2D k-means clustering parallelization

Advanced Computer Architecture lab project report

Luca Todaro, Rasha Zieni – University of Pavia, year 2020/21

Table of contents

Introduction

This lab report showcases a serial implementation of bidimensional k-means clustering algorithm as well as its parallelization using OpenMP [1] library available in C++. The report aims to demonstrate the performance gain achievable with multi-threading architectures and discuss the limitations that comes with them, so after a theoretical overview on the k-means formulation we will propose and discuss a viable serial and parallel script. The tools used are CLion IDE [2] along its integrated profiler, Google Cloud Platform and Git; both the computers used for programming and testing are based on Windows 10. The programming languages involved in this project are both C++ for the actual algorithm and Python for data processing, made easy thanks to its go-to plotting library matplotlib: while it is true that C++ also features some plotting utilities, we thought unnecessary to overcomplicate something that could otherwise be carried out in few lines of code.

Algorithm overview

K-Means is an unsupervised learning algorithm that solves the clustering problem that is, to assign a label to input data according on a given number of “clusters”; clusters are groups that divide data into categories according to specific features shared between each data point. So given a dataset if we want to divide the data points into K different clusters, K-means will initialize K random points as “centroids” and then will start assigning data to corresponding clusters depending on how close they are to each centroid.

Given a dataset with N points and a number of centroids K,

1. Pick K random points from the dataset.
2. For each point, compute the distance between the point itself and every centroid and assign it to the nearest.
3. Recompute centroids on newly formed clusters.
4. Repeat from step 2.

The algorithm converges once it satisfies one or more of the following conditions:

1. A specified number of iterations is reached.
2. At least one centroid does not change its position.
3. Points do not change their cluster assignation or migrate less than a threshold.
4. Sum of distances is minimized.

It is worth noticing that the number of points N can be both given or not but if it is not specified then the algorithm must compute it by itself, typically adding an enumerating function before the actual K-Means loop; because the choice of centroids is random, the algorithm might converge in a different number of iterations among different runs and can also end up with different cluster assignations.

Serial implementation

For the reasons stated above K-Means is a non-deterministic algorithm by nature, and this makes it hard to monitor run outputs. We opted for the purest implementation even though there are a lot of versions available on the internet (and more optimized, i.e., K-Means++) and to suppress some variability we decided to load different datasets from their corresponding file, providing the number of points and clusters as a given. Clusters are picked randomly among the datapoints, and we set as exit condition both the maximum number of iterations and at least one centroid do not change its position.

Algorithm implementation

Because the programming language choice is C++, we immediately took advantage from it by defining two classes: `Point`, to model a point (so it has two coordinates x and y with double data type plus an int as id to assign each point to a cluster, acting as an identifier) and `Cluster` as model for a cluster; because each cluster actually is a Point, carries a Point object and a dimension that is, the number of points within that specific cluster. It is worth noticing that there is no need to add an identifier for each cluster, as it is possible to take advantage from the fact that they can be identified with their array index.

The algorithm is not that hard to implement as there are not many things to take care of such as memory occupancy because the amount of data points and clusters are known and do not change over time. The datasets are always loaded from file so that all experiments have the same number of points in the exact same position and clusters, randomly picked within the datapoints, are instantiated in a fixed length array because their number never change as well. The choice of using an Array compared to a Vector object is mostly due to the fact that usually Vector takes more time to access objects and is less memory efficient, adding to that Arrays can access elements in constant time because they are stored in contiguous memory elements whereas Vectors do not.

Right after algorithm initialization the main loop starts with perhaps the most important function that is `naive_kmeans` [3]. In this function there are two nested loops, the outer loop iterates over all the data points and the inner loop iterates over the cluster array; thus, Euclidean distances are computed between a single point and all cluster pivots to be assigned at the cluster with the least distance. We did not mention earlier that clusters actually have two Point objects because someone might think that it is okay to update cluster coordinates immediately after it gets a new point assigned to; this is wrong though, as the description of the algorithm implicitly states that cluster pivots *do not move* within an iteration. When the outer loop ends an iteration of K-Means is considered concluded and clusters get updated with their new centroid coordinates; notice that the updating function `update_centroid` [4] itself must carry out some math, as it must divide the newly computed centroids coordinates by the total number of points assigned to that cluster. Finally, new centroid coordinates are set as current centroid coordinates and cleared.

