Architectures and Platforms for Artificial Intelligence

Project Report

Alessandro Stockman ⋄ alessandro.stockman@studio.unibo.it

June 2022

1 Introduction

This work discusses a `C++` implementation of the **K-Means** algorithm alongside some minor variations of it, applied both in a sequential and parallel paradigm. Parallelism was achieved using the `OpenMP` API.

In order to evaluate the performances of the developed solution various tests were executed on different synthetic datasets with various parameters, testing both the execution times of the discussed algorithm and its variants and their parallel implementation.
The implemented variations to K-Means are **K-Means++** and K-Medians.

2 Algorithm

The **K-Means** clustering problem aims at partitioning a dataset made of n data points into k different **clusters**, with k being a given integer number. Each data point must belong to the cluster with the nearest mean (also called cluster **centroid**). This clustering minimizes the within-cluster variances.
The problem is computationally **NP-Hard**, but usually, heuristic algorithms are used to converge quickly to local optima using an iterative refinement approach.
The most common implementation, also called **Lloyd's algorithm**, proceeds with the following steps:

- Clusters Initialization: K centroids are randomly chosen between all the data points, those will be the initial clusters, containing only their centroid.
- Iterative repetition of the following two steps until either a max iterations limit is reached or none of the centroids gets updated during an iteration:
  - Clusters Update: For each point of the dataset, the distance between every cluster's centroid is computed, the current point gets assigned to the cluster having the minimum distance.
  - Centroids Update: Each centroid gets updated by computing the mean of each point inside the cluster.

3 Variants

3.1 K-Medians

A major problem in standard K-Means revolves around its sensibility towards outliers and noisy data.
A simple variation which addresses this point is **K-Medians**, which, instead of computing the mean of each cluster to determine its centroid, calculates its median. The obtained effect is the error minimization over all clusters with respect to the 1-norm distance metric, as opposed to the squared 2-norm distance metric used in vanilla K-Means.

3.2 K-Means++

Another drawback arises when a random initialization provides poor clusters, since the proposed algorithm doesn't achieve optimality, but merely an approximation of it.
K-Means++ is an algorithm for choosing the initial values for the centroids in the standard K-Means algorithm.
The steps are the following:

- One center is randomly chosen among the data points.
- The following steps are repeated until k centers have been chosen:
  - For each data point x not chosen yet, the distance $D(x)$ between x and the nearest center that has already been chosen is computed.
  - A new data point is chosen at random as a new center, using a weighted probability distribution where a point x is chosen with probability proportional to $D(x)^2$.
- Then, standard K-means clustering is executed.

4 Implementation and Parallelization

The proposed solution is built to take as input a CSV file containing the data points and its column indexes to be considered for the clustering of its rows, working thus with an arbitrary number of features. The output consists of two separate CSV files:

- Clusters Output: Contains a row for each data point in the input file, stating its features considered for the clustering and the assigned cluster's index.
- Centroids Output: Contains k rows, having the ith row stating the ith cluster's centroid.

Parallelism has been implemented using different techniques in the various sections of the algorithm. Definitely, the most critical operations are the **Clusters Update**, where each point of the dataset is assigned to the correct cluster, and the **Centroids Update**, namely, the recomputation of each cluster's centroid:

- Due to the **embarassingly parallel** nature of the **Clusters Update** operation, since it consists in two nested loops (one through all the datapoints and the other through the different clusters, in order to identify which one the data point should belong to), it was simply implemented using the `omp parallel for` directive.

- About the **Centroids Update** task, the implementation differs between the **Standard K-Means** and the K-Medians variant.

  In the first case, it was implemented following the **reduce** pattern, applying an array sum-reduction in order to obtain the total sum for every feature in every cluster, which would then be serially divided by the size of each cluster to get its mean. Through the `omp parallel for reduction(+:cumulatives_f[:k]) reduction(+:sizes[:k])` directive it was possible to apply the reduction to array variables, allowing an arbitrary number of clusters.

  On the other hand, the computation of the medians was achieved by creating a temporary array, containing all the data points divided by clusters and then ordered. The ordering task was again parallelized using the `omp parallel for` directive.

5 Evaluation

5.1 Datasets

In order to experiment both with computational performances and algorithmic effectiveness, three different synthetic dataset were generated and used:

- **Dataset A**: 500.000 records with 2 features drawn from a uniform distribution ranging in the $(0, 1]$ interval.
- **Dataset B**: 8.000 records with 300 features drawn from a uniform distribution ranging in the $(0, 1]$ interval
- **Dataset C**: Samples with 2 features drawn from different Gaussian distributions to achieve interesting patterns for the comparison of the variants.
  - 1.000 samples with $\mu = (5, 5), \sigma = (1, 1)$
  - 1.000 samples with $\mu = (1, 8), \sigma = (0.2, 0.2)$
  - 1.000 samples with $\mu = (8, 6), \sigma = (0.25, 2)$
  - 1.000 samples with $\mu = (4, 2), \sigma = (1, 0.25)$
  - 1.000 samples with $\mu = (4, 9), \sigma = (0.5, 0.5)$
  - 500 samples of uniform random noise $(0, 12]$.

5.2 Experimentation

The method chosen to test the implementation consists in 5 different runs for each combination of parameters, the timings of the single tasks in every execution was then averaged out and divided by the number of iteration, thus obtaining the time per iteration.
The serial version of the program was tested by running the same code with just one thread available, in order to avoid taking the parallelization overhead into account.
The timing of the program was collected by using the `omp_get_wtime` function.
The analysis consists in the collection of the timing of the single tasks: **Initialization**, **Clusters Update**, **Centroids Update** and the **Total** time given by their sum.
Everything was run on the DISI department HPC Cluster.

