distance<min){
      min=distance;
      minCluster=nextCluster;
     }
     if (nextCluster->next == NULL)
        endCluster = TRUE;
     else
         nextCluster = nextCluster->next;
        }
```

After each iteration, through the function `computesAverangeForAllClusters()`, for each cluster the relative centroid is recalculated with the new points assigned. Its abscissa will be given by the average of the abscissas of the points that belong to it and similarly the ordinate. Finally, identified the new centroid, the `cx` and `cy` fields of the `Cluster()` structure are updated.

The K-Means loop stops when the centroids no longer change, we obviously use a small tolerance TOLL in the comparison between the new and old centroids.

2.4. Parallel implementation

The parallelization concerns the assignment of points to centroids (and so to clusters) and the computation of new centroids. After loading the points and the centroids from files, we divided the

list of points into chunks of points: each chunk is assigned to a specific thread. We decided to distribute evenly the points between all the threads to maintain a good balance on the workload.

The parallel implementation of K-means has been developed using the structures defined for the sequential implementation, with the addition of a global array containing clustering information (sum of x values, sum of y values and total number of points). Each thread accesses to the common structures `Points` and `Clusters` read only, avoiding any type of contention. Hence, each thread works only on his chunk and it uses a local array to store the values which are necessary to calculate new centroids subsequently, it maintains the sum of x values, the sum of y values and the number of points for each cluster, obviously computed on his chunk of points.

In particular, let n be the number of points and let N be the number of thread used, then:

1. The set of points is divided into $\frac{n}{N}$ chunks and each chunk is assigned to a thread;
2. Each thread assigns a cluster to each point of its chunk;
3. After the assignment phase, the sum of x values, the sum of y values and number of points of clusters are updated. This update is done by each thread independently on his chunk and the values are stored in their local array;
4. Threads return their local array, in which are stored values which are necessary to calculate the new centroids.

The main thread, through a join, receives the array from each thread, containing clustering information. The array values are used to update the global array.

This is the code scheme we used:

```
for (i = 0; i < NUM_PROCRESS; i++) {
    pthread_create(&threads[i], NULL,
            MakeCluster, (void *) headChunk);
    headChunk = assignChuncks(headChunk);
}
```

```
for (i = 0; i < NUM_PROCRESS; i++) {
   double **myClusterDescription;
   pthread_join(threads[i], (void *)
                  &myClusterDescription);
   mergeSolution(clusterDescription,
                  myClusterDescription);}
```

Despite the increase in memory usage, we preferred using $N + 1$ arrays containing clustering information (one array for each of the N threads created and one for the main thread) to avoid threads race conditions on shared variables.

In fact the use of shared variables would have caused non-deterministic behavior and therefore the necessity to insert synchronization mechanisms, increasing downtime.

The number of threads N is defined by the constant `NUM_PROCESS`, that is set before the execution of the parallel implementation. In the Section 3 we will analyze the parallel implementation of K-means with different values of N.

3. Analysis

3.1. Running and Testing

At the beginning to execute the parallel and sequential implementation, the path of the .csv files, containing points and initial centroids, has to be updated with the name of the file that will be considered. Moreover, to execute the parallel implementation, the number of threads (i.e. `NUM_PROCESS`) has to be set.

To check our sequential implementation (and so the parallel implementation) we executed the K-means on two datasets with 12 bidimensional points each. We have chosen a small data set, so that we could apply the K-means algorithm manually and then compare the solution found with the one returned in output by our C function.

The first test starts with the following initial centroids: $(0, 0)$, for the cluster 0, and $(4, 7)$, for the cluster 1; the points are stored in `testPoints.csv` and the initial centroids in `initialCentroids.csv`. The final centroids are the following: $(2.67, 2.67)$, for the cluster 0, and $(4.78, 4.66)$, for the cluster 1. The result of the clustering algorithm is displayed in Figure 1.

