Summary

K-Means Clustering

The algorithm

General description

K-Means Clustering is an Unsupervised Machine Learning algorithm. As its name said it allows to partition a dataset of m elements into k cluster.
The algorithm is an iterative one: it starts taking a dataset of m elements and initializing a set of k centroids. At this point each of the m elements is classified to the closest centroid, thus k different clusters are created. In each cluster the mean between the elements is computed and this value is the new centroid of the cluster. The algorithm restart doing the same thing, until some termination condition is reached. The main goal of the method is to find that partition that minimizes the *Total Within Cluster Variance (TWCV).*

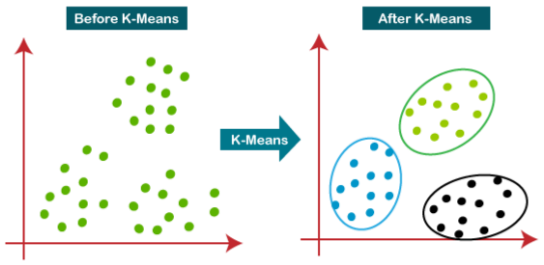


Figure 1 - K-Means with k=3 classes and 2-dimensional data (Srivastava, 2021)

It is important to underline that K-Means is a greedy algorithm, so it is not granted to find the global minimum, but just a minimum, this depends on how the *k centroids* are initialized. A first solution to this problem is to use some heuristics for centroids positioning, but there is no a way to initialize the centroids that guarantees the global minimum achievement. So, a common strategy is executing the algorithm many times and eventually take the best set of clusters.

Formal description

Let D = $\{ x^{(i)} \in \mathbb{R}^n \} \quad where \quad i = 1, \ldots, m$ be a dataset composed by m observations, each one which has dimension n.
Let K be the number of clusters we want to classify our data into.

- Initialize K centroids $\{c_k \in \mathbb{R}^n\} \quad where \quad k = 1, \ldots\ldots, K$
- Assign $x^{(i)}$ to the closest c_k to define the partition $\{Cluster_k\}$ $where \;\; k = 1, \ldots\ldots, K \;\; and$
- $\{x^{(i)} \; nearest \; to \; c_k\}$
- Update centroids taking the mean of contained elements with the goal of finding that set of clusters such that is minimized:

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

Initialization and Termination

As stated in the General Description, before to start with the analysis of the algorithm, it is necessary to define the way the centroids are initialized, and which is the termination condition which is used. In fact, basing on the chosen method the algorithm will be different, thus it cost will change as well.

INITIALIZATION

Here there are 2 main things to decide: the number of clusters and their position.

- The number of clusters
  As regards this point, there are several approaches in literature for selecting the right number. A general review of the methods is proposed by Kodinariya and Makwana in their publication (Kodinariya, 2013). For our project we chose to use a simple Rule of Thumb: K $= \sqrt{m/2}$

- The initial values of the centroids
  Although there are many proposals for the optimal selection of centroids in literature, for this project we decided to choose randomly the values for the centroids, simply by taking K values randomly from the dataset. The research of a more efficient method could be object of future research.

TERMINATION

As regard the termination we followed this idea: since we want to MINIMIZE TWCV, we can stop when the Total Mean Square Error stops changing. Stated in a more formal way:

$$REPEAT \quad until \quad TMSE^{(j\ iteration)} < TMSE^{(j-1\ iteration)}$$

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

Moreover, since algorithm convergence speed is influenced by the initial position of the centroids, it could take a while to reach the condition on MSE, so a further condition on the maximum iterations number *MaxIterations* is required.
At the end the algorithm termination is the following:

$$REPEAT \quad until$$

$$TMSE^{(j\ iteration)} < TMSE^{(j-1\ iteration)} \qquad and \qquad N\_Iterations \ < \ MaxIterations$$

Serial Algorithm - pseudocode

INITIALIZATION

Take a dataset $D = \{ x_{(i)} \in \mathbb{R}^n \} \quad where \quad i = 1,\dots,m$
Initialize a set of K centroids $C_{(k)} = \{ c_{(k)} \} \quad where \quad k = 1,\dots,K$
Initialize value for $MaxIterations$
Set $MSE\_prec = big\ number$

EXECUTION

```
    For $i = 1,\dots,m$

        For $k = 1,\dots,K$

            $ComputeDistance\ d(x_{(i)}, c_{(k)}) = \| x_{(i)} - c_{(k)} \|_2^2$
        End

        $Select\ k\ such\ that\ d(x_{(i)}, c_{(k)})\ is\ minimum$
        $Assign\ x_{(i)}\ to\ cluster\ Cluster_{(k)}$
        $sum\_Cluster_{(k)} = sum\_Cluster_{(k)} + x_{(i)}$
        $IncreaseCounter\ num_{(k)}\ of\ Cluster_{(k)}$
        $MSE = MSE + d(x_{(i)}, c_{(k)})$
    End

    For $k = 1,\dots,K$

        If $num_{(k)} == 0$
           $Assign\ closest\ point\ x'\ from\ another\ Cluster\ to\ Cluster_{(k)}$
           $num_{(k)} = 1$
           $sum\_Cluster_{(k)} = x'$
        End

        $c_{(k)} = sum\_Cluster_{(k)} / num_{(k)}$
    End

    $MaxIterations = MaxIterations + 1$
```

TERMINATION

```
    If
     $MSE > MSE\_prec \ || \ N\_Iterations > MaxIterations \quad then \quad TERMINATE$

    Else

       $MSE\_prec = MSE$
       $REPEAT$
```

Computational complexity analysis

Since the K-means clustering is usually executed with many datapoints is important to have an idea of the time it will require to complete. It is crucial to understand how the execution time changes as a function of the input dimension. To do that we decided to analyze the algorithm, considering the number of floating-point instructions that are executed in the three main part: Initialization, Execution and Termination.

<u>NOTE</u>: The values produced in the iteration j, are those used for starting the iteration j+1. So, each iteration can start only after the previous has been completed and each iteration has the same number of operations executed. The analysis is made considering only ONE iteration. The total number of operations can be easily obtained multiplying the number of instructions in one iterations time the number of iterations.

SYMBOL	MEANING
K	Number of clusters
m	Number of datapoints
n	Dimension of each datapoint
x	Datapoint
c	Centroid
$Cluster$	Cluster
d_{ik}	Distance between $x_{(i)}$ and $c_{(k)}$
$sum\_C_{(k)}$	Vectorial sum of the points in Cluster $C_{(k)}$
P	Number of Processes in MPI
I	Number of iterations of the K-means
T_{FLOPS}	Execution time of floating operation

INITIALIZATION

In this part we have just to upload the dataset into our program and assign values to some variables, as is stated in the paragraph "Serial algorithm – pseudocode". Since the goal of this project is to parallelize in an efficient way the execution of the algorithm, the initialization strategy is not going to be changed by the parallelization, remaining the same in both circumstances.

EXECUTION

This is the most important part of the algorithm and indeed the most demanding in term of number of operations to be executed. To analyze this part, we divided the algorithm execution in many phases and studied separately each phase.