Algorithm validation

Because we decided to implement our own serial code, before considering how to parallelize it we also had to validate how accurate it is; to accomplish this, we made some tests with the iris [5] dataset against the K-Means Python implementation built in Scikit [6] library, which we considered “state of art”. We chose the iris dataset to validate the algorithm, but we also made some synthetic datasets featuring different numbers of points drawn from gaussian distributions.

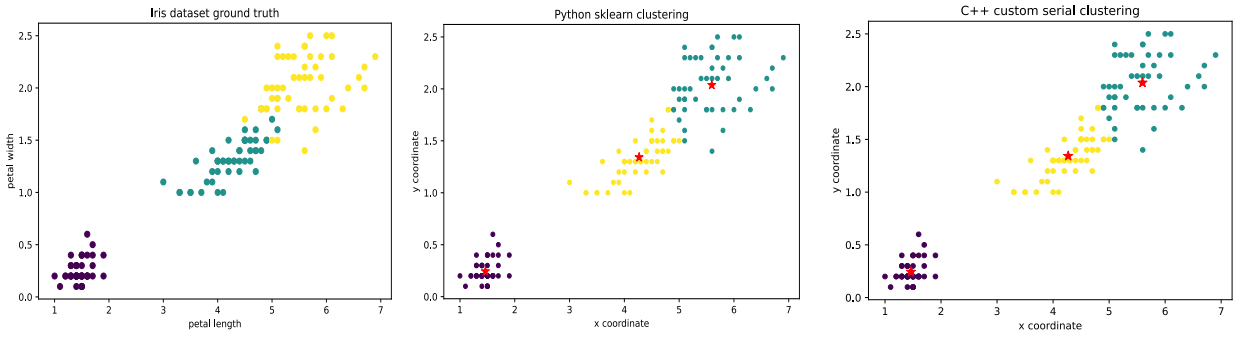

Fig. 1: comparison between clustering outcomes. Ground truth from the labeled dataset itself (left), Python clustering outcome from Scikit library (center) and C++ algorithm outcome we implemented (right). Colors assigned to clusters are irrelevant.

By looking at the plots above, it is safe enough to say our mathematical implementation is good enough to be benchmarked as the algorithm works as intended; the classification errors are due to the fact some flower breeds might be easily confused with those from the nearby class as their data overlaps, while the small difference (if any) in classification incurred with other datasets (not reported here) is mostly consequence of the random choice for initial pivots.

Parallel implementation

K-Means is very prone to parallelization because data points are independent one to each other, however, also involves frequent communication between each node. We want to understand what are the factors that affect the speedup (if there are any) and what are the benefits from running our parallel code on machines with multiple cores; because of this, we first tested using our personal laptop (intel i7-10710U 6 cores 12 threads, 16GB RAM), (intel i5-7200U 2 cores 4 threads, 8GB RAM) and Google Cloud platform VMs (Intel Xeon up to 24 cores).

Identifying relevant code areas to parallelize

The most obvious thing to parallelize is the whole k-means loop because data points are independent from each other; however, it is worth studying what parts of code are more intensive in terms of execution time because, according to Amdahl's Law, we want to make the common case faster: for this reason, we tried to benchmark mainly 3 parts of our serial algorithm using different input datasets, but all set to 10 clusters. Follows the results gathered and expressed as average on 15 subsequent runs.

