K-Means Clustering

Advanced Computer Architecture

Technical Report

UNIVERSITÀ
DI PAVIA

Alessio Cangiano - 543497

Carlo Galante - 535663

Abstract

The goal of this project is to provide an efficient implementation of the K-Means algorithm, that exploits multiple CPUs, using Open MPI (on C++).

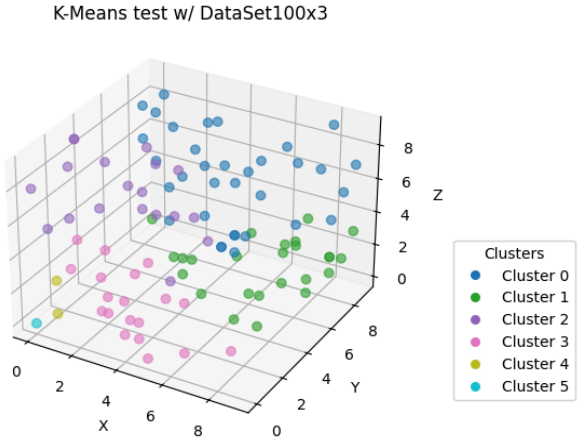
Starting with a serial implementation, the code has been changed to operate in parallel, resulting in a notable speedup and performance gain (in comparison to the serial version).
To determine whether portions of the serial code could have been parallelized, an a priori analysis of available parallelism has been carried out.
The size of the datasets and the number of processors have been changed in order to better understand how this implementation scales. The code has been run on Google Cloud Platform virtual instances to maximize the processing capabilities of a cluster of machines working together.

Contents

Overview of the Algorithm

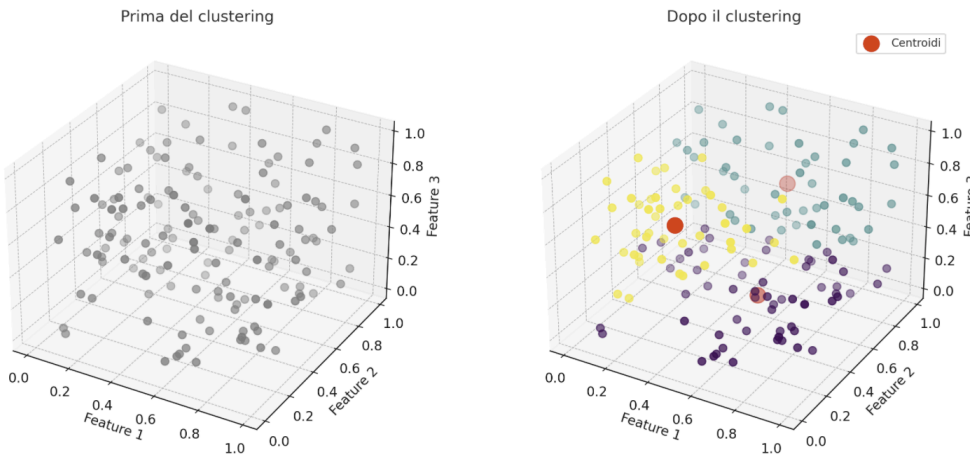
K-Means Clustering is an unsupervised machine learning algorithm that divides a dataset of m elements into k clusters. It starts by initializing k centroids from the m elements, categorizing each element based on the nearest centroid, creating k distinct clusters.

The new centroid of each cluster is determined by calculating the mean of the elements. The primary objective is to find the partition that minimizes the Total Within Cluster Variance.



The global minimum is not guaranteed due to the initialization of k centroids. While heuristics can be used for positioning centroids, they cannot guarantee the global minimum achievement.

A shared strategy involves repeatedly executing the algorithm and selecting the best clusters.



Description

Consider D = { $x(i) \in \Re^n$ } $where\ i = 1, \dots, m$ to be made up of m row, each with dimension n.
Let K be the number of clusters we want to classify our data into.

- Set up K centroids so that $\{c_k \in \Re^n\}$ $where\ k = 1, \dots, K$
- Assign $x(i)$ to the nearest c_k in order to define the partition $\{Cluster_k\}$ $where\ k = 1, \dots, K$ and $\{x(i)\ nearest\ to\ c_k\}$
- Update centroids by taking the mean of the contained elements, aiming to find the set of clusters that minimizes the objective function:

$$TWCV = \sum_{k=1}^{K} WCV_{(k)} \qquad where \qquad WCV_{(k)} = \sum_{i \in Cluster_k} \sum_{j=1}^{n} (x_{ij} - c_{kj})^2$$

Before starting the algorithm analysis, it is necessary to specify the termination condition that is used and how the centroids are initialized. The algorithm will differ depending on the method selected, which will also affect its cost.

The number of clusters and their location are the two primary decisions to be made here.

- The number of clusters
  Regarding this subject, the literature offers a variety of methods for choosing the appropriate quantity. For our project, we choose to use a simple rule of thumb:

$$K = \sqrt{\frac{m}{2}}$$

- The centroids' starting values
  For this project, we choose to randomly select the centroids' values by simply selecting K random points from the dataset used.

Regarding the termination, we adopted the following approach: since the goal is to minimize the TWCV, the process can be halted once the Total Mean Square Error ceases to show any variation.

Formally expressed:

$$\texttt{Repeat until } TMSE < TMSE_{prec}$$

$$TMSE = \sum_{k=1}^{K} MSE_{(k)} \qquad where \qquad MSE_{(k)} = \frac{1}{|Cluster_{(k)}|} \sum_{i \in Cluster_k} \sum_{j=1}^{n} \left(x_{ij} - c_{kj}\right)^2$$

Additionally, as the convergence speed of the algorithm depends on the initial placement of the centroids, it may take considerable time to satisfy the MSE condition. Therefore, an additional constraint on the maximum number of iterations *MaxIterations* is necessary.