- Distance computation

  $Assume\ x, c \in \mathbb{R}^n$
  $i\ indicates\ the\ number\ of\ an\ observation\ x_{(i)}, so\ max(i) = m$
  $k\ indicates\ the\ number\ of\ a\ centroid\ c_{(k)}, so\ max(k) = K$

  The formula we must solve, since we use squared Euclidean distance is the following:

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

Dividing the formula into chunks and studying separately the number of iterations for each chunk we have:

CHUNK	NUMBER OPERATIONS
$x_{(i)} - c_{(k)}$	n
$\|\ldots\|_2^2$	*1 + 1 + n + n ≈ 2*n*
$\forall k$	K
$\forall i$	m
TOTAL OPERATIONS	*3*m*n*K*

- Minimum computation

As regard the minimum computation we must find the closest centroid for each point:

$$\forall i, find\ k\ such\ that\ d_{ik}\ is\ minimum$$

CHUNK	NUMBER OPERATIONS
$find\ k\ such\ that\ d_{ik}\ is\ minimum$	K
$\forall i$	m
TOTAL OPERATIONS	*m*K*

- Within cluster summation

We must sum together all the points in a cluster and do that for each cluster. So, we eventually have the sum of all the points in our dataset, which is composed by m points. Since each point is n-dimensional we have:

TOTAL OPERATIONS	*m*n*

- Counter incrementation

The total number of operations for the counter incrementation would be m, however they are integer operations and since this type of operations is only a small fraction respect to the floating-point operations number, their impact on the time complexity is not very significant for our analysis.
Thus, we decided to discard them and consider the floating-point operations to study the time complexity of the algorithm.

- Mean computation

We have seen that the upper part of the algorithm does the within cluster summation and the counter incrementation. So, the mean computation is just the division between these two quantities which are already available at this point of the algorithm. So, we just need to compute:

$$\forall k \quad c_{(k)} = \frac{sum\_C_{(k)}}{num_{(k)}}$$

CHUNK	NUMBER OPERATIONS
$sum\_C_{(k)}$ / $num_{(k)}$	*n*
$\forall k$	*K*
TOTAL OPERATIONS	*n*K*

- Termination

The termination condition adopted in this project is:

$$REPEAT \quad until$$

$$TMSE^{(j\ iteration)} < TMSE^{(j-1\ iteration)} \quad and \quad N\_Iterations < MaxIterations$$

The condition on the N_Iterations is made just by using a counter and for the same reasons as before we will not consider it. The condition on the TMSE could seem to be time demanding, however at this point of the algorithm we already have all the information we need, because we computed them at the distance computation phase. Thus, the MSE is just the sum of the minimum distances from their centroid of each single point.

TOTAL OPERATIONS	*m*

- TOTAL SERIAL COMPUTATIONAL COMPLEXITY (TIME)

Finally, it is possible to stick together all the operations required by 1 execution of the serial algorithm and the result is the following:

TOTAL OPERATIONS	*3*m*n*K + m*K + m*n + n*K + m*

$$Since\ k >> 1, n >> 1, m >> n, k$$

TOTAL OPERATIONS	*≈ 3*m*n*K*

We are considering only floating-point operations, so assuming T_{FLOP} is the time that a single floating-point operation requires for execution, and I is the total number of iterations of the algorithm, we end up with:

COMPUTATIONAL TIME COMPLEXITY	*≈ (3*m*n*K) * T_{FLOP} * I*

Parallelizable blocks

INFRA-BLOCK PARALLELIZATION

In this section we analyze the blocks of the algorithm, with the goal of finding out dependencies inside each block and thus, identify those algorithm's sections that can be parallelized and those that cannot be.

Distance Computation

$\| x_{(i)} - c_{(k)} \|_2^2$

The algorithm computes the distance between each point and each centroid. So, the output of one computation is not used by the following, cause this latter is performed on a different data-point or on different centroid. Thus, we can say that inside a global iteration of the algorithm, no dependencies among distances are present and this section can be parallelized.

Minimum Computation

Given the distances computed at the previous step, the algorithm just takes the minimum distance for each data-point. Thus, also in this section there are no dependencies among the minimum of different data-points and the section can be parallelized.

Within Cluster Summation

Here the algorithm sums together the points that belong to the same cluster. So, this section is parallelizable as well. For example, it is possible to sum half of the elements with one process and the other half with another process and eventually sum the output together. However, is evident the necessity of a synchronization strategy between processes. Same reasoning applies to the *counter incrementation* section.

Mean Computation

$$c_{(k)} = \frac{sum\_C_{(k)}}{n_{(k)}}$$

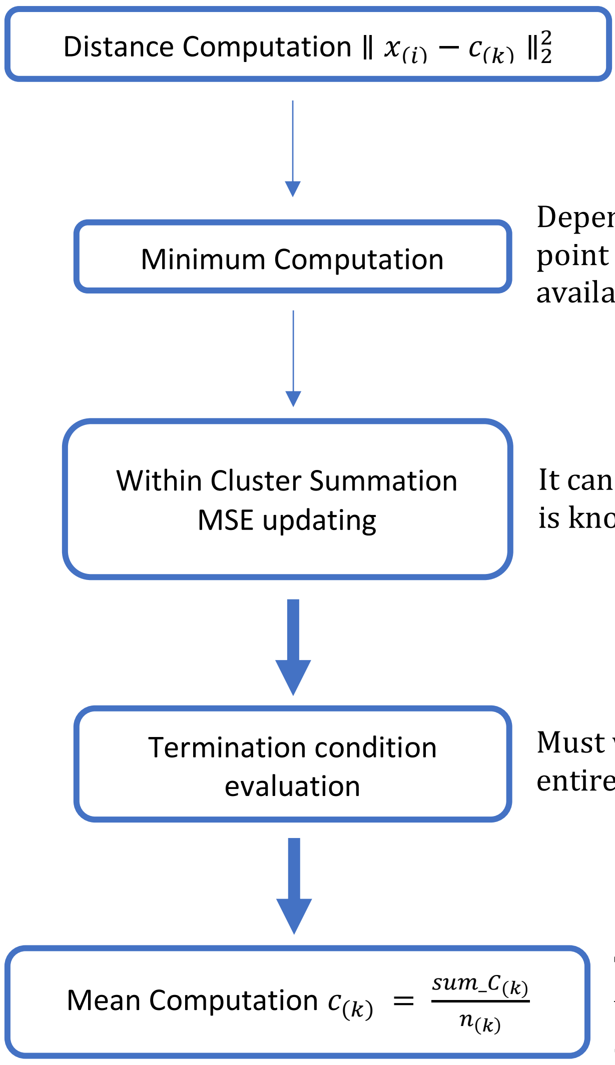
The mean computation is just a division, as explained above. The result of each division is independent from the others. So, this section is parallelizable as well.

Termination

The computational cost of termination is given by the summation of the MSE in each cluster. So, the same reasoning explained for the within cluster summation applies here as well.

INTRA-BLOCKS PARALLELIZATION

Until now we studied the parallelism available inside each algorithm section, but this is not all. We still miss to analyze the dependencies between the sections.


Distance Computation $\| x_{(i)} - c_{(k)} \|_2^2$
Minimum Computation
Within Cluster Summation
MSE updating
Termination condition evaluation
Mean Computation $c_{(k)} = \frac{sum\_C_{(k)}}{n_{(k)}}$

Assume all needed data have been uploaded
Algorithm can start with distance computation

Depends on the previous section only for each single point x(i). It can start when $\{d_{ik}\ for\ k = 1, \ldots., K\}$ is available.

It can start as soon as the cluster where x(i) belongs to is known. So as soon as minimum finish working on x(i).

Must wait the previous sections to have elaborated the entire dataset.

This section must wait for the previous sections to have elaborated the entire dataset. So, it is serial to the others.