	Serial code segments		
Points, clusters	*k-means loop iteration*	*Cluster update*	*Distance computation*
10000, 10	17.184 ms	0.009 ms	10.749 ms
50000, 10	76.408 ms	0.011 ms	47.148 ms
100000, 10	148.816 ms	0.013 ms	94.575 ms

By looking at results reported in the table, the algorithm spends most time (approx. 63%) to perform computations within a whole K-means loop iteration and this provides us a good indication where parallelizing can give a good performance optimization: it makes no sense to parallelize cluster updates because it will waste more time for creating and managing threads than executing the instructions.

Algorithm implementation

The whole k-means loop is represented by `naive_kmeans` [7] function which has the core operations of the algorithm and considering that these operations are independent of each other, parallelizing them gives a very good speedup for the algorithm; said operations include computing the distance between the points and the centroids of the clusters, followed by the assignation of points to the nearest centroids as described in the theory. According to this observation, we parallelized the outer for loop in the `naive_kmeans` function using static scheduling because iterations almost take the same amount of time to execute and to not pay the overhead by using dynamic scheduling; we also made `min_distance` and `min_cluster_index` variables Private for each thread so that each one has its own copy of the two.

For synchronization we used the directive `pragma_omp_critical` [8] to prevent racing between threads when assigning points to clusters; we chose to use it over the directive `pragma_omp_atomic` because, even though atomic is known to be faster, in our specific implementation we had to protect three regions from racing: using three atomic directives (three because atomic is only applicable on simple operations such as assignments and those involving unary operators) more time is wasted when threads need to wait in three regions to access one at a time, so using a single `pragma_omp_critical` was the best choice for this case. And not only that: our class `Clusters` carries two `Point` objects so if we had to use the atomic directive,

we would have to change points into two x and y coordinates changing all other methods related to them, but for sake of understandability it is better to leave the code as similar as possible to the serial one.

Data gathering and analysis

To make things a bit easier we added a function both in serial and parallel implementation to log the run output to a csv file, `runs.csv`: because we already knew all the experiments are made with the random point dataset, we save the number of points constituting it, the number of processors, iteration, wall time and k-means time that is, the time k-means loop takes to finish. We excluded this function from benchmarks because it is just a write output to file *highly dependent* to the number of rows it must write.

Before starting with experimenting, we also made some assumptions on the results we expect. Perhaps the most obvious thing is that timings are dependent both on the number of clusters and on the number of data points; we assumed an increase in data points should cause a bigger impact though. However, we tried to determine a maximum theoretical value for the expected speedup because is very helpful to evaluate our results; to do so, we applied Amdahl's law as follows:

$$S_{speedup} = \frac{1}{(1-F) + \frac{F}{N}}$$

Where F is the fraction of parallelizable code and N is the number of processors. Our serial code only features a very small part that cannot be parallelized that is, the I/O part to load the dataset and to instantiate the pivots. We first needed to estimate the amount of parallelizable code and we did it by measuring the wall time of the whole program compared to the time spent inside the functions we plan to parallelize. Obviously, we cannot exactly compute the number of F as it scales up with problem size but still we can compute our maximum speedup expected regardless of the number of cores that is supposed to be *at least* a factor of 10 at F = 0.9 (it simply is the limit for N to infinite of the formula above).

Points	***Number of processors (N)***	***F factor***	***Theoretical speedup (S)***
100000	2	0.95	1.9
	4		3.5
	12		8
300000	2	0.97	1.9
	4		3.7
	12		9.6

According to the computations above, we expect a huge performance gain from parallelization although we also expect to get weaker performance gains after a certain number of processors involved in computation and as consequence, increasing data points might not necessarily imply to get better scaling. Data in the table above are estimates on a fixed number of clusters but they tell us to expect a linear increase up until 4 processors.

Another fact we immediately stated is that processor type counts (as instance, i5 or i7 or Xeon) and because of this, even with the exact same processors count, it is not possible to start the experiments on a machine and continue them on another one.

Test datasets

As stated earlier, almost all datasets are generated from python scripts and all of them are txt files with a point coordinates on each row. Float values are allowed. The only pre-made dataset we tried is the Iris, but in order to make it suitable for 2D representation we sampled only Petal width and Sepal length field; what

made it not suitable to test our parallel code execution, though, was its data point scarcity at only 150 entries. Because 150 points are too few to appreciate any performance gain from parallelization, we decided to generate datasets of different number of random points from 10000 to 300000.