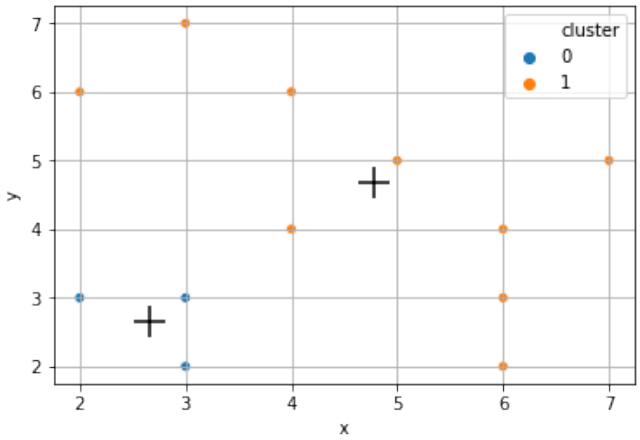


Figure 1: Clusters of the points in *testPoints.csv*

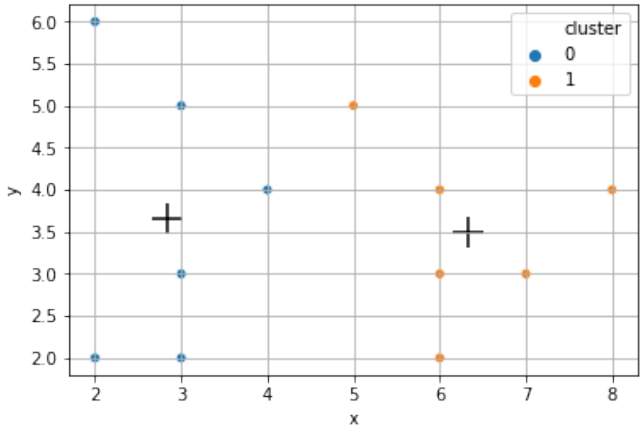


Figure 2: Clusters of the points in *testPoints2.csv*

The second test starts with the following initial centroids: $(4, 2)$, for the cluster 0, and $(4, 3)$, for the cluster 1; the points are stored in `testPoints2.csv` and the initial centroids in `initialCentroids2.csv`. The final centroids are the following: $(2.83, 3.67)$, for the cluster 0, and $(6.33, 3.50)$, for the cluster 1. The result of the clustering algorithm is displayed in Figure 2.

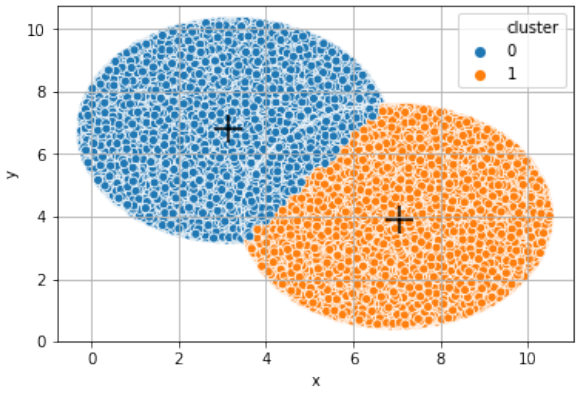
To make a more meaningful test, we applied the algorithm on three different datasets of 46244, 69366 and 125000 points respectively. Then we plotted the results using python, they are represented in Figure 3, in any case, the three clusters are correctly identified.