From this analysis what comes out is that the first three sections must respect a serial order, because they need their "parent's" output. However, they can be parallelized across the dataset because they basically do the same operations on each datapoint. As regard the last two sections they cannot be executed in parallel with the former three because they need these latter results. In fact, the condition evaluation and the mean computation requires the entire dataset has already been processed. This means that in hypothetical parallel algorithm at this point a synchronization between processes in necessary and until the synchronization is not done the algorithm execution is stalled. A detail to be underlined is that, even if the mean computation cannot be performed until all dataset is processed, its execution can be parallelized, as explained in the paragraph: Intra block parallelization.
So, in the end we can have the following situation for each phase:

OPERATION	EXECUTION
Distance computation	PARALLEL
Minimum computation	PARALLEL
Within cluster summation	PARALLEL
Counter incrementation	PARALLEL
MSE in cluster summation	PARALLEL
Mean computation	SERIAL

Now we must have an idea of the theoretical performance of this parallelization strategy and if it will be satisfactory, we can go deeper inside it.
To assess the performance, we use theoretical speedup and scaleup.

First analysis of the speedup and the scaleup

SPEEDUP STUDY USING AMDAHAL'S LAW

A first estimation of the speedup we can obtain exploiting the available parallelism could be obtained using the Amdahl's Law.

$$Speedup = \frac{1}{(1 - parallel\ fraction) + \frac{parallel\ fraction}{number\ of\ processes\ (P)}}$$

$$parallel\ fraction = \frac{number\ of\ instructions\ of\ parallelizable\ blocks}{total\ number\ of\ instructions}$$

To compute the parallel fraction, we need to understand which code's sections are going to be parallelized and those are not going to be parallelized in the original serial code. The table below is used for that.

Assumptions: all the floating-point operations require the same execution time T_{FLOP}, the time used for the communication in the parallel execution is not considered. Doing that the speedup can be intended as the maximum speedup that can be reached, disregarding the overhead introduced by the MPI parallel implementation.

OPERATION	EXECUTION	SERIAL COST (N° FLOPS)
Distance computation	PARALLEL	*3*m*n*K*
Minimum computation	PARALLEL	*m*K*
Within cluster summation	PARALLEL	*m*n*
Counter incrementation	PARALLEL	-
MSE in cluster summation	PARALLEL	*m*
Mean computation	SERIAL	*n*K*

NUMBER PARALLEL INSTRUCTIONS	*[m*(3*n*K + K + n + 1)]*
TOTAL NUMBER INSTRUCTIONS	*[m*(3*n*K + K + n + 1) + n*K]*

At this point is it possible to compute the parallel fraction of the serial code:

$$parallel\ fraction = \frac{[m * (3 * n * K + K + n + 1)]}{[m * (3 * n * K + K + n + 1) + n * K]}$$

$$if\ m\ is\ big \quad \rightarrow \quad parallel\ fraction \approx \frac{[m * (3 * n * K + K + n + 1)]}{[m * (3 * n * K + K + n + 1)]} = 1$$

$$Speedup = \frac{1}{(1 - 1) + \frac{1}{number\ of\ processes\ (P)}} \approx P$$

We can conclude saying that for large dataset we expect a speedup which is almost equal to the number of processes. However, we are not considering the overhead introduced by the parallelization. A partial consideration of these aspects is done in the next chapter and the complete overhead's effect will come out with the experimental results.

SCALEUP ANALYSIS

Another analysis that can be done, is that of the scaleup. In fact, the speedup analyzes how a fixed size problem execution time changes when more processes are added. While the scaleup changes both the number of processes and the problem's size while keeping constant the amount of work for each process. The formula for the scaleup is the following:

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

In our case we have a multidimensional problem, in which the problem size is given by the values *n* and *m*. This latter also influences *K* if we set *K = sqrt(m/2).* We can eventually have many different situations, here some of them are analyzed:

- Assuming *K* independent from *m* and kept constant, *n* constant, we can increase the size of our problem by increasing *m* proportionally to the number of processors. The formulation is:

$$scaleup = \frac{[m * (3 * n * K + K + n + 1) + n * K] * I * T_{FLOPS}}{\left[\frac{P * m * (3 * n * K + K + n + 1)}{P} + n * K\right] * I * T_{FLOPS}}$$

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

In this case the scaleup is perfect, adding processor and enlarging the problem the execution time is always the same.

- If instead, we consider *n* kept constant but *K* changing with the size of *m*, the formulation becomes:

$$scaleup = \frac{[m * (3 * n * K + K + n + 1) + n * K] * I * T_{FLOPS}}{\left[\frac{P * m * (3 * n * K' + K' + n + 1)}{P} + n * K'\right] * I * T_{FLOPS}} \qquad K' = \sqrt{(P * m)/2}$$

$$= \frac{[m * (3 * n * K + K + n + 1) + n * K]}{[m * (3 * n * K' + K' + n + 1) + n * K']} < 1$$

In this second situation we have that the scaleup is strictly smaller than 1, since the size of our problem is increasing in a faster way respect to the processors' number growth. The problem in this case scales in a poor way.

- This is the middle point between the two just discussed. Here we assume that *K* is independent from *m*, but we assume its dimension grows when we have a bigger problem size. The same holds for *n.* So, *m, n, K* are all increasing with the growth in the number of processors. The formulation is the following:

$$scaleup \;=\; \frac{[m * (3 * n * K + K + n + 1) + n \; * \; K\,] * I * T_{FLOPS}}{[\,\frac{m * (3 * n * K + K + n + 1)}{P} \; + \; n \; * \; K]\; P * \; I \; * \; T_{FLOPS}}$$

$$If \; m \; is \; big \qquad \rightarrow \qquad scaleup \;\approx\; \frac{[m * (3 * n * K + K + n + 1)]}{[m * (3 * n * K + K + n + 1)]} \;=\; 1$$

So, in this last, more realistic situation we expect that when we have lot of data, the scaleup tends towards 1, while when we have few data, its value is smaller than 1. This means that the marginal contribution to computation by each new process is going to decrease.

CONCLUSION

By parallelizing the algorithm in the way explained in the previous sections, we obtained good theoretical results as regard both, strong scalability and weak scalability. We can now start thinking more in detail about how that type of parallelization can be deployed and once all is done perform some experimental test with the aim of obtaining the actual performance of our parallel K-means implementation.

Parallel Algorithm

Basing on the results obtained from the previous study we are going to introduce our parallelization proposal.

Parallelization Proposal

In figure 2 there is a representation of what we said until now. It is no intended as a flow chart, but it is just a schematization about how algorithm works. The communication strategy is going to be discussed later.