Local Experiments

Our first experiments are on our local machine with 6 cores to make sure everything makes sense before attempting the same runs with Google Cloud Platform as credits provided are an important resource that should not be wasted. Although our laptop has various processes already running on it, we tried to reduce possible slowdowns by closing all the applications running in background and we plugged it to wall socket to avoid any throttling on the processor caused by battery management profiles.

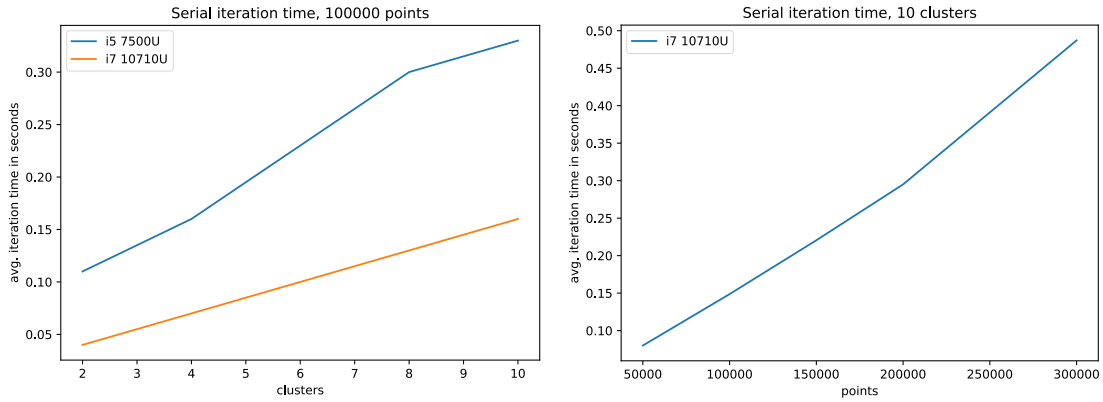

Fig. 2: comparison between average k-means iteration time and a changing number of points or clusters

Regardless of choosing to run the experiments with our best machine, we added another line in the leftmost plot to show as additional information the difference in time between two processors from Intel; while we expected such behavior the most interesting fact is that, comparing the two plots, the slope of the line in the right plot is bigger than the one in the left plot. We can say that, whereas increasing both the number of clusters and number of points causes an increase in time for the algorithm to complete a single k-means iteration, varying the number of data points is more impactful than increasing the number of clusters.

Moving on to benchmarking our parallel implementation, we first want to stress the fact that there is not a specific number of k-means iterations that stays constant between each run and grade of parallelism as the random choice of initial pivots makes it impossible to assume a correlation between number of processors and number of iterations. Because of that, we always run experiments multiple times for each configuration and after that we compute the mean values between all those we gathered. The first experiment consists in changing the number of points for each processor count to test how parallelism handles the increase of data to process.

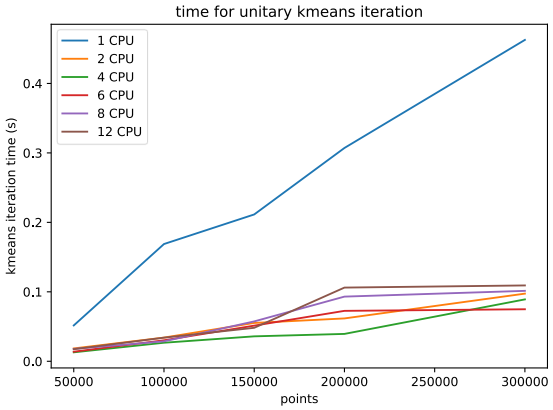

Fig. 3: time taken by a single k-means iteration varying the number of processors and data points.