3.2. Performances and Speedup

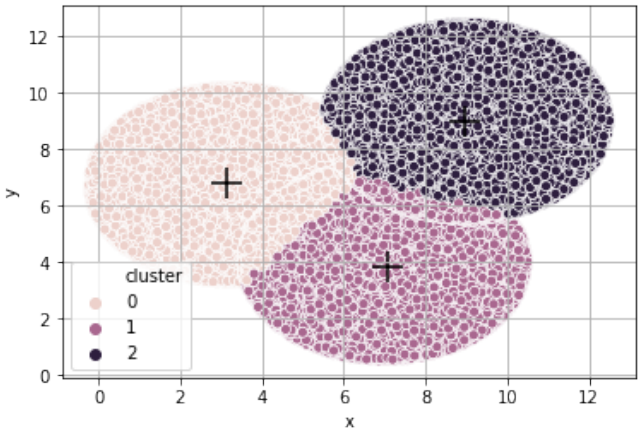
To measure the performances we used three different datasets of bidimensional points:

- the first dataset contains 46244 bidimensional points. 23122 points are distributed in a circle of radius 3.5 and centered in the point $(3.25, 6.75)$; the others points are distributed in a circle of radius 3.5 and centered in the point $(7, 4)$. The initial centroids are $(3.25, 6.75)$, for the cluster 0, and $(7, 4)$, for the cluster 1. The representation of the dataset is shown in Figure 3a, which also reports the clusters obtained by performing the parallel implementation of the K-Means with 10 threads;
- the next dataset contains 69366 bidimensional points. 23122 points are distributed in a circle of radius 3.5 and centered in the point $(3.25, 6.75)$; the others 23122 points are distributed in a circle of radius 3.5 and centered in the point $(7, 4)$; the others points are distributed in a circle of radius 3.5 and centered in the point $(9, 9)$. The initial centroids are $(3.25, 6.75)$, for the cluster 0, $(7, 4)$, for the cluster 1, and $(9, 9)$, for the cluster 2. The representation of the dataset is shown in Figure 3b, like before it reports the clusters obtained by performing the parallel implementation of the K-Means with 10 threads;
- the last dataset contains 125000 bidimensional points. 62500 points are distributed in the region delimited by points $(1, 4.5)$ and $(5.5, 9)$; the others are distributed in the region delimited by points $(4, 2)$ e $(10, 6)$. The initial centroids are $(2, 8)$, for the cluster 0, and $(9, 3)$, for the cluster 1. Like before, the Figure 3c shows the two-dimensional representation of the dataset and the clustering.

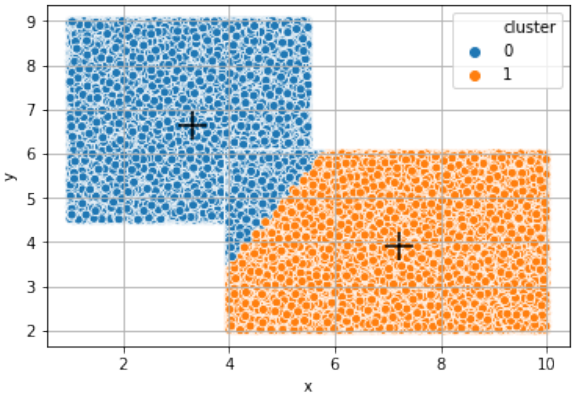
We run our K-Means implementations on a Travis virtual machine Mac OS X 10.13.0 with Xcode 10.1C version. We decided to use Travis builds to reduce the interference with other programs while executing the implementations. Ta-



(a) *First Dataset*



(b) *Second Dataset*



(c) *Third Dataset*

Figure 3: Representation of the datasets used to measure performance. Clustering was calculated using the parallel implementation of K-Means with 10 threads

		Number Of Threads					
		1	2	3	5	10	15
First Dataset	I/O time	8345704	8324868	7910628	8122830	8310987	8131703
	Execution time	32706	14078	24477	23069	14441	11815
Second Dataset	I/O time	18332727	18726619	18490108	18803942	16968374	14076609
	Execution time	141514	88655	114311	113884	57898	62228
Third Dataset	I/O time	47221836	46359689	46281179	46327845	46909715	46399187
	Execution time	140850	68348	80629	81360	71305	75301

Table 1: Times of the execution of parallel and sequential implementation on the two dataset.

ble 1 shows the builds results in *ns*. In the table, we report the I/O and Execution times obtained by performing K-Means implementations on the considered databases with different number of threads: 1 (sequential), 2, 3, 5, 10, 15.

