## Algorithm testing

What has been described up to this point allows to create a parallel kmeans clustering which exploit the Hamerly's algorithm. In this section we present the result some tests that have been done using the Lloyd, Hamerly and parallelized Hamerly algorithm to compare the performances.

## General testing methodology

For every of the following tests has been used the same procedure.
For each of the presented datasets, first a lloyd model is initiated and executed to set a reference time which is then used to measure the speedup of the hamerly algorithm with 1 to 10 threads.
The speedup is calculated as

$$speedup = 1 - \frac{hamerly\_time_T}{lloyd\_time} \quad (1)$$

where $hamerly\_time_T$ is the time taken by the hamerly algorithm with T threads.
For every model, a fixed number of iterations (100) has been used to stop the respective algorithm after the same number of updates.

## Randomly generated dataset

The first dataset is a randomly generated one which allows to easily study how the algorithm scales over N, K and D because it is simple to implement an algorithm which increments one of the three while keeping fixed the other 2.

### Points scaling

To scale over N, the testbench starts from a dataset with 3 centroids and a dimensionality of 2 while the number of points goes from 500 up to 50000. Figure 1 shows the testbench result.

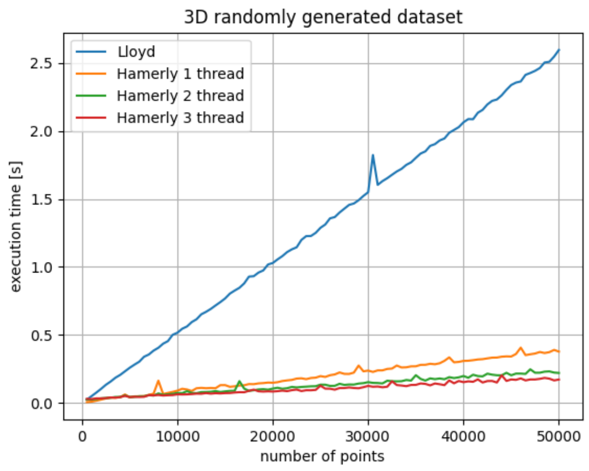

***Figure 1:*** *result of the test on the 3D randomly generated dataset scaling over N*
The result, for each model, is a straight line which can also be proven mathematically because if the number of centroids and the dimensionality are constants (fixed numbers) then, from equation **??** and **??**, it is possible to factor N out which results in a straight line equation (y = mx).

### Centroids scaling

The second testbench scales over K. Therefore, the testbench has a fixed number of points (3000) and dimension (3) while the centroids goes from 2 up to 100. Figure 2 shows the result of the testbench.

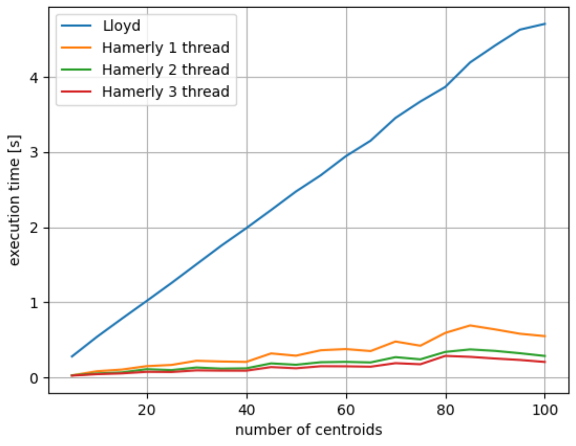

***Figure 2:*** *result of the test on the 3D randomly generated dataset scaling over K*
Since K can be factored out from equations **??** and **??** when N and D are fixed, we see a straight line patter for each model.

### Dimension scaling

Lastly, to study how the algorithm scales over D, this dataset has a fixed number of points (3000), a fixed number of centroids (3) while the dimensionality of the data goes from 2 up to 100. Figure 3 show the result of the testbench.

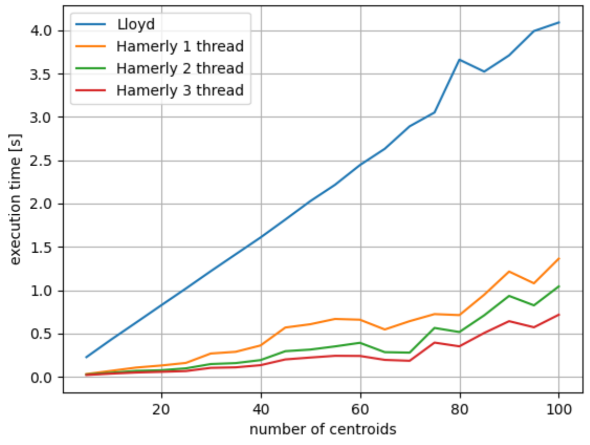

***Figure 3:*** *result of the test on the randomly generated dataset scaling over D*

### Speedup

To summarize the results, Figure 4 shows the speedup line for each of the previous graphs. In the latter it can be seen that the line corresponding to the D scaling has the lowest overall speedup but it is also the one where increasing the number of threads produces the most speedup, in other words, in this case, it is the one where using more threads is most effective.

