# **APAI** Module 2

Giovanni Minelli (giovanni.minelli2@studio.unibo.it)

March 9, 2022

## 1 Introduction

The project is about the analysis, in empirical and practical terms, of the K-Means algorithm in a parallel programming context. It has been applied to Hyperspectral Imaging, an image analysis task which main goal is to identify important aspects of an image acquired at many wavelengths of the light spectrum. The algorithm segment the image by clustering pixels into classes based on the spectral similarity of each one with respect the other members of the class, considering it's representation at each band of the spectrum.
Each pixel can be considered as a vector defining a point in an N dimensional space, and pixels that have similar spectra will be located closer to each other in such N-space respect pixels that are dissimilar. Thus, distance in N-space can be used as a similarity measure for clustering. The clustering algorithms do so splitting the pixels into K clusters grouped by similarity. Each cluster is associated with a "center" value which is representative of (and close to) the pixels in that class. *Note that the RGB spectrum which we can sense is a subset of the hyperspectral one, and pixels which appear similar in the RGB image may not belong to the same clusters since they may differ significantly at other wavelengths.*
The clustering algorithms provide also a very reliable method for data compression.

## 2 Dataset

The hyperspectral imagery used to evaluate the parallel algorithm is a flight-line collected by the Airborne Visible Infrared Imaging Spectrometer (AVIRIS) sensor: an instrument that measures spectra for every location in the image. Data files (provided by NASA/JPL-Caltech) includes different information linked to different stages of data analysis. The three main levels of processing are Level 0 (raw data), Level 1 (calibrated radiances) and Level 2 (atmospherically-corrected surface reflectances) [1]. We will work specifically on the surface reflectance data obtained weightening with sensor parameters the raw data collected and removing effects of atmospheric scattering and gas absorptions. Have to be said that imagery may still have variations in illumination due to topography.
Reflectance is the proportion of the radiation reflected off a surface to the radiation striking it and is particularly relevant in hyperspectral data analysis, since materials can be identified by their reflectance spectra [2].
The data file we will use is a long binary file which express the values at each pixel position over all bands. Additional information like the format, the type of data and the light wavelength value associated to each band (expressed in nanonometers), can be found in the companion header file.

## 3 K-Means algorithm

The k-means clustering algorithm provides an iterative scheme that operates over a fixed number $K$ of clusters, while attempting to simultaneously optimize center locations, also called centroid points, and pixels assignments. For a choosen value $K$, the optimal partition assignment (considering the adopted starting point) is returned in a certain amount of iterations.
The algorithm starts assigning a centroid to each cluster following an implementation strategy that can vary:

- Selecting the first k pixels in the image or selecting them evenly along the diagonal of the image.
- Randomly selecting unique pixels from the image.
- Uniformly or randomly selecting means from the N-dimensional bounding box of the image pixels.

Then the iteration starts and a cluster is associated to each pixel of the image based on minimum distance between a given pixel $n_i, i \in [0, w*h[$ and the centre of that cluster $c_k, k \in [0, K[$. Then, the clusters centers are recomputed and the iterative process can start over. Assigning pixels and updating the centroids is repeated until a stopping criterion is met. For example, consider the fact that after some iterations fewer pixels will tend to be reassigned to different clusters, and therefore the centers will shift (migrate) less during successive iterations.

- A maximum number of iterations has been performed.
- Fewer than a threshold number of pixels are reassigned during an iteration.
- All means migrate less than a threshold distance during an update cycle.

One measure of the quality of a partition is the within-cluster variance; this is the sum of squared distances (Euclidean) from each pixel to the centre of their cluster. The optimal value for the number of partitions should minimize such value for each class.

# 4 Serial and Parallel Code

To understand the complexity of the code of the algorithm it is useful to first examine the serial implementation from which the parallel implementation is derived. We will see that the bulk of the computation for the algorithm is devoted to the pixel assignment and cluster centroid update phases.
To not introduce aleatory factors in the process, in both implementations the centroids are initialized by selecting evenly spaced pixels along the diagonal of the image used as input.

