# Kmeans with OpenMP

**Pozzoli Davide**

**GitHub:**
https://github.com/treyvian/Kmeans-with-OpenMP

module 2 report for
Architectures for Artificial Intelligence

Artificial Intelligence
University of Bologna
Italy

# Contents


**1 Abstract** **2**

**2 Clustering Algorithms** **2**
- 2.1 K-means . . . 2
- 2.2 k-medoids . . . 2
- 2.3 silhouette score . . . 2

**3 Datasets** **3**
- 3.1 Mall Customer . . . 3
- 3.2 California Housing Prices . . . 3

**4 Project Structure** **3**

**5 Evaluation and Results** **4**

**6 K-means** **5**

**7 K-medoids** **5**

**8 Silhouette score** **6**

# 1 Abstract

The task given was to implement the k-means algorithm and measure the speedup obtained thanks to the implementation of parallel programming patterns provided by the OpenMP API. In this project I propose the *k-means* algorithm, the *k-medoids* method, a variants of *k-means*, and the silhouette score as a way to evaluate the correctness of the clustering. I used two different datasets, the mall dataset which contains 200 rows and was used to verify the correctness of the implementations and the house one, with more than 20 thousands rows, which was used to measure the increment in speed brought by the parallelization.

# 2 Clustering Algorithms

## 2.1 K-means

The k-means is a method of vector quantization, that aims to partition $n$ observations into $k$ clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

In my implementation there is an external for loop with dependencies, so not parallelizable, and two inner loop which are parallelized separately, the first one is used to calculate the distance (using the euclidian metric) of each point from the current centroids and the other one updates the centroids value. The outer loop stops when the max number of iterations is reached.

## 2.2 k-medoids

The k-medoids method is a clustering algorithm similar to k-means. In contrast to k-means, k-medoids chooses actual data points as centers (medoids). k-medoids can be used with arbitrary dissimilarity measures, whereas k-means generally requires Euclidean distance for efficient solutions. Because k-medoids minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances, it is more robust to noise and outliers than k-means.

My implementation does not use any strategies for the optimization of the search, it's a naive implementation recomputing the entire cost function every time and having a runtime complexity of $O(n^2k^2)$. This high runtime complexity can be observed in the California dataset with the high number of entries. In this case I used the Manhattan distance to calculate the total cost.

## 2.3 silhouette score

Silhouette refers to a method of interpretation and validation of consistency within clusters of data.

The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from

$-1$ to $+1$, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters.

The silhouette can be calculated with any distance metric, such as the Euclidean distance (the one I chose) or the Manhattan distance,

# 3 Datasets

I've used two dataset for this project, both taken from keggle.

## 3.1 Mall Customer

The Mall Customer is created only for the learning purpose of the customer segmentation concepts, also known as market basket analysis.

This is a quiet small dataset containing 200 rows and 5 columns. I only use the last two dimensions/columns of the dataset("Annual income","spending score") since they are well divided and easy to visualize. This allowed me to check the correctness of my implementations thanks to the immediate feedback from the plots.

The number of clusters which gives the highest silhouette score is 5 which can be easily confirmed by looking at the data distribution.

## 3.2 California Housing Prices

The California Housing Prices dataset contains information from the 1990 California census.

This dataset is bigger with respect to the other one, it contains more than 20k rows and 9 columns. In this case I took 3 columns("Longitude","Latitude","Median Income") to further test my implementations. While the k-means function is not bothered by the increased size of the dataset, the k-medoids has an hard time with its $O(n^2k^2)$ runtime complexity.

Based on the silhouette score the best number of clusters is 2, but all the notebooks that I found on Keggle use the 6 as number of clusters. In this case the fixed number of cluster is preferable also to avoid waiting around for the k-medoids algorithm to finish all the possible combinations.

The csv file in the *data* folder of the project corresponding to this dataset, has been preprocessed by removing all the columns that were not necessary leaving only the three that we interested in.

# 4 Project Structure

- **data**: contains the 2 data sets in csv format
- **output**: contains the the csv files created after the execution of the clustering algorithms. It does also contain two python files that allow to have a graphical representation of the clustered data sets.

- **results**: contains the data and the graphs for the strong and weak scaling efficiency.
- **src**: contains all the code of the project:
  - rw-csw: contains two helps method for reading and writing csv files
  - silhouette-score: contains the implementation of the silhouette method
  - kmeans: contains the implementation of the k-means method
  - kmedoids: contains the implementation of the k-medoids method
  - mall-costumer-dataset: contains the main method for the execution of the clustering algorithms with the Mall_costumer.csv dataset.
  - hosue-dataset: contains the main method for the execution of the clustering algorithms with the house_pre.csv dataset.
- **project.sbatch**: to execute the project on the hpc cluster
- **launch_house**: file bash launched by the project.sbath, it execute the clustering algorithm on the house dataset. It will execute first the version for measuring the strong efficiency performances and then the version with increasing work load in input for the weak efficiency performances.
- **launch_mall**: bash file to execute the clustering algorithms on the mall customer dataset.

