PC24/25: Parallel k-means clustering in openMP

Lorenzo Pasquini
E-mail address
`lorenzo.pasquini3@edu.unifi.it`

Abstract

The K-means is a popular clustering algorithm that aims to partition n observations in k clusters, as such, this algorithm can be parallelized quite well with some dependencies to consider, but quite easily resolved. This implementation uses random positions for both the centroids of the clusters and the points that represent the observations. It was chosen to operate on a 3D plane and so both the points and the centroid have 3 that have to be managed.

1. Introduction

The K-means clustering algorithm aims to find the best clusters in which certain data (in this case points) belong. A centroid is the mean of a cluster and as such is prone to variations, for instance if a point is added to the cluster, the centroid will inevitably move. The points tend to be assigned to the closest centroid they can find and, if a centroid moves, then the point must check again where is the closest centroid. Thus, for a set of points P in the 3dimensional space, and for a set of centroids C we have to:

$$P_j = p_i | min(dist(x_i, c_j)) \tag{1}$$

where the dist is the euclidean distance and $\mathrm{P_j}$ is the set of points belonging to the cluster j, then the centroids are updated and placed in the mean position between the point assigned to them

$$c_j = \frac{1}{|Pj|} \sum_{x_i \in Pj} x_i \tag{2}$$

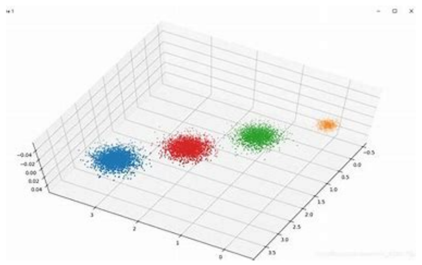

Figure 1. A rapresentation of a K-means algorithm in 3D

1.1. Parallelization choices

The k-means algorithm is renown and as such has many variations to choose from, but the basics can be summed up by two choices, to optimize the centroids' placement or to optimize the points' speed of assignment. In this implementation the priority will be to create the clusters as fast as possible, and thus to speed up the assignment of the points to the clusters, this choice is made because each assignment to a centroid is an independent action for each point, so there are no dependencies and the threads need no synchronization which would slow down the program. Another reason is the load of data that is put on the thread: while when calculating the point's distance to the centroid the thread does a pretty heavy task but every thread has the same load balance, centroids on the other hand can have more or less points assigned to them and so it would be more difficult (if not impossible) to have an equal load balanced between threads.

2. Implementation

In this section will be explained how the basic algorithm was written into C++ and openMP. The idea behind the code is to optimize as possible the basic algorithm that was introduced earlier. The first and most important objective in this code was to create a single parallel section, since the code ha many loops and the algorithm itself is encased in a while statement, opening and closing too many parallel sections would generate overhead and slow down the program without reason.

2.1. Points and centroids

One of the fundamental parts of k-means is good representation of both the points and the centroids as this is the first step to have a good parallelization. First of all the points and clusters are defined as structs with their coordinates as data and for the points the centroid (as a number) which they belong to in that moment, and for the centroid the number of point that are assigned to it in a certain moment. These structs are then saved into arrays of points and arrays of centroid and their coordinates, since the algorithm proposed doesn't search for optimal positioning for the centroid, are then generated randomly. The generation of the coordinates is done by a mersenne twister generator, this to make sure that, even when working with a big number of points and centroids the numbers used will be actually random and not to risk having too many points in the same location.

2.2. Safeness in randomness

The first idea that comes to mind when talking about random numbers (especially hundred of thousands of them) is inevitably the risk of nondeterminism and, how is it guaranteed that the program will always eventually stop in an acceptable time, and to answer this, it is needed to prove that for each point and centroid location the k-means stop, or at least this implementation does. The point of this implementation, as suggested earlier, is that is completely surrounded by a *while* clause that every loop checks if there were any changes in the assignment of the points, if there were, the algorithm runs again, if there aren't the program ends, this somehow gives the idea, to stop the program there must be an instance of the loop where the points don't change their *assignment* variable (matter of fact, the k-means does not guarantees to find the global minimum of the distance between the points and the centroid of their cluster, it only guarantees to find a local minimum). The choice to surround the code in a while segment was made to have the possibility to parallelize the internal fors without needing to open a new parallel section each time.

2.3. Assignment and mean

When the code enters the while there are two for, the first one cycles between each point, resetting the minimum distance and the *temp* value which is needed later, the second for cycles between every centroid, searching for the minimum distance and, when found, uses temp to know which centroid to assign the point we are currently assigning, then in a local copy of the array of centroids are added to the found centroid the coordinates of the point and the number of points is increased, this is made in a local copy to prevent the need to synchronize now the threads, which would cause severe overhead since all the threads are computing data and accessing memory. In fact, if changes do happen, the local array of centroids for each thread is then saved in a common one using atomic update to prevent collisions, a barrier is used to guarantee that the common array is effectively usable and then the mean in calculated, another barrier is used to be certain that every thread has completed the computation of the mean and then the cycle repeats until no changes are made in the points' assignments.