- To compute the distance between each pixel and the center of each cluster three nested cycles are needed (pixels, clusters and bands) with a comparison and an assignment at the second level of nesting and the operation to compute the distance inside the inner loop. Furthermore, an additional counter for the number of pixel reassigned in the current call is maintained in the outer loop.
- To compute the clusters's center, the mean of all pixels belonging to each cluster are calculated. That is done in a sum operation inside two loops (over all pixels and bands), and after have accumulated all values, one per cluster, a division operation has to be performed inside other two loops (over all clusters and bands).

A single k-means iteration will perform O(NBK) ops, with $N$ points, $B$ spectral bands and $K$ centers.
As a metric of distance I always used the Euclidian distance, which operation requires computing the square of a number. Considering a point $n_i$ and a cluster center $c_k$, the Euclidian distance is defined as following: $||n_i - c_k||^2 = \Sigma|n_{i_b} - c_{k_b}|^2$ where $b$ indexes the spectral components of each cluster.

For the parallel implementation I firstly evaluated the burden of computation to know a priori what operation optimize as a priority. Following the Dataset in use, can be evaluated the magnitude of the data: it is of order $O(10^1)$ for the number of clusters, $O(10^2)$ for the number of bands and $O(10^5)$ for the number of pixels, resulting in approximately $O(10^8)$ MAC operations for the pixels assignment phase and $O(10^7)$ additions + $O(10^3)$ divisions for centroids update.

## 4.1 assignObjects()

We have to compute the euclidean distance for each pixel on each dimension respect each centroid. Data has been stored in memory with the format HWC, in a way to have the same pixel coordinate represented in different spectral bands in subsequent memory cells. The code is composed of three loops not collapsable:

- The inner most compute the euclidean distance (sum of squared differences) between the current centroid and the bands of the current pixel. (x425 times)
- The middle loop select the centroid and at the end compare the current computed distance with the best found. (x10 times)
- The outer most, loops on each pixel assigning it the nearest centroid index and maintain a counter with the number of pixels reassigned. (x300k times)

To test the performance of the different possible implementations the snippet of code has been timed comparing the results of execution: parallelizing just the inner loop takes 200s per execution, 25s the middle and a few seconds per execution with the outer most. This is probably due to the slowdown of creating and merging new threads since the computationally expensive operation is just the squared difference inside all loops, which also have to access the data memory cells. Then, with the single outer loop parallelized, has been evaluated the difference between the use of `#pragma omp atomic` and the `reduction(+:numChanged)`, with `numChanged` as an accumulation variable possibly increased of 1 at each iteration: the results were equivalent.

## 4.2 computeCentroids()

The operations involved in the function are:

- The initialization of data objects to store the accumulations, which can be easily optimized by considering the different objects at the same time with parallel sections.
- Two nested loops to sum each pixel value at each band to the corresponding accumulation cell of it's centroid. That piece of code can be parallelized collapsing the loops, affecting the execution with a small decrease in performance (0.70s wrt a non parallel solution which takes 0.50s per execution), probably due to the overhead of a big quantity of threads to start and kill. Meanwhile a solution which execute in parallel only the outer loop slightly improves the time of execution (0.50s to 0.40s).
- A second snippet cycling over smaller data dimensions (clusters and bands) make a division storing the result in place. The iterations are fully independent and a parallel implementation with `collapse` of cycles let us save almost a second per call.

## 4.3 loadData()

Marginal from the execution of the algorithm, but still being a costly operation, the parallelization has been extended to the method used to store in memory the data read from the file. Considering a slice of data, a serial version takes around 3s for execution where (excluding the file reading) the most of the work reside in visiting the image array to copy the pixel values, and determine the normalization factors for a rescale to [0-255]). For a finer evaluation i considered the two visit independently to reunify them later with the knowledge of the single behaviours.
The first snippet composed of three loops fully collapsable was measured under different implementations decreasing the parallelization and increasing the work of a single thread (i.e. collapse all three loops, only two, or parallelizing the execution of only the outer one), but the performances seems to remain almost unaffected. About the second operation loops, it's possible to collapse the two just taking care that the inner assignment is safely performed. To handle that, `#pragma omp critical` instruction has been compared with a `reduction(max)` over the three variables used:

- `parallel for, collapse(2)`, and the operations as `critical` sections: 2.05s
- `parallel for, collapse(2)` and `reduction(max:maxR, maxG, maxB)`: 1.95s

The final implementation merge the two operations in a same visit using `collapse(2)` and `reduction(max:maxR, maxG, maxB)`, allowing to perform on my hardware a data read in 0.8 seconds.