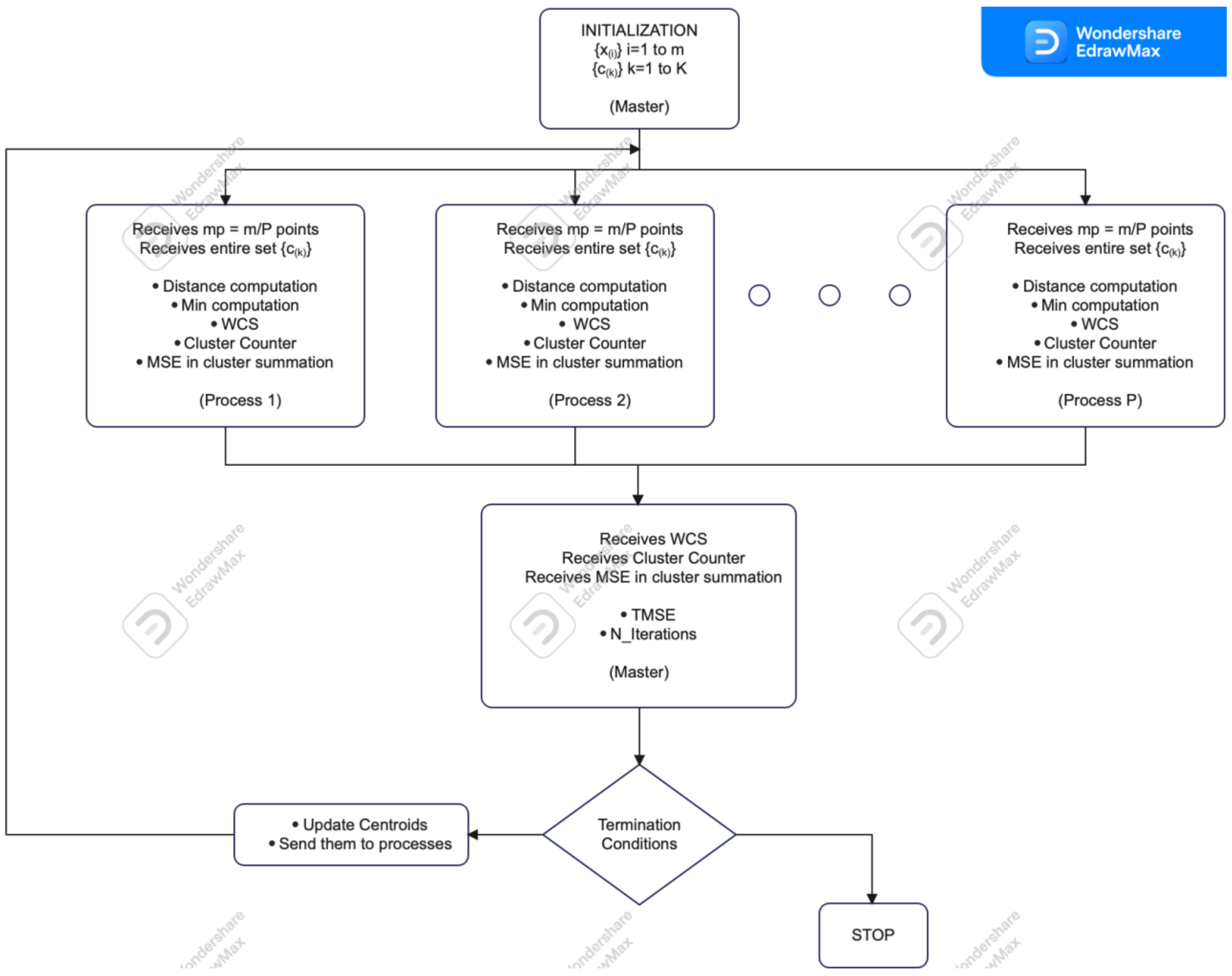


Figure 2 - Parallelization proposal

ALGORITHM DESCRIPTION

This is the description of the algorithm. To make it more intuitive we refer to well-known structures as matrices and vectors. However, as we will see in the code implementation they are replaced by objects because we used C++.

- Initialization (SERIAL)
  A dataset of proper cardinality is loaded by the master into a matrix $\boldsymbol{X} \in \mathbb{R}^{m*n}$. Each of the m rows is an observation with dimension n.
  Master also creates a matrix $\boldsymbol{C} \in \mathbb{R}^{k,n}$ taking randomly k observations from the dataset. Each of the k rows is a centroid with dimension n.
  The matrix ***X*** is partitioned into **P** submatrices $\boldsymbol{X'_{(k)}}$, each one containing $m_p = m / P$ points.
  The matrix ***C*** is broadcasted to all **P** processes together with a partition of the data. Each process receives a different partition of **X** ($\boldsymbol{X'_{(k)}}$).
  Each process can now start working in parallel with the others doing the following operations:

  - Distance computation (PARALLEL)
    Each process computes the squared Euclidean distance between each of the m_p points received (rows of $\boldsymbol{X'_{(k)}}$) and each centroid (rows of ***C***).

  - Minimum computation (PARALLEL)
    Each process selects for each point $x_{(i)}$ the closest centroid $c_{(k)}$. Each point is then assigned to a cluster.

  - Within cluster summation (PARALLEL)
    Each time a point is assigned to a centroid $c_{(k)}$, a vector $sum\_cluster_{(k)}$ is updated, summing to the already present quantity the just added point.
    (Note: since each process has its own $sum\_cluster_{(k)}$ for each cluster based on its m_p elements, the total summations for each cluster will be done in serial way by the master once it will have obtained the values from each process).

  - Counter incrementation (PARALLEL)
    Each time a point is assigned to a centroid $c_{(k)}$, a scalar $num_{(k)}$ is updated by summing 1. At the end this latter contains the number of elements in each cluster.
    (Note: for the same reason as the point before, there is a serial part carried out by master).

  - MSE in cluster summation (PARALLEL)
    Using the minimum distances found at the minimum computation phase, a scalar quantity *Sum_distance* is updated, by summing for all points their minimum distance to their centroid.

    Each process sends to master:
    - matrix *sum_cluster_matrix.* Each row is the vector $sum\_cluster_{(k)}$
    - vector *num.* Each element is the scalar $num_{(k)}$
    - scalar *sum_distance*

  At this point master does:

- MSE total summation (SERIAL)
  Master sums all *Sum_distance* received obtaining *total_sum_distance* and divides it by *m*, the output is thus the *Total_MSE*. At this point it evaluates termination condition and decides if to terminate or proceed through next steps.

- Total within cluster summation (SERIAL)
  Master sums over the rows all the *sum_cluster_matrix* received by processes.

- Total counter incrementation (SERIAL)
  Master sums over the rows all the *num* vectors received obtaining *total_num.*

- Mean computation (SERIAL)
  Compute the ratio between rows of *total_sum_cluster* and rows of *total_num.* The result is matrix, and each row is a new centroid value $c_{(k)}$. Now a new iteration starts.

Communication and Synchronization Schema

In our parallel algorithm each process works on its own subset of data-points by doing the already explained operations. When a process finishes its job must send the results to the master process, because it is the master that takes care about the unification of the results of each process, to perform the condition evaluation and the updating of the centroids, to set-up all what is needed to start a new iteration of the algorithm. The master process requires the entire dataset to be elaborated before to start working, so here a synchronization between processes is necessary.

Keeping the same parallelization strategy, we made different implementation as regard the communication strategy.

FIRST VERSION

In the first version we used the standard MPI Point to Point communication functions: MPI_Send and MPI_Recv.