3. Execution

This section is dedicated to the analysis of the execution times and how the difference between threads, centroids or points can affect the code. The writing and testing of the code were made on a modified version of DevC++ release on 2021 based on the latest version of the discontinued 2016 version. Windows 11 OS on a laptop with a 13th Gen Intel core I7-1360P processor running 4 performance cores and 8 efficient cores and a 16 thread limit and C++ 14 is used.

3.1. Analyzing data

In image 2 is shown the different speedup based on the different datasets, while for bigger numbers of centroids the speedup seems to decline (as seen also in the figure 5, that studies the case with 100k points), it seems also that the speedup stands somewhat constant independently by the number of points (also seen in image 4), this goes to confirm the fact that the program was optimized to work better for a big number of points. The time of execution as seen in the figure 3 seems instead to decrease in a quite orderly manner, obviously in a coherent way with how the speedup increases or decreases. Notable the fact that usually thread 4 and further multipliers of 4 seem to have a better speedup, probably caused by the machine which, having 4 performance cores, optimizes calculations to use them at their fullest. Also interesting to see how the speedup decreases drastically when over 10 threads and 50 centroids, while that could be some sort of noise, the cause could be the machine itself or, perharps most probably, the optimization that was focused on parallelizing points' assignment more than centroids' median calculations.

3.2. Conclusions

While having to deal with the choice of maximizing efficiency for points may have made the code a little slow when too many centroids are given, the parallelization paid off even in a tridimensional environment, where distance calculation, one of the most used functions in the code, is particularly tedious. The speedup seems most dependent on the number of centroids having a sort of ”sweet spot” around 30 centroids. Points on the contrary don’t seem to really slow down the speedup, just slowing the time of execution when too many are created.

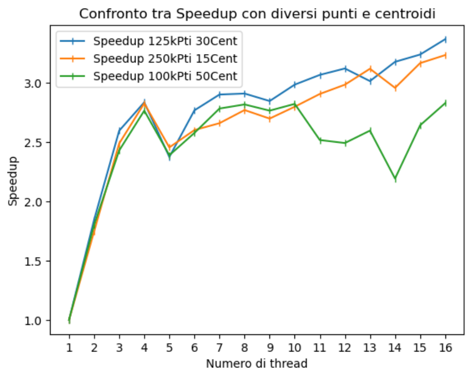

Figure 2. Speedup differences between various datasets

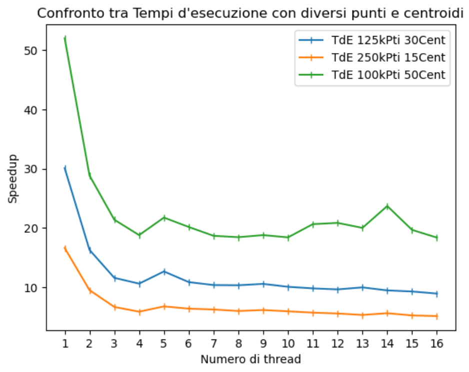

Figure 3. Execution time between datasets

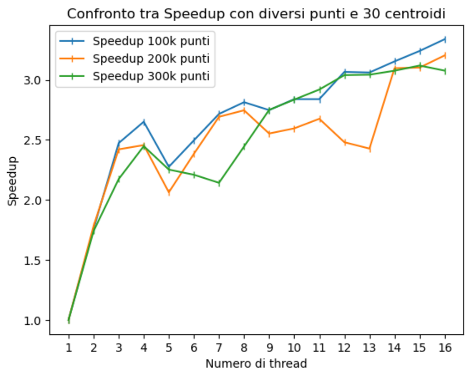

Figure 4. Speedup given 30 centroids

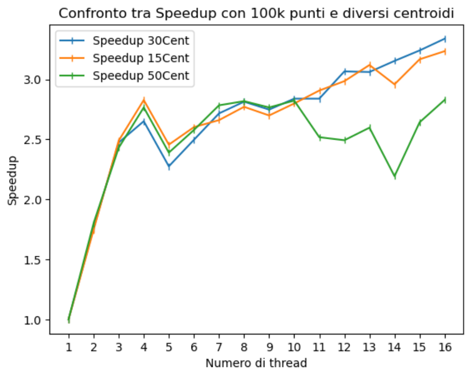

Figure 5. Speedup given 100k points