Of course, our laptop does not actually have 12 CPUs but it does have 6 cores with Hyperthreading technology, so we decided to also run the experiments enabling it to test how OpenMP behaves. Also, we decided to monitor the time each single iteration takes to complete in order to make the measurements independent from the number of iterations done: by doing this way it is possible to completely eliminate the aleatory process that is the initial pivot selection. We made sure to disable hyperthreading for experiments up until 6 cores directly in the BIOS to avoid any possible environment variable override somewhere in Windows that may lead us in the misbelief of running without it.

Overall, the results are promising: the runs in “serial mode” have an almost perfect linear increase, which is expected as it has nothing to scale onto. Runs with two and four cores provide good results and scale properly up until 200000 data points, after that threshold they start to struggle into handling the load increase; it is worth saying that in the region between 150000 and 200000 points they almost stay constant. This is important to notice because instead parallel runs with six cores scale better when data points are more than 200000, and this is relevant because starting from that threshold it behaves exactly how the runs with less cores are doing earlier with 50000 less points. After 6 cores we tried enabling Hyperthreading [9], but we did not get any significative improvement on what we previously observed: the increase of time compared to the 6-core run is also due to context switch highlighted by the small difference between the run with 8 and 12 cores. Thus, hyperthreading may help in gaining some performance but having a physical core is still strictly better.

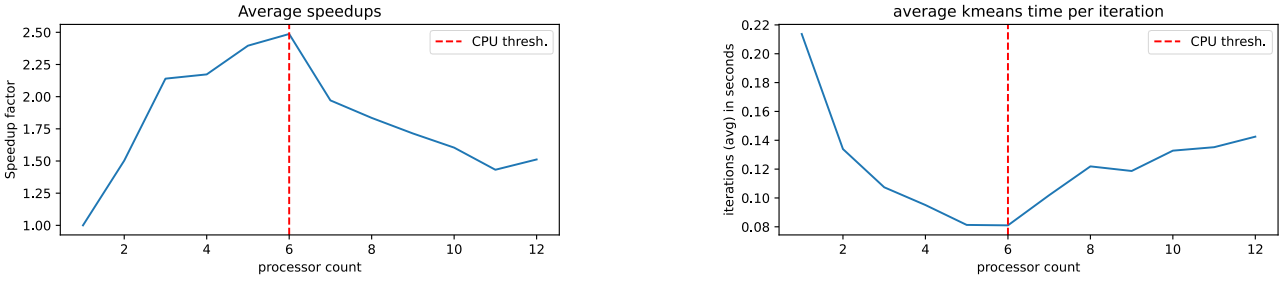

Fig. 4: speedup (left) and close-up detail (right) on 300000 points, 10 clusters run. The red dashed line indicates the threshold after which Hyperthreading is enabled.

The two plots in figure 4 validate the discussion on the results we did: parallelism indeed provide a huge performance gain of almost 2.5 times the serial implementation took to complete; after 6 cores the performance gain starts decreasing because of OpenMP that cannot take advantage of Hyperthreading as it

simply maps 1 to 1 thread with physical cores. Not only that: because the total k-means time decreases, as well decreases its relevance when computing the wall time, as it gets nearer to the timing scored by the non-parallelizable parts meaning that wall time will not get lower than the serial code time. We also noticed after running the code with Vtune profiler, that with 12 threads the critical section we introduced becomes more relevant in the loop compared to the normal one whereas on 4 threads it becomes negligible, as if hyperthreading spends more time waiting for locks.

Google Cloud Platform Experiments

We started the experiments on Google Cloud immediately with a 24-CPUs virtual machine so that we did not have to continue changing the instance parameters: the only thing we had to change were few variables in our code implementations according to the type of experiment we want to run and gradually increasing the number of processors as needed.