### 4.4 K-search

To make the program more complete and dynamic, the behaviour for K-search has been implemented as well, allowing the execution of the algorithm over a range of K and retrieving the number of clusters which minimize the within-variance. Both the search iterations over different K and the algorithm itself can be executed in parallel specifying the number of threads allocated for the job. Note that with the argument `-ksearch 2 4` two iterations will be executed in parallel and each one will make use of a pool of 4 threads.
This extension allowed also a more comprehensive testing phase for the code implementation.

## 5 Performance Testing

Since the "embarrassingly parallel" nature of the K-means clustering algorithm I was expecting very good results executing the algorithm in a parallel manner. The evaluation was conducted in order to determine speedup, scaleup and sizeup measuring the times of execution with the `omp_get_wtime()` instruction.

### 5.1 Data analysis

First I ensured the validity of the data and the correct execution of the algorithm evaluating the k-means output for the current dataset. I executed the algorithm on different portions, and with different K. Using the within-cluster variance as metric of accuracy the necessary amount of iterations was choosen at 25, and the optimal value of K at 7 using the rule of thumb of the elbow method.

The Fig.1 shows the RGB channels for the original image, the cluster map after one iteration and the cluster map after 20 iterations. The appearance of the cluster is already good around 10 iterations primarily due to the fact that the region covered by the image consists of a relatively small number of spectrally similar rocks and minerals. Fig.2 shows the number of pixels that are assigned to a new cluster, as a function of the k-means iteration number. Convergence of the clusters is apparent from the number of migrating pixels: around values of 2% of pixels still migrating the within-cluster variance start increasing (even if the n° of pixels are decreasing), therefore I've picked 25 as optimal number of iterations at which stop.

### 5.2 Implementation analysis

I've timed the main section of the algorithm collecting for the evaluation the total time of execution, the average time of a single iteration and the also the average time per iteration divided by the number of clusters, to compare the k-searches performance.
Further of my personal computer used for the initial evaluation part, the High Performance Computing service on cluster (Cluster HPC) of the DISI from University has been used. It, consists of quad-core nodes allocated to user's needs following Slurm job-scheduling system.
To determine the fastest parallel search and compare basic and parallel clustering algorithms I executed various simulations of k-search with different number of threads, collecting the results in the table below. Since the focus, there, was exclusively regarding the time performance, to collect the results I've considered a single execution per parameters combination, but with an higher number of iterations (50). Indeed being the initialization not random the results do not differ so much from execution to execution.

We can see from the Tab.1 and more clearly from the Fig.3 that parallelizing the K-Means algorithm on many processor brings gains in terms of computation time. These positive effect holds till 4 parallel threads which is the number of physical cores in the machine of execution; exceeding such limit and increasing the