Ultimately, the termination criteria for the algorithm are as follows:

$$\texttt{Repeat until } TMSE < TMSE_{prec} \texttt{ and } MaxIterations \neq 0$$

Pseudocode - Serial Algorithm

INITIALIZATION

Take a dataset $D = \{ x(i) \in \Re^n \}$ *where* $i = 1, \dots, m$

Initialize a set of K centroids $C(k) = \{c_k\}$ *where* $k = 1, \dots, K$

Initialize value for *MaxIterations*

Set precMSE = 0

EXECUTION

For $i = 1, \dots, m$

For $k = 1, \dots, K$

ComputeDistance $d(x_i, c_k) = ||x_i - c_k||^2$

End

S*elect k such that* $d(x_i, c_k)$ *is minimum, Assign* x_i *to* $Cluster_k$

$sumCluster_k = sumCluster_k + x_i$

Increase Counter of $Cluster_k$

$MSE = MSE + d(x_i, c_k)$

End

For $k = 1, \dots, K$

If $num(k) == 0$

Assign closest point x′ from another Cluster to $Cluster_k$

$num(k) = 1$

$sumCluster(k) = x'$

End

$c_k = sumCluster_k / num_k$

End

MaxIterations $--$

TERMINATION

If (MSE > precMSE || *MaxIterations* $==$ 0) *then TERMINATE*

Else

MSE_prec = MSE,

REPEAT

Analysis of Computational Complexity

K-means clustering, widely utilized in scenarios with large datasets, requires an understanding of its time complexity as a function of input dimensions. This implies analyzing the number of floating-point operations executed across three phases: Initialization, Execution, and Termination. Each iteration depends on the previous one's results and operations in a single iteration are consistent in volume. To determine the total computational effort, we analyze a single iteration and multiply the resulting cost by the number of iterations.

Symbols and Definitions:

- *c: Centroid*
- *m: Number of data points*
- *n: Dimension of each data point*
- *x: A data point*
- *K: Number of clusters*
- d_{ik}*: Distance between* x_i *and* c_k
- *I: Total iterations in K-means*
- *P: Number of parallel processes*
- $sumC_k$ *: Vectorial sum of the points in Cluster* C_k
- *TFLOP: Time required for one floating-point operation*

Phases of Algorithm:

Initialization Phase

This step involves loading the dataset and initializing variables, as described in the *pseudocode*.

The initialization remains identical for both serial and parallel versions of the algorithm.

Execution Phase

This phase constitutes the most computationally intensive segment of the algorithm.

For analytical clarity, it is broken into smaller tasks.

1. **Distance Computation**

   To determine the squared Euclidean distance between each data point x_i and all centroids c_k, the following formula is applied:

$$\| x_{(i)} - c_{(k)} \|^2$$

   This involves the following operations:

   - $x_i - c_k$ n operations
   - $||\dots||^2$ $1 + 1 + n + n \approx 2*n$
   - $\forall k$ K
   - $\forall i$ m

   For all m data points and K clusters, the total computational load becomes:

$$3 \cdot m \cdot n \cdot K$$

2. **Minimum computation**

   For each data point, the algorithm identifies the closest centroid by computing:

$$\min_k d_{ik}$$

   where d_{ik} is the distance to centroid k.

   Total computational cost: $m*K$

3. **Cluster Sum**

   All data points assigned to the same cluster are summed together. Since each data point has nn dimensions, the computation requires: $m*n$

4. **Counter incrementation**

   The overall count of operations for incrementing the counter is m, but these are integer operations. Since such operations account for only a small portion compared to the number of floating-point operations, their effect on the time complexity is

minimal for our analysis. Therefore, we opted to disregard them and focus on the floating-point operations in order to evaluate the algorithm's time complexity.

5. **Mean Calculation**

After summing the points, the mean for each cluster is determined by dividing the sum by the number of elements in the cluster. The formula is:

$$\forall k,\ c_k = \frac{sumC_k}{num_k}$$

Total operations for all clusters: $n*K$

6. **Termination Check**

The termination condition evaluates whether the Total Mean Square Error (TMSE) has stabilized or the maximum number of iterations has been reached. TMSE is computed as:

$$\text{TMSE} = \sum_{k=1}^{K} \text{MSE}(k)$$

Since all necessary values (distances and cluster assignments) are computed earlier, this step involves minimal additional cost: m

7. **Serial Computational Time Complexity**

The total number of operations for one iteration is the sum of all these components:

$$3*m*n*K + m*K + m*n + n*K + m$$

Simplifying for large datasets where $k >> 1$, $n >> 1$, m >> n,K, we approximate:

$$\approx 3*m*n*K$$

In our analysis, we consider only floating-point operations. Here, TFLOP denotes the time taken for a single floating-point operation, and I represents the total number of iterations of the algorithm. From this, we derive a computational time complexity:

$$\approx (3*m*n*K) * \text{TFLOP} * I$$

Parallelizable Blocks

Infra-Block Parallelization

The algorithm is divided into different phases, and each phase is analyzed to identify independent computations that can be parallelized within the same block:

1. **Distance Computation**

   The distance between each data point $x\,(i) \in R$ and each centroid $c\,(k) \in Rn$ is calculated using the squared Euclidean distance:

   $$\| x_{(i)} - c_{(k)} \|^2$$

   Since the computation for one data point or centroid does not depend on others, this phase is entirely parallelizable.

2. **Minimum Computation**

   Once the distances are computed, each data point is assigned to the cluster corresponding to the closest centroid:

   $$\min_k d_{ik}$$

   This operation is independent for each data point, making it fully parallelizable.

3. **Within-Cluster Sum**

   The Sum of all points belonging to the same cluster involves updating a vectorial sum for each cluster. This phase can also be parallelized, with partial sums computed independently for subsets of data and merged later.

4. **Counter Incrementation**

   Each cluster maintains a counter to track the number of points assigned to it. Since the update is local to each process, this phase is parallelizable.

5. **MSE in Cluster Sum**

   The Mean Squared Error (MSE) for each cluster is computed as the sum of the squared distances between data points and their assigned centroids. This computation, like the Sum phase, is parallelizable with synchronization at the end.