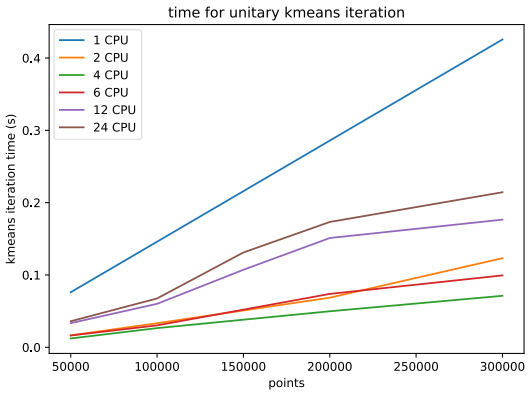

Fig. 5: GCP run time to complete a single K-Means iteration. Clusters are fixed to 10, no hard limits on iterations.

Scaling up until 6 CPUs confirmed the benchmarks we obtained locally because we obtained almost the exact same timings. The only thing is missing compared to our local runs are plateau areas already discussed, but it simply is something related to the processor type provided by Google: it has an almost linear behavior with the increase of the load. Another noticeable difference is the fact that serial runs take less time on 300000 points dataset and almost the same time on the one with 50000 entries again compared to our local results, yet this might still be something related to the processor type itself. However, increasing the number of threads lead to an overall increase in the time to finish a single iteration, which again validates the results we obtained in local tests. Not only that: it is worth noticing that cranking up the number of cores performs even worse in handling a relatively small set of datasets.

We again focused on the dataset of 300000 points to compute the speedup we obtained varying the number of processors: it is clear even by only looking at Fig. 5 that the expected behavior is a steady decrease until four to six core counts.

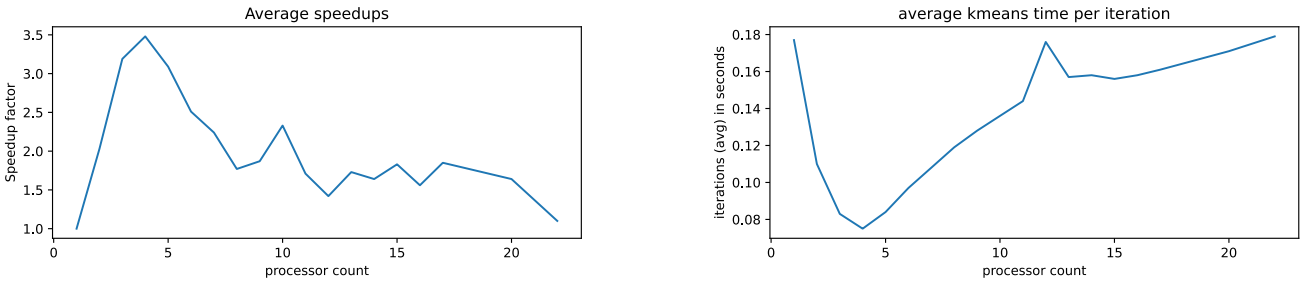

Fig. 6: speedup (left) and detailed run (right) on 300000 points, 10 clusters dataset on Google Cloud Platform.

While it is true that the speedup plot in figure 6 is not regular, we still can appreciate a big increase up until 4 processors only to let the line decrease slowly after this threshold. The increase indeed is reflected in the huge drop in the rightmost chart that corresponds to a drop in a single iteration time by a factor of more than 3. Compared to what we got in local tests the results are not much far: the behavior remains the same but on Google Cloud the maximum number of cores that provided the best performance are 4 against 6 processors with our computer; however, the best timing we achieved was the same (0.08 s) although the serial time to complete a single iteration is a bit higher.

Conclusion

K-Means is an easy algorithm to implement but is very sketchy when it comes to parallelize because of it being nondeterministic, and both in this case or any other application we learned that when it comes to parallelization it is important to start from a good serial implementation at first instance, as it will massively affect the results obtained. In fact, serial performance problems tend to get worse when run in parallel. The measurements we reported from all our experiments all agree on the fact that we managed to get a linear speedup instead than a super linear one, which is something may happen if serial algorithm involves more overhead or its hardware favors parallel algorithms instead. The linear increase also confirmed Amdahl's law along with our theoretical hypotheses on the maximum speedup expected and in fact from 2 to 4 processors we even managed to get very close to the theoretical values; this result is both reflected in local and remote experiments.