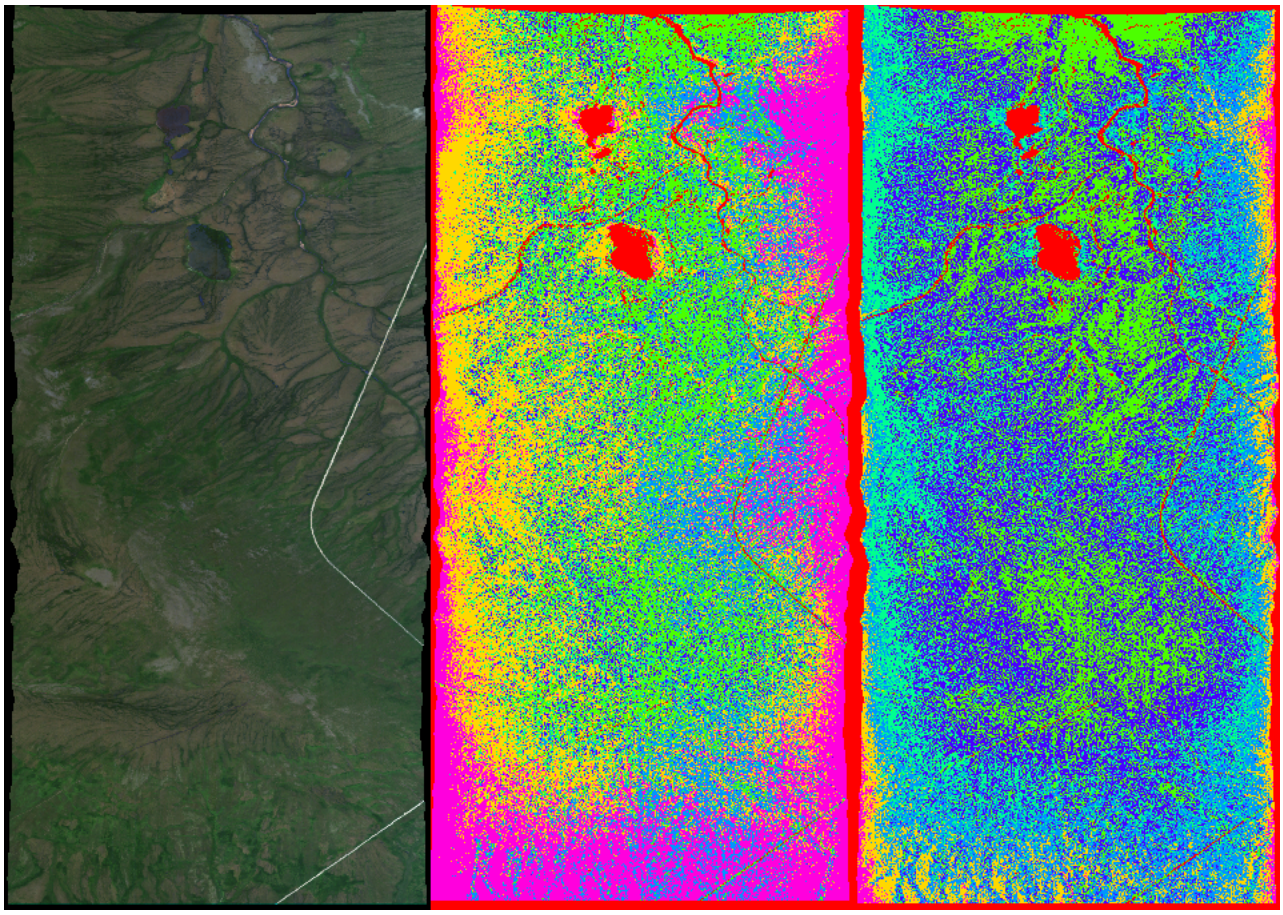

Figure 1: Results of k-means algorithm for dataset: (left) test image, (center) 1 iteration, (right) 20 iterations.

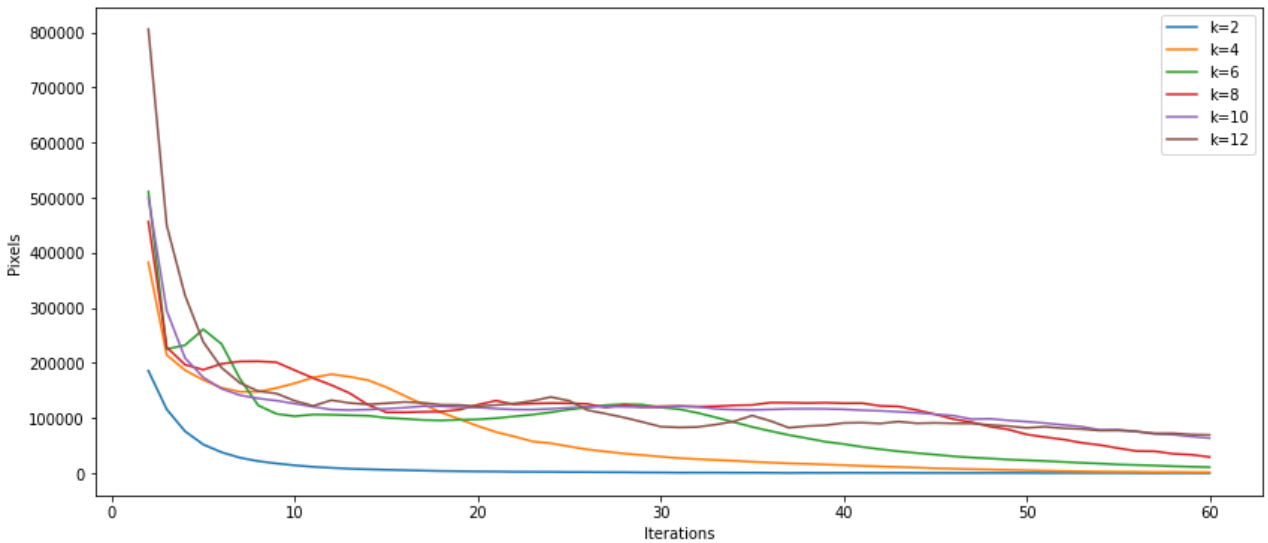


Figure 2: Pixels changed in function of the iterations for K-Means executions with different K and same amount of samples.

