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

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 best cluster number. 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 I use the euclidean distance to calculate the distance between points and centroids.

2.2 Parallel K-means

In my implementation I kept the outer while loop of the sequential version and divided it's body in two method to improve code readability. The first method *compute_distance* find and assign the nearest centoroid to the each point, this is the method that I decided to parallelize. The second method *update_clusters* updates the values of the centroids based on the results of the previous one. I decided to keep it sequential since based on my observation the time taken to execute this part is negligible w.r.t. the previous one.

The idea behind the parallel version is to equally partition the number of points between each of the threads available when computing the distance.

In order to avoid mutual exclusion the two arrays *num_point_clusters* (Which keeps track of the number of points in each cluster) and *sum_points* (which keeps the sum of the the coordinates for each point in each cluster) have been modified. The first dimension of each of the two arrays has been increased in order to give to each thread it's own element in which they can write without interfering with the others.

2.3 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 run-time complexity of $O(n^2k^2)$. This high run-time 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.

2.4 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 $O(n^2k^2)$ run-time 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.
  - house-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.

5.1 K-means

Figure 1: Strong Scaling

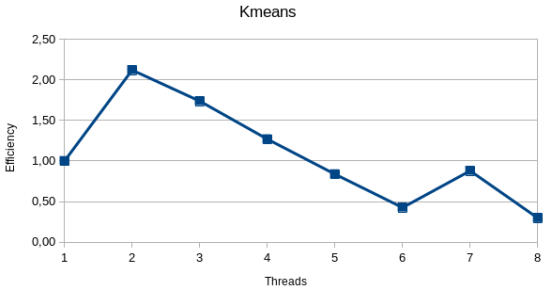

Figure 2: Weak scaling

Based on the strong scaling graph we can see an improvement with the increasing number of threads. The speedup gets as high as a 2.4× with 4 threads. These results are not as exiting as the ones obtained with the others algorithms but I think this is the best I can obtain with this particular implementation.

The weak scaling graph has a peculiar shape, it sometimes (with 2 and 7 threads) gets a speedup even though the amount of data has increased. I have no idea what it is causing it since this behaviour is not reflected in the Strong scaling analysis.

5.2 K-medoids

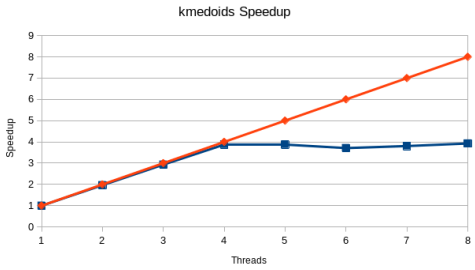

Figure 3: Strong Scaling

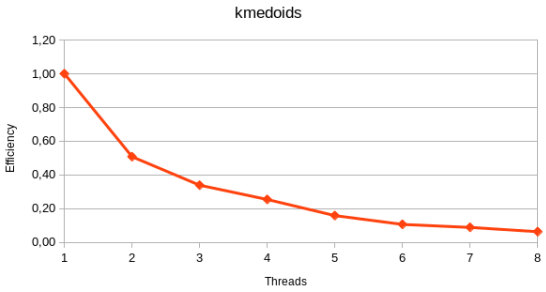

Figure 4: Weak scaling

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 almost 3 minutes to complete, while by just adding an other core the times took is 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 observe from the graph the speed up increases until the 4^{th} then hits the hardware limit and then the performances start to decrease a little. This behavior can be seen also in the other methods.

5.3 Silhouette score

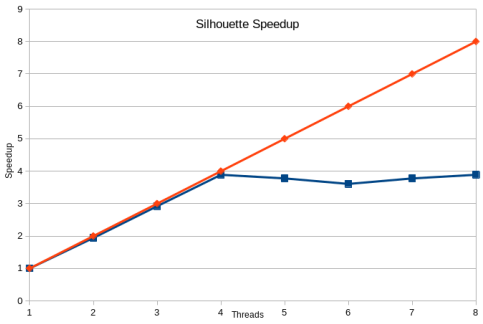

Figure 5: Strong Scaling

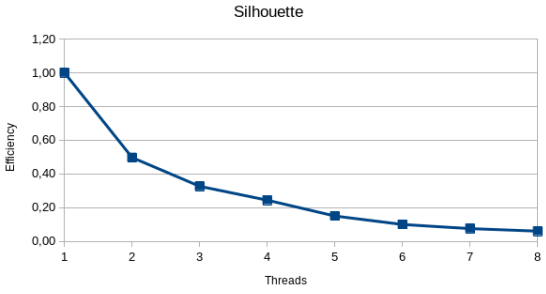

Figure 6: Weak scaling

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 almost 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.