We confirmed our hypothesis that parallel performance is also a function of problem size and number of processors involved in the execution: the more data points we input, the more the wall time increases; not only that, because the more the points are the more increases the parallel section relevance compared to the serial part. However, parallelism remains a problem deeply tied to the algorithm and the architecture chosen for the implementation and while the best thing to do would be to try and parallelize the most out of the serial code, the performance gains achievable from running it on a multi-processor machine are still limited by the serial code timings left.

In our specific problem turned out the number of clusters are not as much impactful compared to the number of data points and the number of iterations does not have a correlation with anything at all; this makes sense, as we are dealing with a random process.

Possible improvements: K-Means++

The weakness point in our algorithm is how we choose pivots: according to their choice, the whole algorithm can converge in much less iterations and a poor choice of centroids may result in poor classification results as well. K-Means++ [10] somehow manages to eliminate the variability in the selection of cluster centers by doing an initial pivot-selection loop before starting the actual naïve K-Means algorithm, as follows:

1. Choose one center at random among all data points.
2. Compute the Euclidean distance between the nearest center and all the remaining points.
3. Select as centroid the point such that it has maximum distance from the nearest center.
4. Repeat steps 2 and 3 until K centroids are selected.

The idea is, by doing these steps we are selecting centroids that are very far one each other and this increases the chances of initially picking up pivots in already different clusters. While it is true that it adds another loop on top, the performance boost should be made possible thanks to pivots already placed somewhere near their respective center clusters; however, we did not test it because it is outside our lab experience scope.

Contributions

Project work was carried out together and we both contributed to the whole engineering process of the code as we discussed extensively all design choices we made. We thought it was important for both of us to do everything, so we helped each other when doing certain tasks: when one was busy, the other helped. Workload division was as follows:

Rasha Zieni: parallel code implementation, python serial kmeans implementation, data gathering, project report, github repository documentation

Luca Todaro: serial code implementation, data visualization, data gathering, parallel code implementation, GCP server setup bash script, project report, github files merging

References

[1] OpenMP, "The OpenMP API specification for parallel programming," [Online]. Available: https://www.openmp.org/.

[2] JetBrains, "C Lion IDE," JetBrains, [Online]. Available: https://www.jetbrains.com/clion/.

[3] R. Z. Luca Todaro, "Github - kmeans clustering parallelization, naive_kmeans function," [Online]. Available: https://github.com/A7F/aca-kmeans/blob/43a722f78e10222f0957e2a7346ea48b02f9393a/serial.cpp#L131.

[4] R. Z. Luca Todaro, "Github - kmeans clustering parallelization, update_centroid function," [Online]. Available: https://github.com/A7F/aca-kmeans/blob/43a722f78e10222f0957e2a7346ea48b02f9393a/cluster.h#L63.

[5] R. A. Fisher, "The Iris dataset," Center for Machine Learning and Intelligent Systems, 1988. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/iris.

[6] Scikit, "Scikit-Learn: K-Means clustering documentation," 2019. [Online]. Available: https://scikit-learn.org/stable/modules/clustering.html#k-means.

[7] R. Z. Luca Todaro, "GIthub - kmeans clustering parallelization, parallel code region," [Online]. Available: https://github.com/A7F/aca-kmeans/blob/43a722f78e10222f0957e2a7346ea48b02f9393a/parallel.cpp#L133.

[8] R. Z. Luca Todaro, "Github - kmeans clustering parallelization, critical section," [Online]. Available: https://github.com/A7F/aca-kmeans/blob/43a722f78e10222f0957e2a7346ea48b02f9393a/parallel.cpp#L151.

[9] Intel, "Intel thread affinity interface," [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/openmp-support/openmp-library-support/thread-affinity-interface-linux-and-windows.html.

[10] S. V. David Arthur, "k-means++: The Advantages of Careful Seeding," 2006.