6 Results

By looking at the results of the algorithm over the plottable datasets, it's clear how few differences are notable over the uniformly distributed **Dataset A**.

On the other hand, in **Dataset C**, **K-Means++** achieves the identification of smaller a cluster, probably because of the better initialization, compared to the standard algorithm.

K-Medians also recognizes easily the elongated cluster, showing a better handling of outliers.

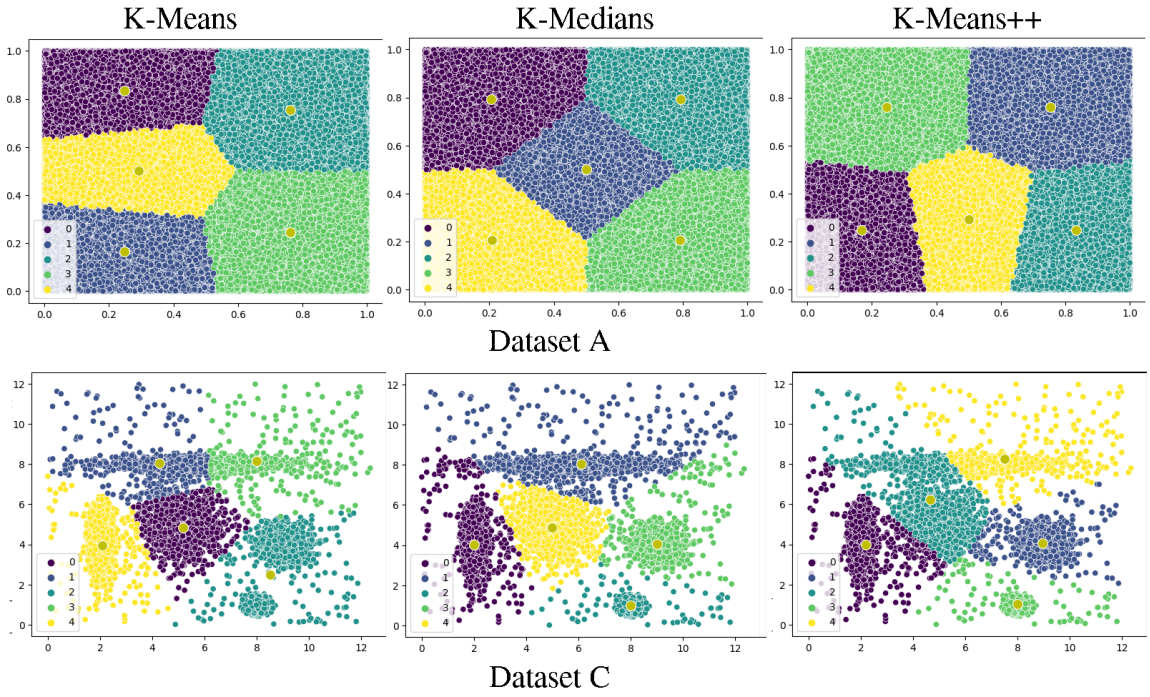

Figure 1: Plots of the results on Datasets A and C with different algorithm variants

	Variant	Threads	Task (ms)			
			Initialization	Update Clusters	Update Centroids	Total
Dataset A	K-Means	1	0.008	115.930	11.388	127.326
		2	0.009	59.815	6.173	65.997
		4	0.009	44.000	5.271	49.280
	K-Medians	1	0.008	116.051	184.585	300.645
		2	0.008	60.050	103.679	163.737
		4	0.008	44.855	77.521	122.384
	K-Means++	1	665.514	115.800	11.398	792.712
		2	620.079	59.871	6.241	686.190
		4	576.770	44.554	5.286	626.611
Dataset B	K-Means	1	0.032	88.415	55.012	143.459
		2	0.035	46.490	25.351	71.876
		4	0.039	39.499	19.803	59.341
	K-Medians	1	0.032	86.071	303.181	389.284
		2	0.032	44.792	199.762	244.586
		4	0.032	40.694	149.891	190.617
	K-Means++	1	466.533	87.008	53.054	606.595
		2	440.385	45.449	24.725	510.559
		4	432.184	40.100	19.792	492.076
Dataset C	K-Means	1	0.005	0.940	0.083	1.027
		2	0.004	0.703	0.066	0.773
		4	0.004	0.389	0.041	0.435
	K-Medians	1	0.005	1.028	1.062	2.096
		2	0.005	0.594	0.732	1.331
		4	0.004	0.401	0.572	0.976
	K-Means++	1	4.587	0.889	0.077	5.552
		2	4.442	0.682	0.065	5.189
		4	4.124	0.382	0.038	4.544

Table 1: Average performances per iteration over the three datasets

From the collected data, it's visible how the computational time is dominated by the **Update Clusters** task in the standard **K-Means** implementation, while in K-Medians, Update Centroids also gives great contribution to the total time. On the other hand, **Initialization**, which runs only once for every execution, takes a relevant amount of time only in **K-Means++**, contributing greatly in the final computational time in small datasets where few iterations of the algorithm is needed.
The scalability This results in enjoying the most speedup with the threads number scaling, as it implements parallelism using the pattern.

As expected, the scaling in the number of available threads affects positively the computation time. By comparing the execution with different numbers of threads, the speedup is around a factor of 2 when comparing the single thread execution and the 2 threads execution. A further speedup of roughly 1.5 is achieved when comparing the 2 and 4 threads executions.