| CPU for search | | 1 | | | | | | | 2 | | | | 3 | | | | 4 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CPU for K-Means | | 1 | 2 | 4 | 6 | 8 | 12 | 16 | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 | 1 | 2 | 4 | 8 |
| Times per iteration (s) | k=2 | 9,4 | 5,7 | 4,4 | 4,4 | 4,1 | 4,1 | 4,8 | 10,6 | 9,0 | 9,0 | 10,2 | 9,9 | 14,7 | 14,5 | 14,8 | 15,2 | 17,8 | 18,5 | 19,4 |
| | k=3 | 12,7 | 7,3 | 5,5 | 5,3 | 5,3 | 5,2 | 6,4 | 14,5 | 11,8 | 12,0 | 13,1 | 13,4 | 19,4 | 19,0 | 19,8 | 21,5 | 24,8 | 25,3 | 25,8 |
| | k=4 | 16,0 | 9,0 | 6,6 | 6,3 | 6,3 | 6,1 | 7,9 | 18,3 | 14,5 | 14,8 | 16,3 | 16,9 | 24,0 | 18,8 | 24,8 | 26,8 | 31,0 | 31,8 | 32,3 |
| | k=5 | 19,2 | 10,4 | 7,8 | 7,4 | 7,3 | 7,3 | 9,4 | 21,9 | 17,6 | 17,6 | 19,7 | 20,9 | 28,9 | 26,9 | 29,2 | 34,6 | 37,6 | 38,0 | 38,9 |
| | k=6 | 22,4 | 12,0 | 9,1 | 8,6 | 8,6 | 8,2 | 10,7 | 25,6 | 20,6 | 18,9 | 23,4 | 23,8 | 33,1 | 33,1 | 34,0 | 36,6 | 42,5 | 43,0 | 43,4 |
| | k=7 | 25,6 | 13,7 | 10,2 | 9,7 | 9,6 | 9,5 | 11,8 | 27,9 | 23,2 | 22,6 | 26,4 | 26,8 | 37,9 | 36,5 | 39,1 | 44,5 | 39,0 | 38,7 | 39,1 |
| | k=8 | 29,0 | 15,2 | 11,3 | 11,0 | 10,8 | 10,5 | 10,7 | 33,9 | 25,8 | 25,1 | 28,6 | 33,0 | 34,3 | 28,6 | 34,8 | 53,3 | 56,8 | 57,2 | 58,2 |
| | k=9 | 32,8 | 17,0 | 12,5 | 12,0 | 11,8 | 11,5 | 11,8 | 36,9 | 28,9 | 28,5 | 32,0 | 37,1 | 32,5 | 37,4 | 33,2 | 52,3 | 52,3 | 52,6 | 53,2 |
| | k=10 | 36,4 | 18,6 | 13,7 | 13,1 | 11,4 | 12,8 | 12,7 | 40,0 | 32,1 | 27,2 | 36,1 | 37,3 | 51,6 | 49,4 | 53,0 | 43,2 | 32,3 | 30,2 | 31,3 |
| | k=11 | 39,8 | 20,2 | 15,1 | 14,5 | 10,9 | 13,9 | 13,8 | 39,9 | 24,8 | 27,0 | 26,2 | 43,2 | 49,8 | 46,8 | 50,8 | 68,0 | 75,6 | 76,5 | 77,6 |
| | k=12 | 42,1 | 21,9 | 16,1 | 15,5 | 11,6 | 15,5 | 15,0 | 43,9 | 22,0 | 21,8 | 21,8 | 48,1 | 40,1 | 41,0 | 40,1 | 65,3 | 55,7 | 56,2 | 56,9 |
| Sum of times per iteration divided k (s) | | 3,61 | 2,26 | 1,56 | 1,50 | 1,40 | 1,45 | 1,66 | 4,27 | 3,28 | 3,22 | 3,65 | 4,14 | 5,22 | 5,02 | 5,32 | 6,30 | 6,61 | 6,69 | 6,83 |
| Total time (s) | | 13483 | 7547 | 5638 | 5412 | 4899 | 5303 | 5819 | 9753 | 6701 | 6499 | 7259 | 6448 | 7085 | 7169 | 7210 | 7461 | 7089 | 7019 | 7156 |