6. **Mean Calculation**

   The cluster centroids are updated by dividing the total sum of points in each cluster by the number of points in that cluster:

   $$\forall k,\ c_k = \frac{sumC_k}{num_k}$$

   Each cluster's computation is independent, allowing for parallel execution, but the final update requires synchronization.

7. **Termination**

   The termination condition, based on Total Mean Squared Error (TMSE), is evaluated as:

   $$\text{TMSE} = \sum_{k=1}^{K} \text{MSE}(k)$$

   Since the required distances are already computed, this phase involves minimal computation and is parallelizable.

Intra-Block Parallelization

While individual phases can be parallelized internally, dependencies exist between phases, meaning that certain steps must wait for the completion of preceding ones:

- Distance Computation → Minimum Computation: The minimum distance can only be calculated after all distances are computed.
- Minimum Computation → Cluster Sum and MSE: Sum and MSE updates depend on completed cluster assignments.
- Cluster Sum and MSE → Mean Calculation: The centroids cannot be updated until the cluster sums and counts are finalized.
- Mean Calculation → Termination: The termination condition is checked after centroids are updated.

Despite these dependencies, all operations within the same phase (e.g., calculating distances for all data points) are independent and can be parallelized. Synchronization is required between phases to aggregate results and ensure correctness.

Operations with Parallel Execution:

- Distance computation
- Minimum computation
- Within-Cluster Sum
- Counter incrementation
- MSE and cluster Sum

Operation with Serial Execution:

- Mean computation

We now need to evaluate the theoretical performance of this parallelization strategy. If the results are satisfactory, we can explore it in more detail. To assess performance, we will use theoretical speedup and scaleup.

Evaluation of Speedup and Scaleup

Speedup Analysis Based on Amdahl's Law

We can estimate the potential speedup from utilizing the available parallelism through Amdahl's Law.

$$\text{Speedup} = \frac{1}{(1 - p_{\text{parallel}}) + \frac{p_{\text{parallel}}}{P}}$$

Where:

- $p_{parallel}$ is the parallelizable fraction of the code,
- P is the number of processing units (processors).

The parallel fraction ($p_{parallel}$) is calculated as the ratio of the number of parallelizable instructions to the total number of instructions:

$$p_{\text{parallel}} = \frac{\text{Number of parallelizable instructions}}{\text{Total number of instructions}}$$

To compute the parallel fraction, we first need to identify which sections of the code can be parallelized and which sections will remain serial in the original code.

Operations and Serial Cost (N. Flops):

- Distance computation: 3×m×n×K
- Minimum computation: m×K
- Within-cluster Sum: m×n
- Counter incrementation: -
- MSE in-cluster Sum: m
- Mean computation: n×K

All the operations in the list above have parallel execution, except for mean computation which is serial.

The **number of parallel instructions** can be computed as:

$$m \times (3 \times n \times K + K + n + 1)$$

The **total number of instructions** (including both parallel and serial) is:

$$m \times (3 \times n \times K + K + n + 1) + n \times K$$

At this point, we can compute the **parallel fraction** of the serial code:

$$p_{\text{parallel}} = \frac{m \times (3 \times n \times K + K + n + 1)}{m \times (3 \times n \times K + K + n + 1) + n \times K}$$

If m is large, the parallel fraction simplifies to:

$$p_{\text{parallel}} \approx \frac{m \times (3 \times n \times K + K + n + 1)}{m \times (3 \times n \times K + K + n + 1)} = 1$$

Thus, the **speedup** becomes:

$$\text{Speedup} = \frac{1}{(1-1) + \frac{1}{P}} \approx P$$

This means that, for sufficiently large *m*, the maximum achievable speedup is approximately equal to the number of processing units *P*, as the parallel fraction approaches 1.

We can conclude that for large datasets, the maximum achievable speedup is approximately equal to the number of processes. However, this estimation does not take into account the overhead introduced by parallelization. A partial consideration of these overheads will be addressed in the next chapter, while the complete impact will be analyzed through experimental results.

Scaleup Analysis

In addition to speedup, another important metric to evaluate performance is scaleup. While speedup examines how the execution time of a fixed-size problem changes with an increasing number of processes, scaleup analyzes how both the problem size and the number of processes can grow while maintaining a constant amount of work per process.

The formula for scaleup is:

$$\text{Scaleup} = \frac{\text{Time for parallel execution with } P = 1}{\text{Time for parallel execution with varying } P \text{ (constant work)}}$$

In this scenario, we are dealing with a multidimensional problem where the size is determined by the values *n* and *m*.
Additionally, K may also be influenced by *m* if we set:

$$K = \sqrt{m/2}.$$

Several possible cases are analyzed below:

Case 1: *K* Independent of *m* (Constant), *n* Constant

If *K* is kept constant and n remains fixed, we can scale the problem size by increasing *m* proportionally to the number of processors *P*.

The scaleup is calculated as:

$$\text{Scaleup} = \frac{[m \times (3 \times n \times K + K + n + 1) + n \times K] \times I \times TFLOPS}{[P \times m \times (3 \times n \times K + K + n + 1) + n \times K] \times I \times TFLOPS}$$

Simplifying:

$$\text{Scaleup} = \frac{[m \times (3 \times n \times K + K + n + 1) + n \times K]}{[m \times (3 \times n \times K + K + n + 1) + n \times K]} = 1$$

In this scenario, the scaleup is **perfect**. As more processors are added and the problem size grows proportionally, the execution time remains constant.

Case 2: *n* Constant, *K* Varies with *m*

If *n* is fixed, but *K* changes with the size of *m*

$$\text{(e.g., } K' = \sqrt{P \times m/2}\text{)}$$

$$\text{Scaleup} = \frac{[m \times (3 \times n \times K + K + n + 1) + n \times K] \times I \times TFLOPS}{[P \times m \times (3 \times n \times K' + K' + n + 1) + n \times K'] \times I \times TFLOPS}$$

Simplifying:

$$\text{Scaleup} = \frac{[m \times (3 \times n \times K + K + n + 1) + n \times K]}{[m \times (3 \times n \times K' + K' + n + 1) + n \times K']}$$

Here, the scaleup is **strictly smaller than 1**. This is because the size of the problem grows faster than the number of processors, resulting in poor scalability.