# 5 Evaluation and Results

The data gathered that is found in the folder *results* is the output of an execution on the hpc cluster slurm.cs.unibo.it setting *–ntasks-per-node* equals to 4 as can be seen in the *project.sbatch* file. The *OMP_NUM_THREADS* variable in the bash file is set to reach at most 8 threads even though only 4 are available on hardware, this creates a situation that can be easily seen in the results, where past the 4 threads line the performance does not improve but actually worsen.

The number of rows for the housing dataset is 20641, but in order to measure weak scaling efficiency I took the number 20000 which divided by 8, max number of threads tested, gives 2500 and used it as baseline. The script increases the amount of points read from the csv file by 2500 for every core added.

# 6 K-means

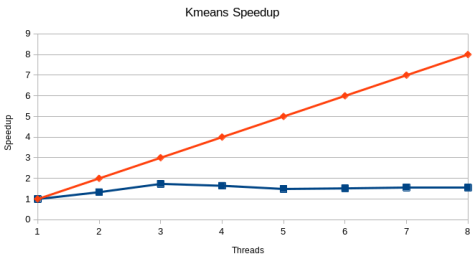


Figure 1: Strong Scaling

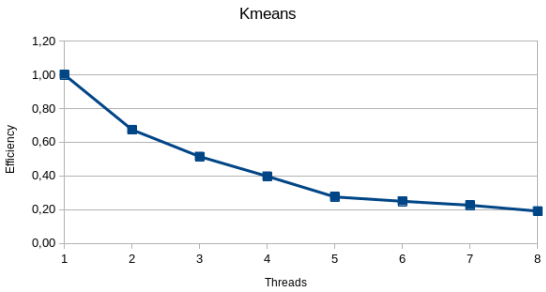


Figure 2: Weak scaling

Between the three implemented algorithms k-means is the one that has the worst results when parallelization is introduced. The problem is connected to the fact that the outer loop, that counts the number of iterations, can not parallelized since the $n^{th}$ execution depends on the results (value of the centroids) of the $n^{nt}-1$ iteration. Even though the body of the loop is completely parallelized and the overhead of the creation of the new threads is minimized by initializing the parallel region before the loop begins, the performances are still being limited by the fixed number of epochs that the algorithm has to go through each time.

# 7 K-medoids

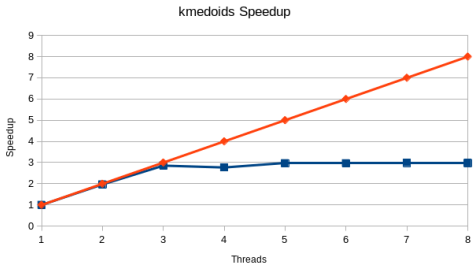


Figure 3: Strong Scaling

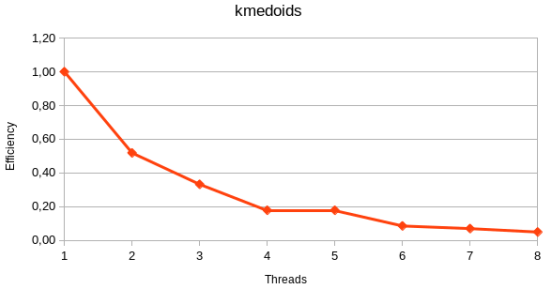


Figure 4: Weak scaling

K-medoids is the algorithm that most benefits from parallelization. Due to the high number of rows in the input data the serial version takes over 3 minutes to complete, while by just adding an other core the times took is almost cut in half. Inside the parallel region there are some critical blocks that protect the

shared variables necessary in order to avoid random results but with an effect of course on performances.

As we can also see from the graphs and the results with 3 cores a speed up of 2,85 can be obtained, while the performance with 4 cores drastically decreases. This behaviour, with the $4^{th}$ core not performing as well as the previous ones, I think is to be attributed to the inability for the cluster to find time and resources for other processes that are running in background.

# 8 Silhouette score

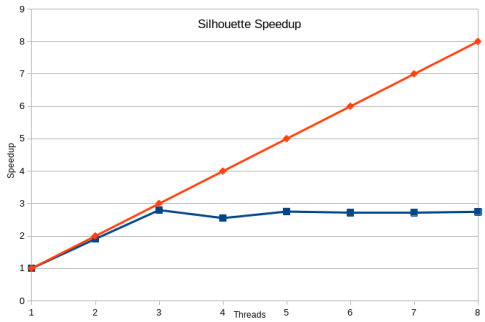


Figure 5: Strong Scaling

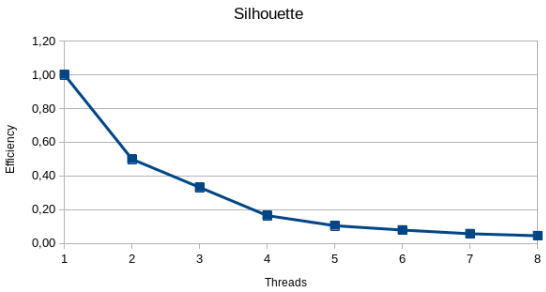


Figure 6: Weak scaling

The implementation of the silhouette score is the ones that has the minimum amount of non parallelizable portion, the only thing that is always executed sequentially is the initial part where all the variables and arrays are initialized. The rest is inside a parallel for, which contains only private variables, so no critical or atomic sections needed. The only variable shared is the *silhouette_score* but the *reduction* clause takes care of all the problems connected with concurrent access.