In Figure 3 is reported the schema used for:

- Sending Points
- Sending Clusters information
- Sending Updated Centroids
- Sending finish Condition

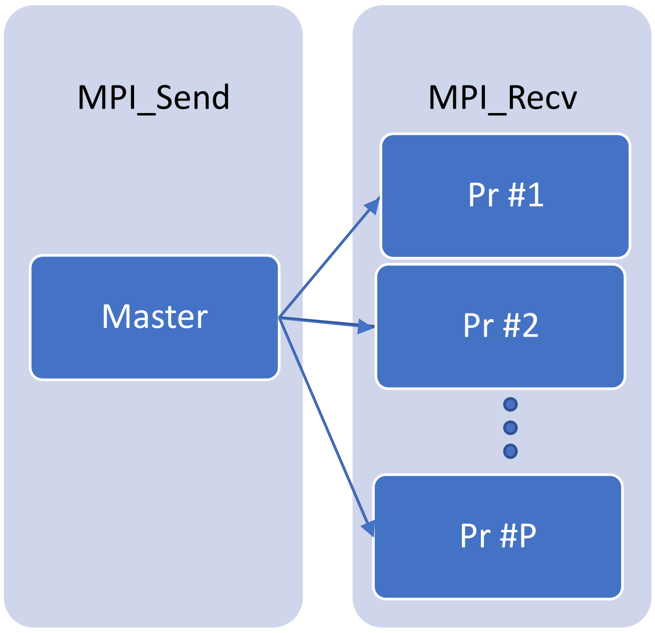


Figure 3 - Communication schema n.1

In Figure 4 is reported the schema used for:

- Sending sum and number of points in each cluster
- Sending the Sum_distance

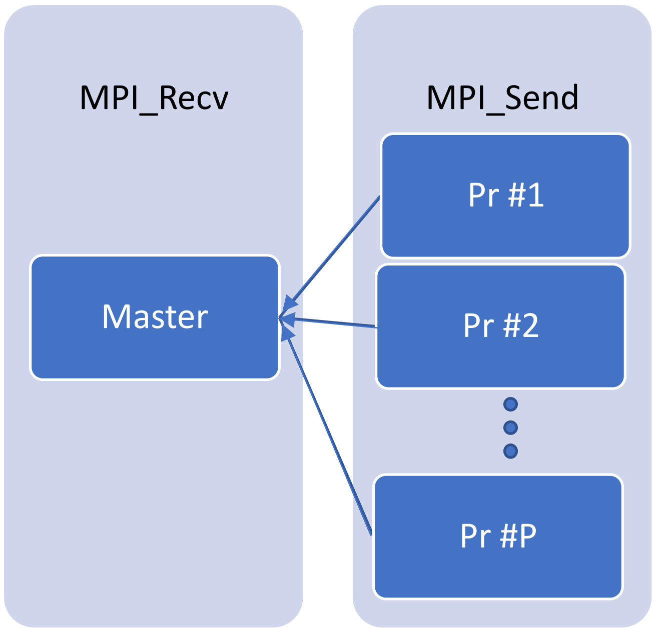


Figure 4 - Communication schema n.2

SECOND VERSION

Looking at the communication strategy and at how the algorithm works, we can see that the master does a lot of work, and everything pass through it. Moreover, using the standard point to point functions is not convenient when the number of processes grows. We have identified a possible alternative, that exploits the MPI Layer to carry out some computations and the collective communication functions to speed up the communication time.
The changes apported basically involve the communication strategy we used, the logic behind the algorithm is kept unaltered.

In Figure 5 is reported the schema used for:

- Sending points

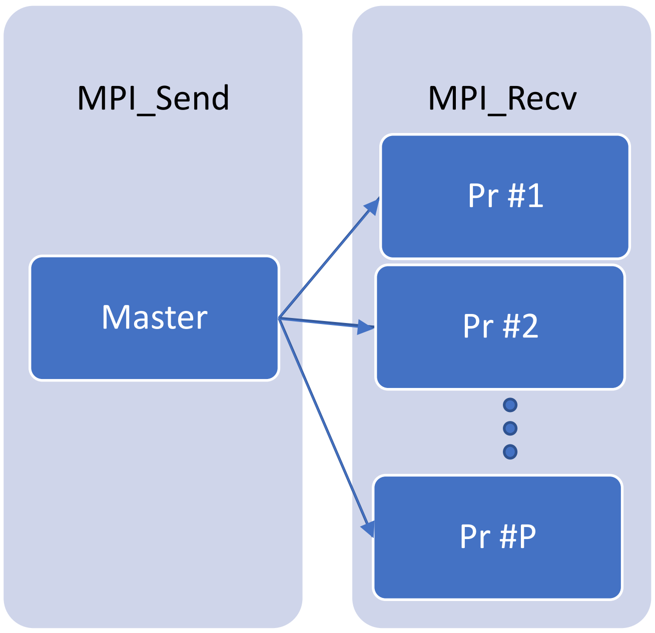


Figure 5 - Communication schema n.3

In Figure 6 is reported the schema used for:

- Sending clusters information
- Sending updated centroids
- Sending finish condition

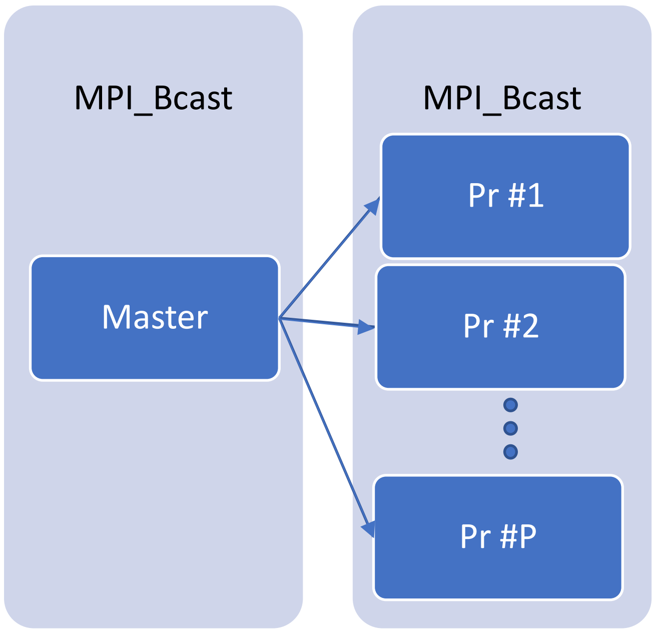


Figure 6 - Communication schema n.4

In Figure 7 is reported the schema used for:

- Sending sum and number of points in each cluster
- Sending the Sum_distance

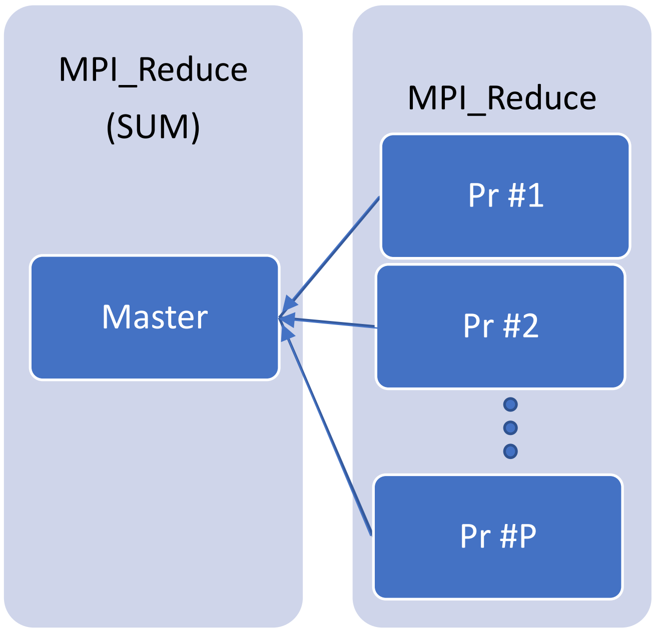


Figure 7 - Communication schema n.5

This second communication strategy is the one we used for going on in our study and for the experimental results with Google Cloud.

Deeper analysis of the speedup and the scaleup

SPEEDUP ANALYSIS

Given the parallelization proposed it is possible to study its time complexity and then compute the theoretical speedup which can be reached, taking into consideration also the number of additional operations that the use of MPI induced in the algorithm. The speedup is estimated using the formula:

$$speedup = \frac{Execution\ time\ old}{Execution\ time\ new}$$

<u>Assumptions</u>: all the floating-point operations require the same execution time T_{FLOP}, the time used for the communication in the parallel execution is not considered. Doing that the speedup can be intended as an upper bound to the speedup that can be reached, considering the computational time of the parallel implementations and a partial overhead, in the sense that the number of additional operations to manage parallelism are considered but the communication time is disregarded and will come out in the experimental results.