Case 3: Balanced Growth

This scenario represents a middle ground between the two cases previously discussed. Here, we assume that K is independent of m, but its size increases as the problem size grows.
The same applies to n. In this case, m, n, and K all increase proportionally with the number of processors. The scaleup formulation becomes:

$$\text{scaleup} = \frac{[m \cdot (3 \cdot n \cdot K + K + n + 1) + n \cdot K] \cdot I \cdot TFLOPS}{\left[\frac{m \cdot (3 \cdot n \cdot K + K + n + 1)}{P} + n \cdot K\right] \cdot P \cdot I \cdot TFLOPS}$$

If m becomes sufficiently large:

$$\text{scaleup} \approx \frac{m \cdot (3 \cdot n \cdot K + K + n + 1)}{m \cdot (3 \cdot n \cdot K + K + n + 1)} = 1$$

In this more realistic scenario, we observe that as the dataset size grows significantly, the scaleup approaches 1. Conversely, when dealing with smaller datasets, the scaleup value falls below 1. This indicates that the marginal contribution of each additional process to the overall computation diminishes as the number of processes increases.

Conclusion

The theoretical analysis conducted in the previous sections highlights promising results in terms of both strong and weak scalability. These findings suggest that the parallelization approach should be effective in managing larger computational workloads.

The next step involves focusing on the practical implementation of the parallelization strategy. Once completed, experimental tests will be carried out to evaluate the real-world performance of our parallel **K-means** algorithm and compare it to the theoretical expectations.

Algorithm Description

In this section, we describe the parallel algorithm used for the clustering process. To make the explanation more intuitive, we use common data structures such as matrices and vectors, although in the code implementation, these are represented as objects due to the use of C++.

1. Initialization (SERIAL)

- The master process loads a dataset of appropriate size into a matrix $X \in \Re^{m \times n}$, where each of the m rows represents an observation with *n* features.
- The master creates a matrix $C \in \Re^{k \times n}$ by randomly selecting *k* points from the dataset. Each of the k rows represents a centroid with n features.
- The matrix *X* is partitioned into *P* submatrices *X′(k)*, each containing $m_p = m/P$ points.
- The matrix *C* is broadcasted to all *P* processes along with their respective data partitions. Each process receives a different partition of *X* (denoted as *X′(k)*).

At this point, the parallel operations can begin.

2. Distance Computation (PARALLEL)

Each process computes the squared Euclidean distance between each of the m_p points it received (the rows of *X′(k)*) and each centroid (the rows of *C*).

3. Minimum Computation (PARALLEL)

Each process selects, for each point x_i, the closest centroid *c(k)*, and assigns the point to the corresponding cluster.

4. Within Cluster Sum (PARALLEL)

Whenever a point is assigned to a centroid *c(k)*, the process updates a vector sumCluster(k), which accumulates the values of the points assigned to that centroid.

(Note: Since each process maintains its own sumCluster(k) for each cluster based on its *mp* elements, the total Sums for each cluster will be done serially by the master process after collecting the results from each process).

5. Counter Incrementation (PARALLEL)

Each time a point is assigned to a centroid *c(k)*, a scalar *num(k)* is updated by incrementing it by 1.

At the end, this scalar contains the number of elements in each cluster.

(Note: Similar to the previous step, the Sum of *num(k)* values will be carried out serially by the master process).

6. MSE in Cluster Sum (PARALLEL)

Using the minimum distances computed in the previous step, each process updates a scalar quantity Sum_distance by adding the minimum distance from each point to its assigned centroid.

Each process sends the following data to the master:

- **Matrix sumCluster_matrix**: Each row corresponds to a vector sumCluster(k) for a cluster.
- **Vector num**: Each element corresponds to a scalar num(k) for a cluster.
- **Scalar sum_distance**: The total sum of the minimum distances for each process.

7. MSE Total Sum (SERIAL)

The master process sums all the Sum_distance values received from the processes to obtain total_sum_distance. The master then divides this by mmm to compute the totalMSE, which is used to evaluate the termination condition. The algorithm decides whether to terminate or proceed to the next iteration.

8. Total Within Cluster Sum (SERIAL)

The master process sums all the rows of sum_cluster_matrix received from the processes to compute the total within-cluster Sum.

9. Total Counter Incrementation (SERIAL)

The master process sums all the num vectors received from the processes to compute total_num, which represents the total number of elements across all clusters.

10. Mean Computation (SERIAL)

The master computes the new centroids by calculating the ratio between the total sums of the clusters (from total_sum_cluster) and the total number of points in each cluster (from total_num). This produces a new matrix of centroids, and the next iteration begins.

This process continues iteratively until the termination condition is met.

In-depth Analysis of Speedup and Scaleup

Speedup Analysis

The parallelization approach proposed allows for an assessment of the algorithm's time complexity, which enables the estimation of the theoretical speedup that can be achieved. This analysis considers the additional operations introduced by MPI, providing a more accurate representation of the potential performance gains.

The speedup can be expressed by the following formula:

$$\text{Speedup} = \frac{\text{Execution time old}}{\text{Execution time new}}$$

Assumptions

- All floating-point operations take an equal amount of time, denoted as TFLOP
- The communication time during parallel execution is neglected in this estimation
- The computed speedup serves as an upper bound, as it only considers the computational time and a partial overhead due to parallelism. The full impact of communication overhead will be evaluated through experimental results

The execution time old (serial) algorithm is defined as:

Execution Time Old = $(3 \cdot m \cdot n \cdot K + m \cdot K + m \cdot n + n \cdot K + m) \cdot I \cdot TFLOP$

To calculate the execution time new (parallel), the parallel and serial components of the code must be considered separately.

Algorithm Phases and Costs

- Distance computation: $3 \cdot m \cdot n \cdot K$
- Minimum computation: $m \cdot K$
- Within-cluster Sum: $m \cdot n$
- Counter incrementation: -
- MSE in-cluster Sum: m
- MSE total Sum: $K \cdot P$
- Total within-cluster Sum: $K \cdot P$
- Mean computation: $n \cdot K$

Of these operations, the first five are performed in parallel, while the remaining ones are executed serially.