Table 1: Execution times of k-searches with different n° of threads (red highlight the physical cores exceeded)

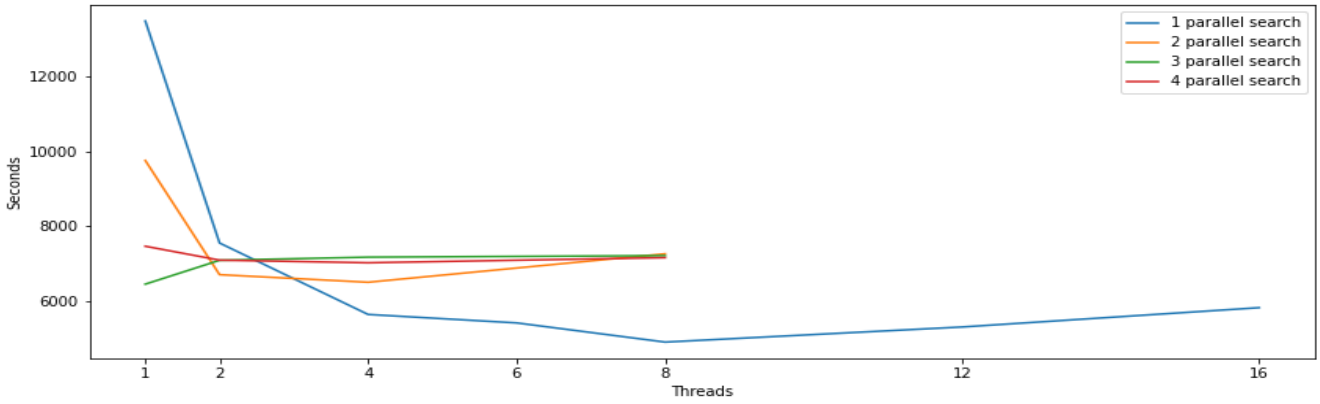


Figure 3: Total time of execution of k-searches compared on the base of threads used

number of threads won't lead to significant improvements. This is more visible in the case of many searches in parallel where the cores, having to run multiple threads, will compete for cpu cycles and resources. Furthermore comparing results with same number of threads in use, highlight the disparity of performance in having multiple searches at the same time on the same node. This is probably because we are forcing the execution on larger areas of memory at the same time.

While the results in Tab.1 and Fig.3 provide some information about the differences between the serial and parallel K-Means algorithms, runtime alone is not a sufficient measure of scalability. Therefore I incorporated measures of speedup, scaleup, and sizeup to gain more insights into the implementation advantages.

- Speedup is measured by keeping the number of observations in a dataset constant but varying the number of processors being used. To calculate speedup you must capture the time it takes for the algorithm to run using 1 processor ($T_1$) and the time it takes for the algorithm to run using p processors ($T_p$). An algorithm is described as perfectly parallel if it displays a linear speedup [3].

- Scaleup can be defined as the ability of a p-times larger system to perform a p-times larger job in the same execution time [3]. To measure scaleup you must calculate the time it takes for the algorithm to run using 1 processor on a dataset of size s ($T_{1s}$) and the time it takes for the algorithm to run using p processors on a dataset of size p x s ($T_{ps}$), therefore increasing both the number of processors and the size of of the dataset. An algorithm is considered to have good scaleup if the ratio is close to 1 [3].

- Sizeup is measured by keeping the number of processors used constant but increasing the number of observations in the dataset by a factor p. It represents the capability of an algorithm to handle a p-times larger dataset. Measuring sizeup requires calculating the time it takes for the algorithm to run on a dataset of size s ($T_s$) and the time it takes for the algorithm to run on a dataset of size p x s ($T_{ps}$). An algorithm has good sizeup performance if the ratio is close to p [4].