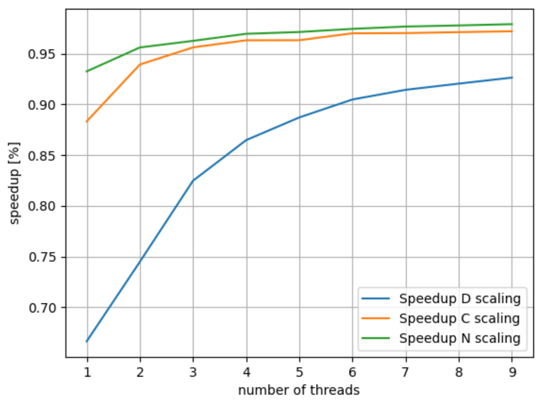

***Figure 4:*** *overall speedup on the random dataset*

## Wine quality dataset

To test the algorithm with a more realisitc dataset, we downloaded[a] a dataset which contains features about different wines which can be used to group them in classes representing their quality between 0 and 10.
The number of points in the dataset is 1143, with a dimensionality of 11. The centroids are 10 and were generated by randomly select them between the whole dataset.

This dataset has been tested by scaling the points N starting from 9 up to 1143. Figure 5 shows the result of the execution.

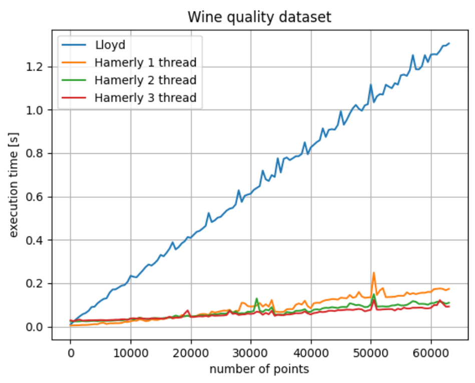

***Figure 5:*** *result of the test on wine dataset*

### Speedup

Figure 6 shows the speedup of the algorithm when scaling both for N and D.

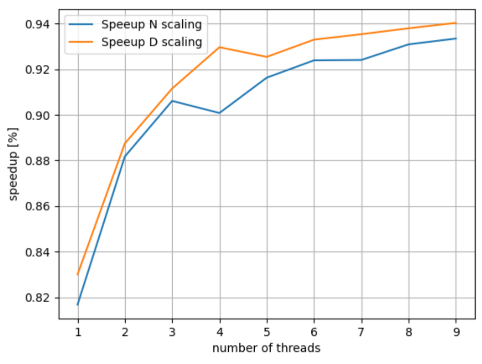

***Figure 6:*** *speedup on the wine testbench*

[a]https://www.kaggle.com/datasets/yasserh/wine-quality-dataset

This time, there is not much difference between scaling over D and N because both lines are similar. In this case, scaling over the dimension means to use less attributes to classify the wine quality.

## Speedup when the total workload is small

If the overall workload is too small, it may be that increasing the number of threads is no longer an improvement but may be counterproductive. This happens because by increasing the number of threads, it increases also the overhead need to handle them and therefore if the workload is sufficiently high this overhead can be neglected but if the load is low the overhead can affect the overall speed.

Talking about the workload, as a first approximation, it can be related to the quantity $N \cdot K \cdot d$.
To show this phenomena, we downloaded another dataset[a]. The latter is set of values taken by some high-frequency antennas with the pourpose of finding free electrons in the ionosphere. In other words, the goal is to distinguish between good measurement, where some electrons have been found, and bad measurement where there are no evident electrons. Therefore, the centroids are 2, while the dimension is 34 and the number of points is 351. The workload, in this case, is:

$$N \cdot K \cdot d = 351 \cdot 2 \cdot 34 = 23.869$$

which is much smaller compared to the previous examples. Figure 7 shows the speedup with respect to lloyd time using an increasing number of threads.

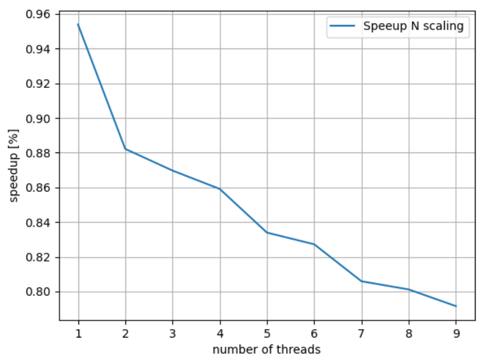

***Figure 7:*** *speedup on the ionosphere testbench*
The hamerly algorithm with 1 thread still be much faster than the lloyd version but, increasing the number of threads the speedup starts to become lower.

[a]https://archive.ics.uci.edu/dataset/52/ionosphere