The execution time new (parallel) is calculated as:

$$\text{Execution Time new} = \left[\frac{m \cdot (3 \cdot n \cdot K + K + n + 1)}{P}\right] \cdot I \cdot \text{TFLOP}$$

The serial execution time is given by:

$$\text{Serial Execution Time} = [K \cdot (2 \cdot P + n)] \cdot I \cdot TFLOP$$

Finally, the overall speedup is expressed by the formula:

$$\text{Speedup} = \frac{m \cdot (3 \cdot n \cdot K + K + n + 1) + n \cdot K}{\frac{m}{P} \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 \cdot P + n)}$$

When m becomes large enough, the speedup approaches the number of processing units, P.

The $2 \cdot K \cdot P$ operations introduced by parallelization do not significantly affect the theoretical speedup when m is sufficiently large.

ScaleUp Analysis

In this analysis, we proceed similarly to the previous one but with different assumptions about how the problem size and the number of processors are related.

> **First scenario:** Assuming K is independent of m and remains constant, and n is also constant, we can increase the size of the problem by scaling m proportionally to the number of processors P.
>
> The formulation is as follows:
>
> $$\text{scaleup} = \frac{[m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 + n)] \cdot I \cdot \text{TFLOPS}}{[P \cdot m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 \cdot P + n)] \cdot I \cdot \text{TFLOPS}}$$
>
> This simplifies to:
>
> $$\text{scaleup} = \frac{[m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 + n)]}{[m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 \cdot P + n)]} < 1$$

In this scenario, we observe that scalability is not perfect. As processes are added (i.e., P increases), the denominator grows, causing the scalability to be less than 1. However, with large datasets, we expect scalability to approach 1.

Second scenario: If n is kept constant but K changes with the size of m, the formulation becomes:

$$\text{scaleup} = \frac{[m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 + n)] \cdot I \cdot \text{TFLOPS}}{[P \cdot m \cdot (3 \cdot n \cdot K' + K' + n + 1) + K' \cdot (2 \cdot P + n)] \cdot I \cdot \text{TFLOPS}}$$

where

$$K' = \sqrt{P \cdot m}$$

This simplifies to:

$$\text{scaleup} = \frac{[m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 + n)]}{[m \cdot (3 \cdot n \cdot K' + K' + n + 1) + K' \cdot (2 \cdot P + n)]} < 1$$

In this case, the scaleup is strictly smaller than 1, as the problem size increases at a faster rate than the increase in the number of processors. Furthermore, compared to the initial analysis, the scaleup is expected to worsen due to the operations introduced by parallelization, which increase the denominator,

where

$$K' = \sqrt{P \cdot m}$$

Third scenario: In this case, we assume that K is independent of m, but grows proportionally to the number of processors, while the other dimensions m and n also grow.

The formulation is as follows:

$$\text{scaleup} = \frac{\text{Execution Time for Parallel Execution with } P = 1}{\text{Execution Time for Parallel Execution Varying with } P}$$

This can be expressed as:

$$\text{scaleup} = \frac{[m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 + n)] \cdot I \cdot \text{TFLOPS}}{[P \cdot m \cdot (3 \cdot n \cdot K + K + n + 1) + K \cdot (2 \cdot P + n)] \cdot P \cdot I \cdot \text{TFLOPS}}$$

When m is large, the scaleup simplifies to:

$$\text{scaleup} \approx \frac{[m \cdot (3 \cdot n \cdot K + K + n + 1)]}{[m \cdot (3 \cdot n \cdot K + K + n + 1)]} = 1$$

In conclusion, we can say that the operations added by parallelization should theoretically have a minimal impact on the performance of our algorithm, except for the scenario where K is a function of m.

MPI Parallel implementation

As we found the main macro-operations of which the algorithm is composed, as well as their computational cost in FLOPS, we then designed a parallel implementation.

We implemented the algorithm in parallel using C++ because of the flexibility in structuring the code by objects having certain properties. In fact, to represent the data under which the algorithm works we implemented 4 classes: Tupla, Point, Centroid and Cluster.

We will have that:

- The Master divides the dataset into portions of points based on how many vCores we will have. It then proceeds to:
    - Send a portion of the dataset to all processes:
        - Broadcast the size of the portion of the dataset that they will have, via `MPI_Bcast`
        - The points of the portion to each process via `MPI_Send`
    - Broadcast the number of clusters and their initial centroids to all processes via `MPI_Bcast`
- Both the Master and the other processes will calculate the vector sum of the points in their clusters and return it as a total to the Master via `MPI_Reduce`
- The Master also receives the sum of the minimum distances of each point from the centroids in the clusters via `MPI_Reduce`
- Finally via `MPI_Bcast` the Master will send the termination condition of the algorithm to the other processes

Having defined our own data type to work on, we then implemented some methods for its serialization and deserialization, including for example:

```
void Cluster::serializeCluster(double* buffer){
    int K = Cluster::get_sclusters ();
    int dim = Cluster::get_cluster(0)->get_centroid()->get_dim();
    buffer[0] = K;        // Number of Clusters
    buffer[1] = dim;      // Centroid dimension
    Cluster::serializeCentroids(buffer + 2);
}
void Cluster::deserializeCluster(double* buffer) {
    int K = buffer[0];
    int dim = buffer[1];
    Cluster::create_clusters(K,dim);
    Cluster::deserializeCentroids(buffer + 2);
}
```

Each method takes as input a buffer that will be used for serialization and deserialization. The previous method, for example, will be used for serializing Clusters and will contain in the first two elements K and the size of the points. The next elements of the buffer, on the other hand, will contain the Centroids of each Cluster.

Other methods will be those for serializing and deserializing the vector sum of the points and the points to send.