We observe that implementations spend a lot of time in I/O operations, this is due to the fact that we have decided to read the data from file and it's very expensive. However, after all, this is what generally happens in the data architectures where

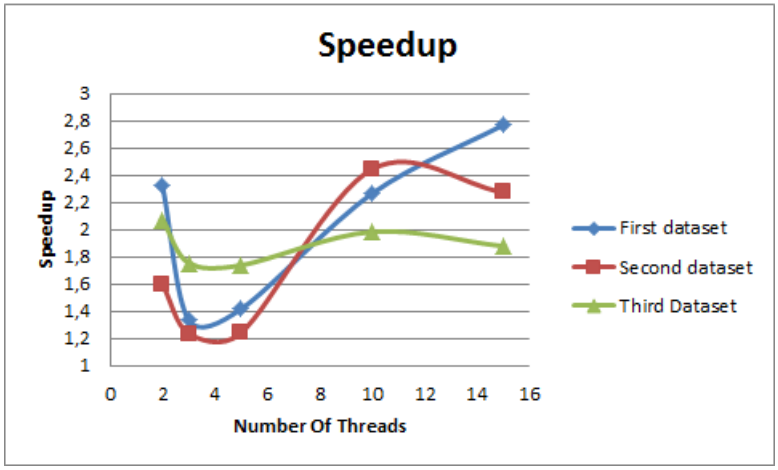


Figure 4: Speedup

data are stored in database or in big file.

The first and second datasets are very similar in terms of execution times, in fact they both have hundreds of thousands of points and form globular clusters, the third, on the other hand, is a little different because of how it was built.

However it can be noted that the executions of K-Means on the first dataset are little faster than those on the second dataset, since the initial centroids in the latter case are very close, so many iterations are needed to find the optimal clustering. In particular, comparing the execution times on the second dataset with those obtained for the first dataset, a decrease in execution times between 20% and 27% is observed for the second dataset.

In order to compare the performance of the parallel implementation with the sequential one, we calculated the speedup using the following formula (see [1]):

$$S_p = \frac{t_1}{t_P}$$

where P is the number of processors, t_1 is the completion time of sequential implementation and t_P is the completion time of parallel implementation with P processors. We have a good speedup if the speedup is equal to the number of core. Each Travis build has 2 cores available.

In the Figure 4 the speedup trend for the considered datasets is shown. It can be noted that the speedup trend in the three datasets considered (when the number of points varies) is very similar.

In particular that speedup has a peak when the number of threads is close to two (the number of cores), then it decreases until the number of threads is about five, after the curve rises and it reaches the maximum when the number of threads is around 12. Finally, the curve decreases with the increasing of the number of threads.

This trend is probably due to the fact that a high number of threads has to be scheduled by the operating system using only two cores, in this case threads don't work parallel, but most threads work concurrently. In particular more threads are swapped in and out by the operating system on a single core processor. Indeed we get a good speedup when the number of threads is near to two.

For example, let consider the case with 10 threads, we use a Travis virtual machine with only two cores then the operating system will assign each thread to a core based on its scheduling policies. Suppose that 5 threads are assigned to the first core, we indicate these with $v_1, v_2, \ldots, v_5$, and the others to the second core, we indicate these with $w_1, w_2, \ldots, w_5$. Each pair of threads (v_i, w_j) are actually executed in parallel, considering instead each pair of threads (v_i, v_j) or (w_i, w_j), their execution time depends beyond the execution time of threads on the same core, also

	Number Of Threads					
	1	2	3	5	10	15
I/O time	33546893	30722007	30931188	30955249	32302254	30726587
Execution time	41141	18657	14808	11518	9831	10506
Speedup	1.0000	2.2051	2.7783	3.5719	4.1848	3.9160

Table 2: Times of the execution of parallel and sequential implementation on 8 cores virtual machine.

from the thread scheduling time.

Looking at Table 1 we note that the maximum speedup for the third dataset (with more points) is reached for 2 threads (which is equal to the number of cores) and its value is 2.06 while, for the first and second dataset it 's 2.86 and 2.44; since this dataset has an ugly initial centroid configuration, as above explained.

For a more complete view of K-means performance we have also run our algorithm on a virtual machine with 8 cores on AWS, using the third dataset and the results are presented in Table 2.

Looking at Figure 5, in which the speedup trend is shown, we can observe that the speedup trend reflects that of the test done with two cores; in fact it correctly grows until the number of threads is approximately equal to the number of cores (8) and then it starts to decrease because the parallel execution is flanked by the concurrent one.

In particular the maximum speedup is reached for 10 threads and its value is 4.1848. We have also computed the *efficiency* (see [1]) of our algorithm using eight processors and we have found:

$$efficiency = \frac{speedup_8}{8} \approx \frac{4.1848}{8} = 0.5231$$

which is close enough to 1, so the eight processors are well-utilized in solving the problem, in comparison to the effort that is lost in idling and in communication.