The *execution time old* has already been computed previously:

EXECUTION TIME OLD	$(3*m*n*K + m*K + m*n + n*K + m) * I * T_{FLOP}$

To compute the *execution time new*, it is necessary to consider separately the sections computed in parallel from those still computed in a serial way. In the following table all the algorithm's phases are reported together with their serial cost and their modality of execution (serial or parallel).

OPERATION	EXECUTION	SERIAL COST (N° FLOPS)
Distance computation	PARALLEL	*3*m*n*K*
Minimum computation	PARALLEL	*m*K*
Within cluster summation	PARALLEL	*m*n*
Counter incrementation	PARALLEL	-
MSE in cluster summation	PARALLEL	*m*
MSE total summation	SERIAL	*K*P*
Total within cluster summation	SERIAL	*K*P*
Total counter incrementation	SERIAL	-
Mean computation	SERIAL	*n*K*

PARALLEL EXECUTION TIME	$[m*(3*n*K + K + n + 1) / P] * I * T_{FLOP}$
SERIAL EXECUTION TIME	$K*(2*P + n) * I * T_{FLOP}$

In conclusion we have:

$$speedup = \frac{m * (3*n*K + K + n + 1) + n*K}{\frac{m}{P} * (3*n*K + K + n + 1) + K * (2*P + n)} \approx P\ \ if\ m\ big\ enough$$

Regard the speedup we can say that the *2*K*P* operations introduced by parallelization don't alter the theoretical speedup when m is big enough.

SCALEUP ANALYSIS

The scaleup is analyzed in the same way we did before.

1. Assuming K independent from m and kept constant, n constant, we can increase the size of our problem by increasing m proportionally to the number of processors. The formulation is:

$$scaleup = \frac{[m * (3 * n * K + K + n + 1) + K * (2 + n)] * I * T_{FLOPS}}{\left[\frac{P * m * (3 * n * K + K + n + 1)}{P} + K * (2 * P + n)\right] * I * T_{FLOPS}}$$

$$= \frac{[m * (3 * n * K + K + n + 1) + K * (2 + n)]}{[m * (3 * n * K + K + n + 1) + K * (2 * P + n)]} < 1$$

Contrary to the case in which we didn't consider the overhead, we have that the scalability is not perfect. In fact, when we add processes, we increase P and so the denominator grows making the scalability < 1. However, when we have lots of data, we can expect scalability like 1.

2. If instead, we consider n kept constant but K changing with the size of m, the formulation becomes:

$$scaleup = \frac{[m * (3 * n * K + K + n + 1) + K * (2 + n)] * I * T_{FLOPS}}{\left[\frac{P * m * (3 * n * K' + K' + n + 1)}{P} + K' * (2 * P + n)\right] * I * T_{FLOPS}}$$

$$where\ K' = \sqrt{P * m}$$

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

In this second situation we have that the scaleup is strictly smaller than 1, since the size of our problem is increasing in a faster way respect to the processors' number growth. Moreover, with respect to the initial analysis we expect an even worst scaleup since the number of operations introduced to manage parallelization are going to have a considerable impact on the denominator where $K' = \sqrt{P * m}$.

3. Here we assume that K is independent from m, but we make it grows proportionally to the number of processes, together with the other two dimensions m and n. The formulation is the following:

$$Scaleup = \frac{Time\ for\ parallel\ execution\ with\ P = 1}{Time\ for\ parallel\ execution\ varying\ P}$$

$$scaleup = \frac{[m * (3 * n * K + K + n + 1) + K * (2 + n)] * I * T_{FLOPS}}{[\frac{m * (3 * n * K + K + n + 1)}{P} + K * (2 * P + n)] * P * I * T_{FLOPS}}$$

$$If\ m\ is\ big \rightarrow scaleup \approx \frac{[m * (3 * n * K + K + n + 1)]}{[m * (3 * n * K + K + n + 1)]} = 1$$

In conclusion we can say that the operations added by the parallelization, theoretically should not significantly influence the performance of our algorithm, except for the case of the scaleup in which K is function of m.

To see a graphical representation of the above formulae:

- Figure 8 – Speedup plots (Note that for the speedup plots $K = \sqrt{m / 2}$)
- Figure 9 – Scaleup plots case 1
- Figure 10 – Scaleup plots case 2
- Figure 11 – Scaleup plots case 3

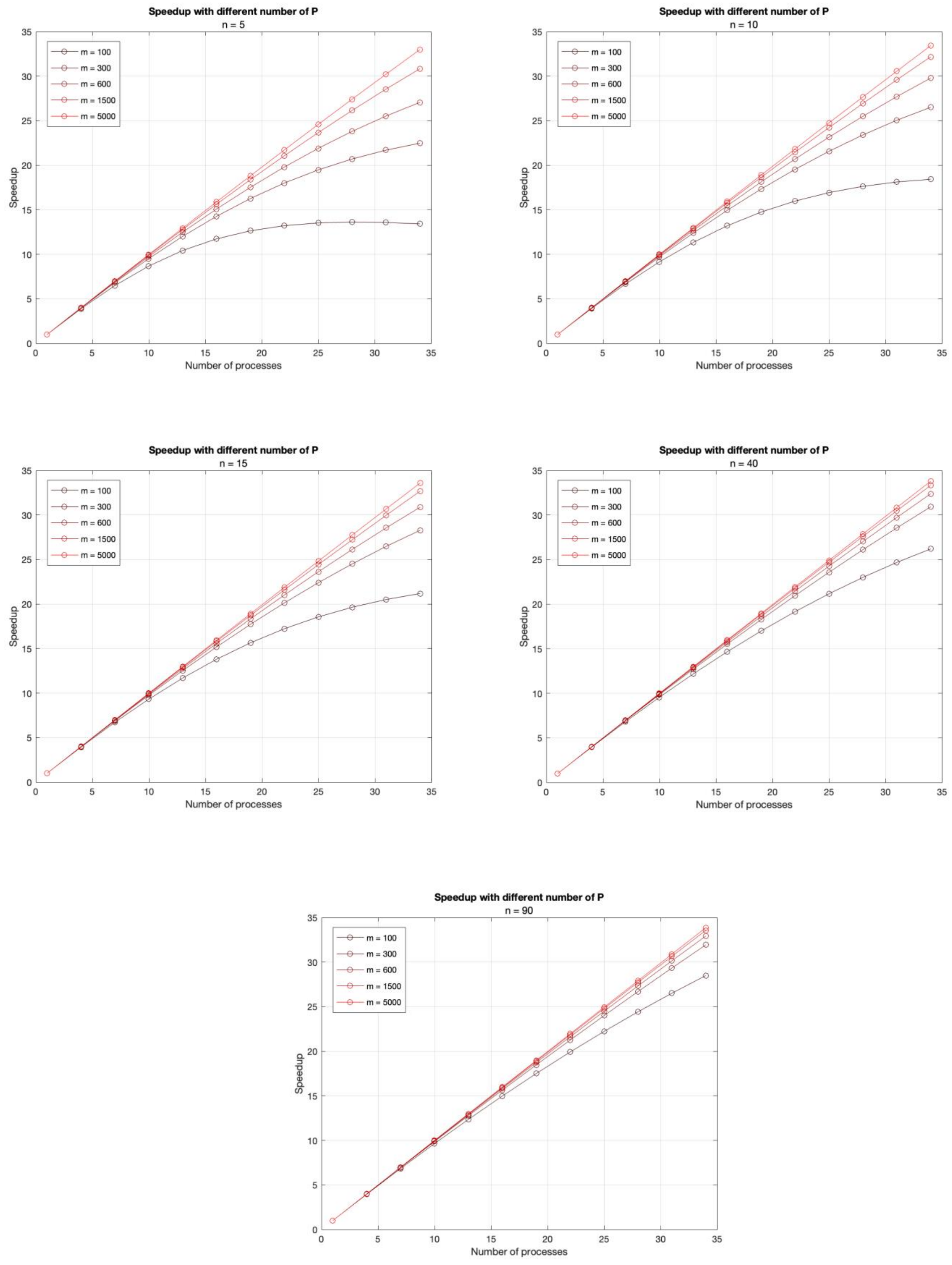


Figure 8 - Speedup plots. Realized with Matlab.