Testing and Debugging

To test our project, we created a code snippet in Python that generates $\Re^{m \times n}$ matrices of random points.

```
import numpy as np
def write_matrix(M, N):
    matrix = np.random.randint(0, 10, (M, N))
    np.savetxt(f"DataSet{M}x{N}.txt", matrix, fmt = "%d", delimiter = ",")
write_matrix(100, 3)
print("Done!")
```

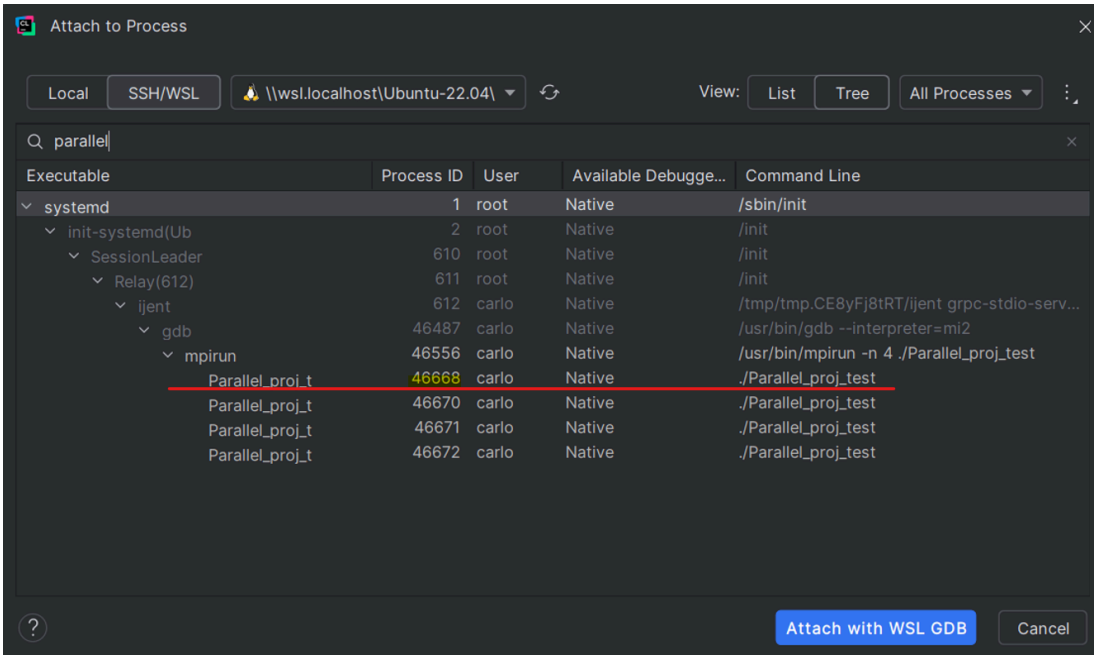
This allowed us to test our design on a wide range of datasets, observing execution times.

Another test that was done was about the correctness of the algorithm between Serial and Parallel implementations. It is important that the results at each iteration and final of the two versions are the same. We ensured this by taking the same initial centroids, using the same `seed` in the random function - `srand(<fixed_seed>)` - when choosing the points.

As for debugging, we used the CLion IDE which allowed us to easily debug the code in Serial and Parallel.

In Parallel, however, we had to implement a workaround to debug correctly:

1. Use a dummy_loop in the code block of a process and put a breakpoint in on that line
2. Start the debugger and 'attach' it to that process with Attach to Process such as:



Performance and scalability analysis

After testing and debugging the code locally, we then proceeded to test the final version in Google Cloud, creating two clusters of virtual machines: Fat and Light.

Google Cloud imposes some limits on both the number of IPs in the same area and the number of Cores available. Therefore, to have similar resources between the two clusters, we decided to set them up in this way:

		Tipo	
		Intra	Infra
Cluster	Fat	(2) e2-standard-8	
	Light	(8) e2-standard-2	

Using 2 e2-standard-8 machines for the Fat and 8 e2-standard-2 machines for the Light, so that in both clusters we would have 16 Cores and 64 Gb RAM in total.

We then plotted the results for the formulas:

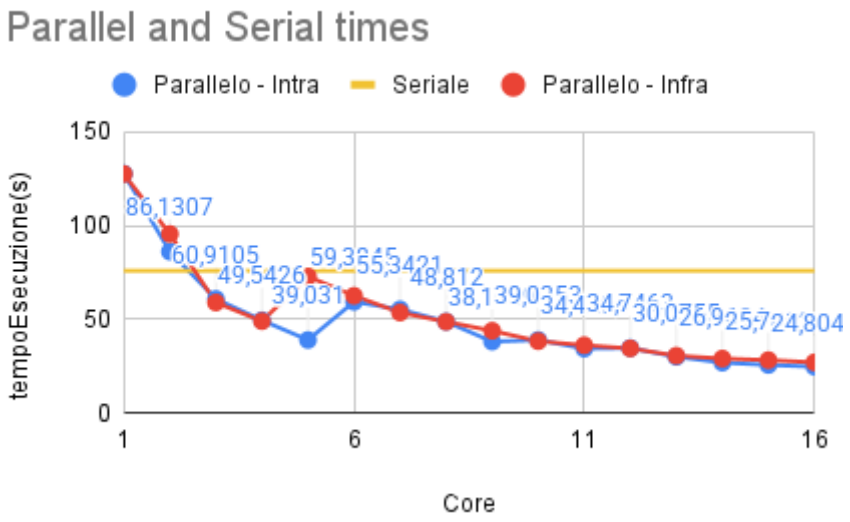
- $SpeedUp = \frac{Tempo\ Seriale}{Tempo\ Parallelo\ (P)}$
- $ScaleUp = \frac{Tempo\ Parallelo\ (1)}{Tempo\ Parallelo\ (P)}$
- $Efficiency = \frac{SpeedUp_P}{(P)}$

All tests were performed on a dataset consisted of a $\Re^{10'000\times10}$ matrix and setting MAX_ITER of the algorithm at 10.