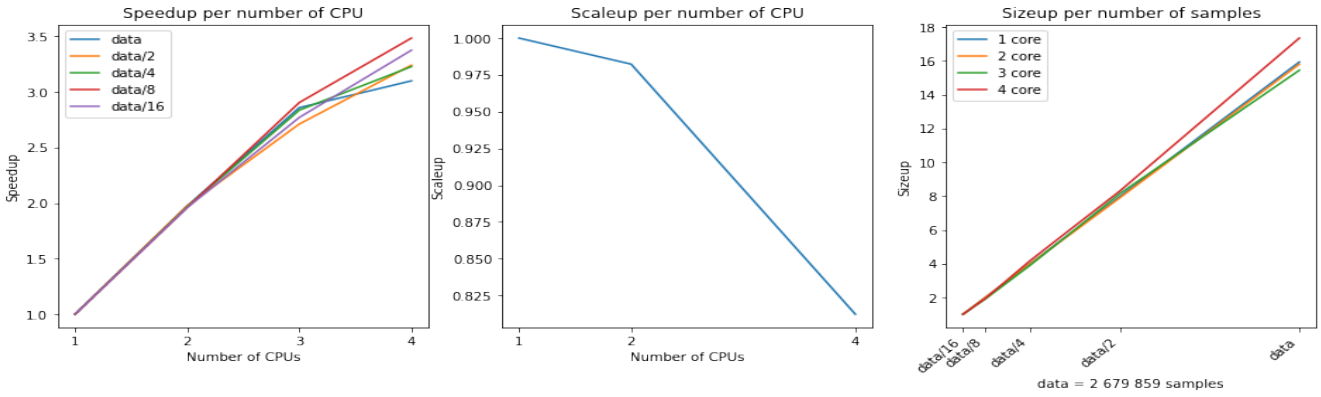


Figure 4: Total time of execution of k-searches compared on the base of threads used

We see from Fig.4(left) how speedup increases linearly with the number of threads up until 4 threads, after which speedup remains almost unchanged. This, again, is expected since the program was executed on a 4-core system. Fig.4(middle) display a decreasing slope for the scaleup measure but anyway the values are enough to say that the parallel K-Means has a good scalability. Finally Fig.4(right) show the inverted ratio for sizeup. The value is perfecetly in accordance with the ratio of samples used for the analysis therefore the algorithm has got also the capacity to efficiently support large datasets.

# 6 Conclusions

In this report, I propose a parallel version of the K-Means clustering algorithm implemented with C++'s OpenMP parallel programming API. The implementation cover also aspects like K-search functionality, executable in a multiprocessor manner as well, and a visual demonstration of the algorithm execution through GUI.
A series of simulations to compare the runtime and speedup between the serial K-Means algorithm and the parallel version has been executed. The collected results allowed a finer analysis comprehensive of the impact of using more or less threads in the parallel algorithm. The findings confirm what I intuitively thought would happen based on the "embarrassingly parallel" nature of the algorithm. The parallel version of such algorithm come particularly handful with big datasets and measures for speedup, scaleup and sizeup provide further evidence that the algorithm is both robust and scalable.
Out of the reported analysis, has been observed how using a really small portion of the dataset, 20k out of 2.7M samples, the serial version outperforms the parallel version.
A further extension of this work could focus on the scalability in a cluster environment with many nodes at disposal, and in this context parallelize the k-search.

# References

[1] https://vedas.sac.gov.in/aviris_web/pdf/20150726_AVRISINGDataGuide_v4.pdf

[2] https://www.l3harrisgeospatial.com/docs/PreprocessAVIRIS.html

[3] DeWitt, David, and Jim Gray. "Parallel database systems: the future of high performance database systems." Communications of the ACM, vol. 35, no. 6, June 1992, pp. 85+. Gale Academic OneFile, link.gale.com/apps/doc/A12353479/AONE?u=googlescholarsid=googleScholarxid=bc9bacb0.

[4] Xian-He Sun, John L. Gustafson, Toward a better parallel performance metric, Parallel Computing, Volume 17, Issues 10–11, 1991, Pages 1093-1109, ISSN 0167-8191, https://doi.org/10.1016/S0167-8191(05)80028-6.