We have also tried to change the number of centroids to test our algorithm, in Figure 6 we have the resulting trend. In particular, fixed the number of threads to two (the number of cores utilized),

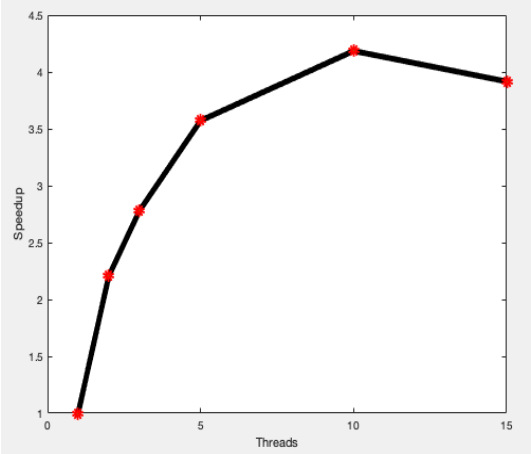


Figure 5: Speedup with 8 cores virtual machine: it changes with increasing number of cores.

we can observe that as the number of centroids increases, also the execution time increases. In fact, as the number of clusters increases, both the number of structures to be managed and the number of operations to be performed increase.

Luckily our K-means algorithm implementation doesn't need to synchronize different threads, because they work only on local variables.

It can also be noted that datasets to be considered must have a rather high number of points, otherwise the cost of creating and destroying threads would be higher than the cost of executing the sequential algorithm. This also happens because the operations carried out by the K-Means are essentially sums and subtractions, at most divisions for the recalculation of the centroids, but in a smaller number, so they are carried out very quickly by the calculator. So the parallelization of K-means makes sense only for large amounts of data: Big Data, which are widespread today.

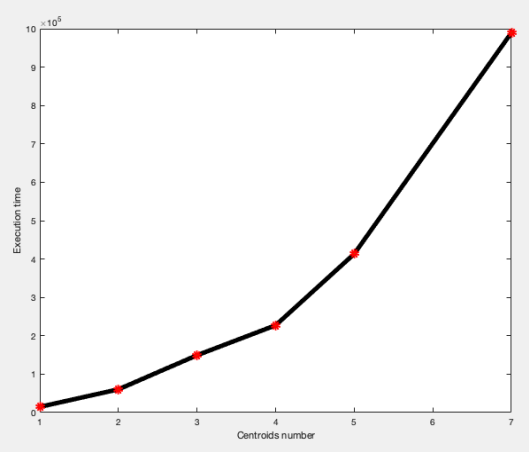


Figure 6: Execution time as the number of centroids varies

4. Conclusions

In conclusion it can be stated that K-means is a highly parallelizable algorithm, indeed its parallelization is indispensable with large data sets. In this project we have proposed a C implementation in which we used the Pthreads library. The performances we have obtained are good, the speedup has a peak when the number of treads is equal to the number of available cores, then it depends on the scheduling policies of the operating system, since the execution of the algorithm will no longer be only parallel but will become concurrent. The available processors are well used for the resolution of the problem, in fact we have obtained an efficiency equal to 0.52, there are no latency times due to synchronization and communication between the various threads, they work independently on their own subset of data and exclusively on local variables (it's an embarrassingly parallel problem). In particular we have parallelized decomposing data by blocks, because we had to manipulate a lot of data, and similar operations had to be applied on the same data. The association between tasks and data chunks takes place *statically*, created blocks are all of the same size and in number equal to the number of threads. We have made this choice to maintain simplicity, in fact complex data compositions would have been more difficult to manage, and also to ensure efficiency and a correct load balance between threads. The safety and liveness properties are guaranteed.

References

[1] C. L. Lawrence Snyder. *Principles of Parallel Programming*. Pearson, 2009.

[2] V. K. P.-N. Tan, M. Steinbach. *Introduction to Data Mining*. Pearson, 2006.

[3] G. R. Thomas Dauber. *Parallel Programming - for Multicore and Cluster Systems*. Springer, 2010.