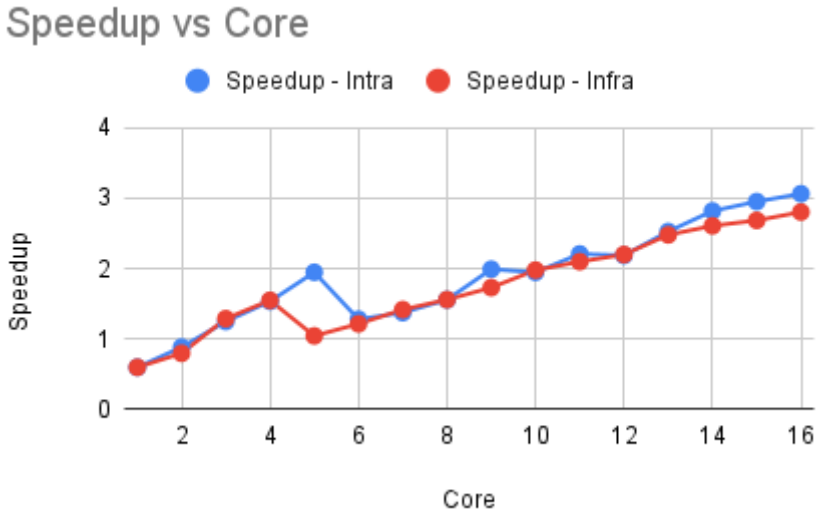
Fat Cluster

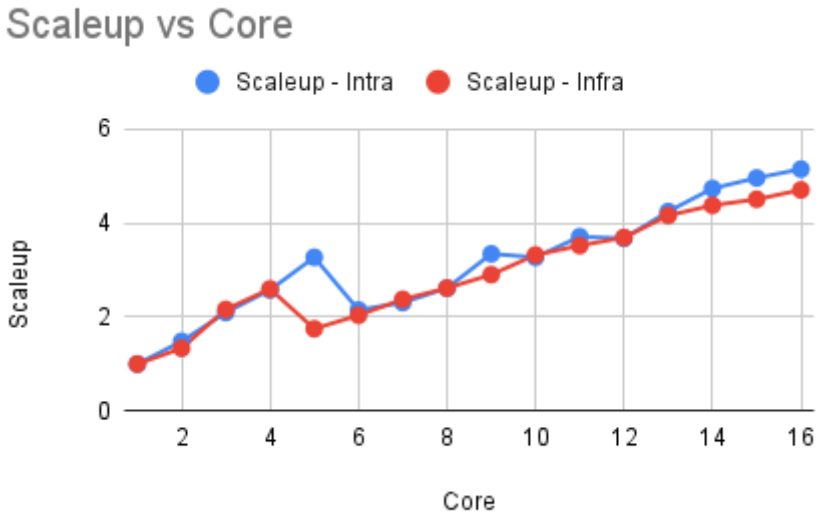
Formed by 2 machines located:

- Intra-Regional, in `us-central`.
- Infra-Regional, one in us-central and the other one in europe-west.

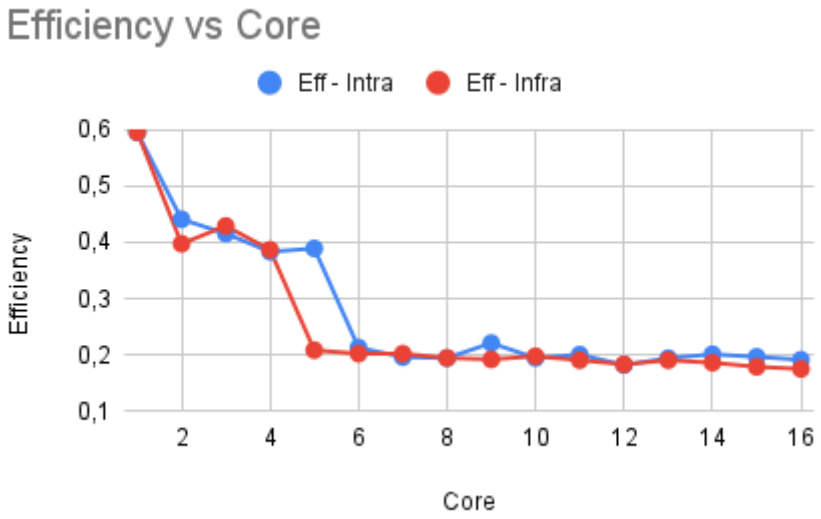


It can be seen that the execution time is reduced quite a bit compared to the Cores used, considering the `76s` taken by the Serial. It does not vary much between Intra and Infra, however the Intra-Regional cluster is only slightly faster.





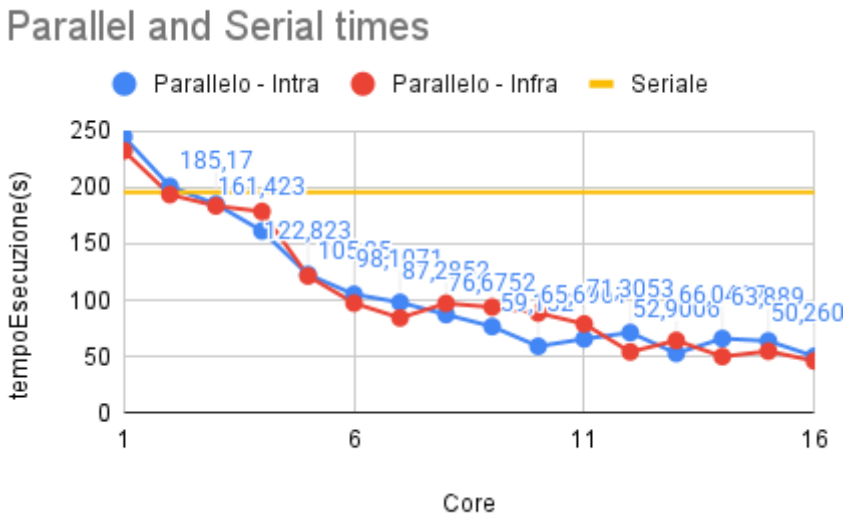
SpeedUp and ScaleUp (strong-scalability) confirm what can be seen from the first plot, having the Intra-Regional cluster slightly better.



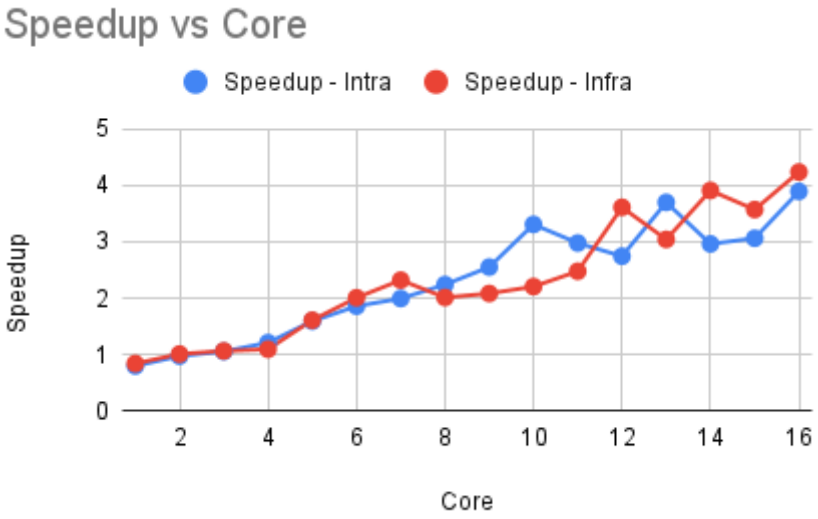
Light Cluster

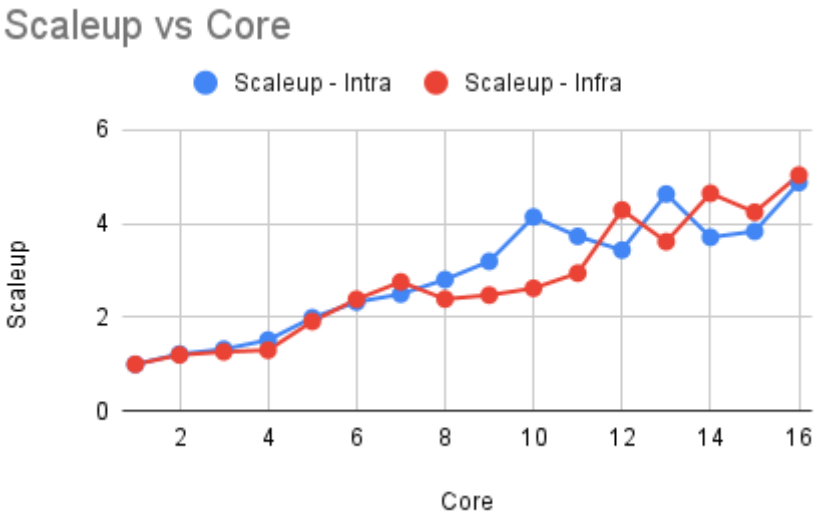
Formed by 8 machines located:

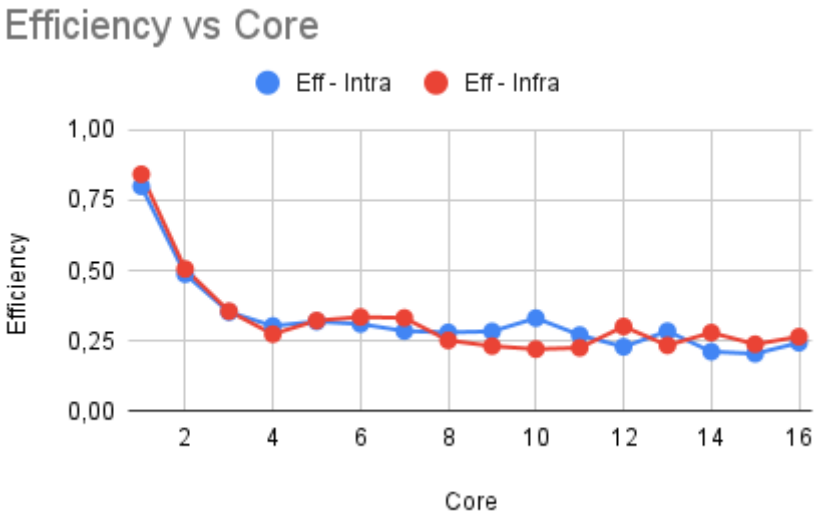
- Intra-Regional, all in us-central
- Infra-Regional, in pairs in: us-central, europe-west, asia-east and me-central



In this case the time in parallel is better since the machine used has few available resources, so in serial it will go quite badly (`195s`).


Speedup vs Core
Speedup - Intra
Speedup - Infra
Speedup
0
1
2
3
4
5
2
4
6
8
10
12
14
16
Core


Scaleup vs Core
Scaleup - Intra
Scaleup - Infra
Scaleup
0
2
4
6
2
4
6
8
10
12
14
16
Core


Efficiency vs Core
Eff - Intra
Eff - Infra
Efficiency
0,00
0,25
0,50
0,75
1,00
2
4
6
8
10
12
14
16
Core

Conclusion

We can see from the plots the differences between Fat and Light Clusters.

For instance, the execution time of Light Clusters tends to decrease but is not close to that returned by Fat Clusters. This may be due to the communication overhead between the machines located quite far apart.

Considering the SpeedUp achieved by both clusters, we can say that its growth is not satisfactory. In fact, as the number of Cores increased, we did not find a similar SpeedUp increase.

We can also state that the overall implementation of the algorithm could improve by going to:

1. reduce the number of times the processes and the Master pass bufferSize to each other for each data type, because by having a precise scheme of the algorithm its value cannot change between those steps
2. using vectors instead of lists to handle cluster points, because their handling is optimized while with lists it might take much longer to extract points

Contribution to the Project

From an initial view of the project, we identified two basic parts: the study of the algorithm with its parallelizable blocks and its implementation in C++.

The first was carried out by Alessio and the second by Carlo.

However, during the course of the project some doubts and problems occurred which were promptly discussed and solved by collaborating. So was done during the latest testing of the algorithm and its results in Google Cloud testing.