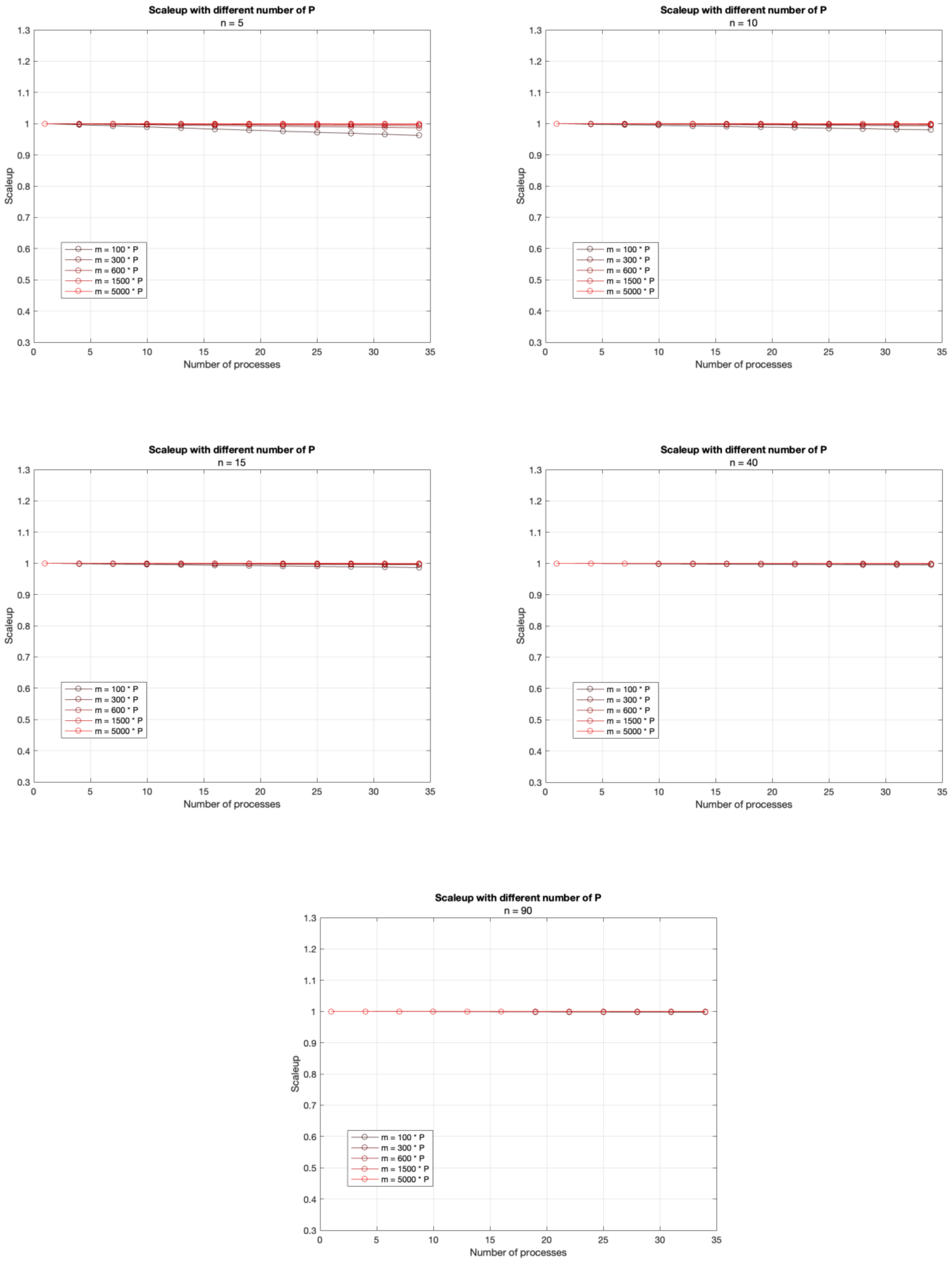


Figure 9 - Scaleup plots case 1. Realized with Matlab

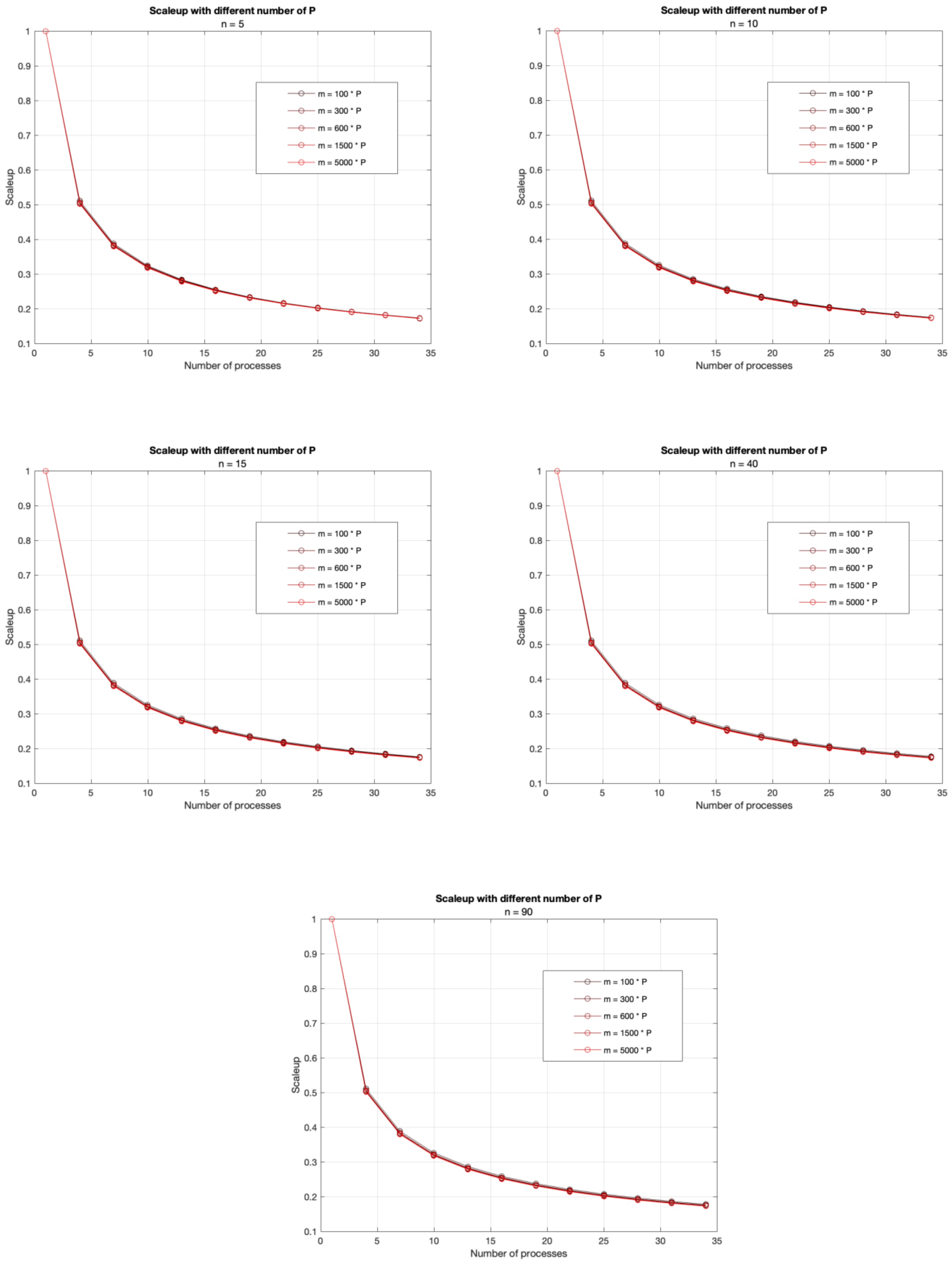
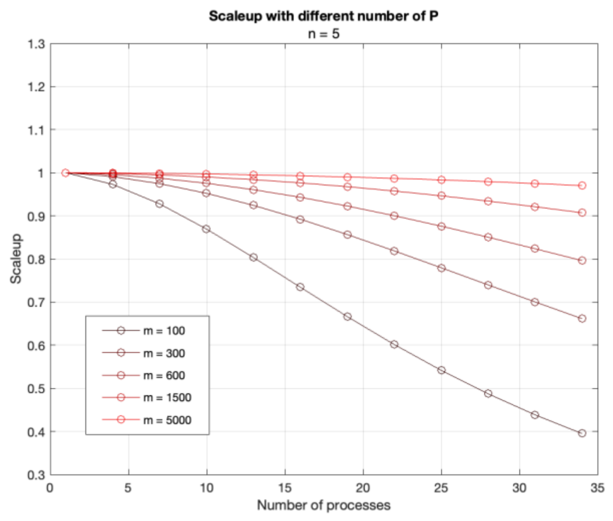
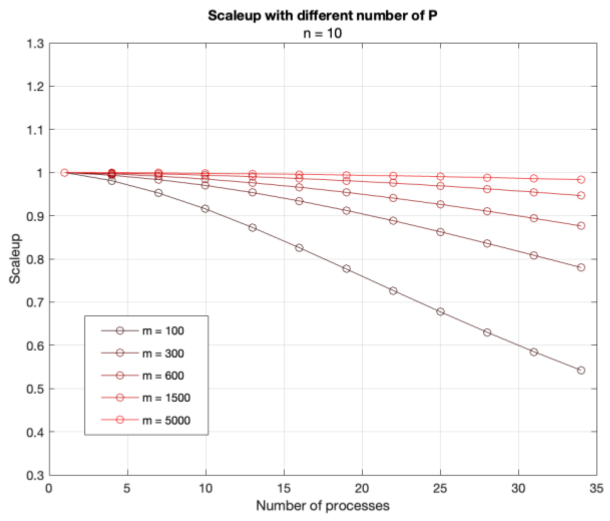
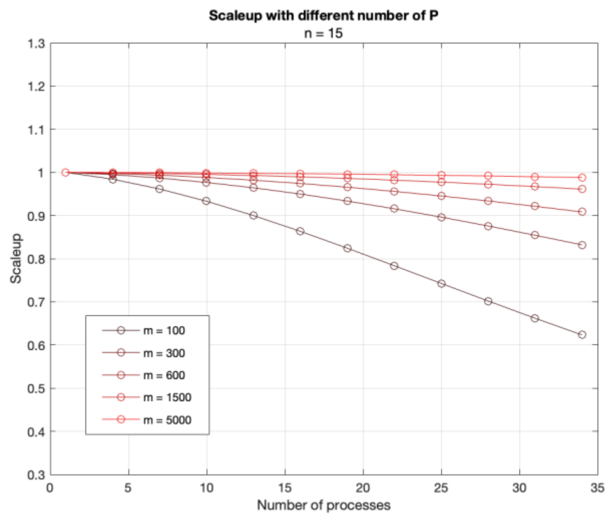
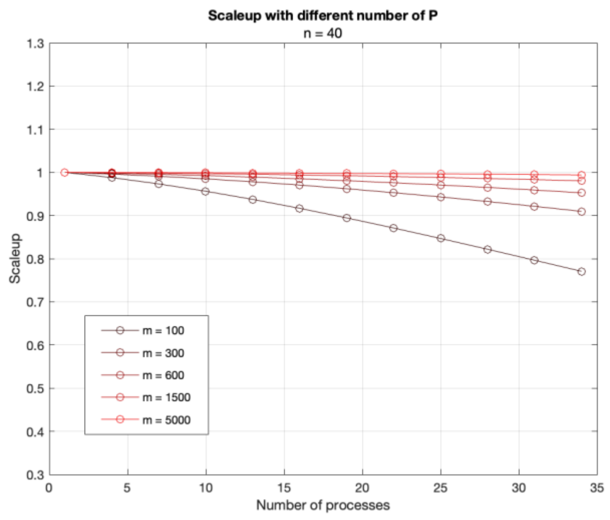


Figure 10 - Scaleup plots case 2. Realized with Matlab.









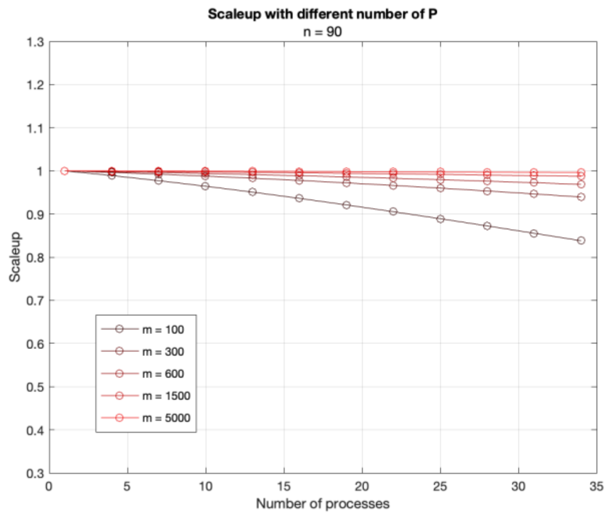


Figure 11 - Scaleup plots case 3. Realized with Matlab.

In general, we can say that for a big size problem ($m \geq 5000$) the speedup is almost linear with a value that is almost equal to the number of P and the scaleup follows the same trend and its value is close to 1, except for the case 2. Obviously, these are just theoretical measures because they take into consideration just a fraction of the overhead caused by the parallel program, for example the interaction with the memory subsystem (the parallel algorithm might cause more cache misses) or the synchronization and communication time between the different processes.

MPI Parallel Implementation

Why Object Oriented?

MPI (Message Passing Interface) is a widely used library for parallel programming on distributed memory systems. In this implementation, an object-oriented approach was used to design the code, with C++ as the programming language. The object-oriented design of the code allows for a modular and flexible structure, making it easy to add new features and maintain the code.

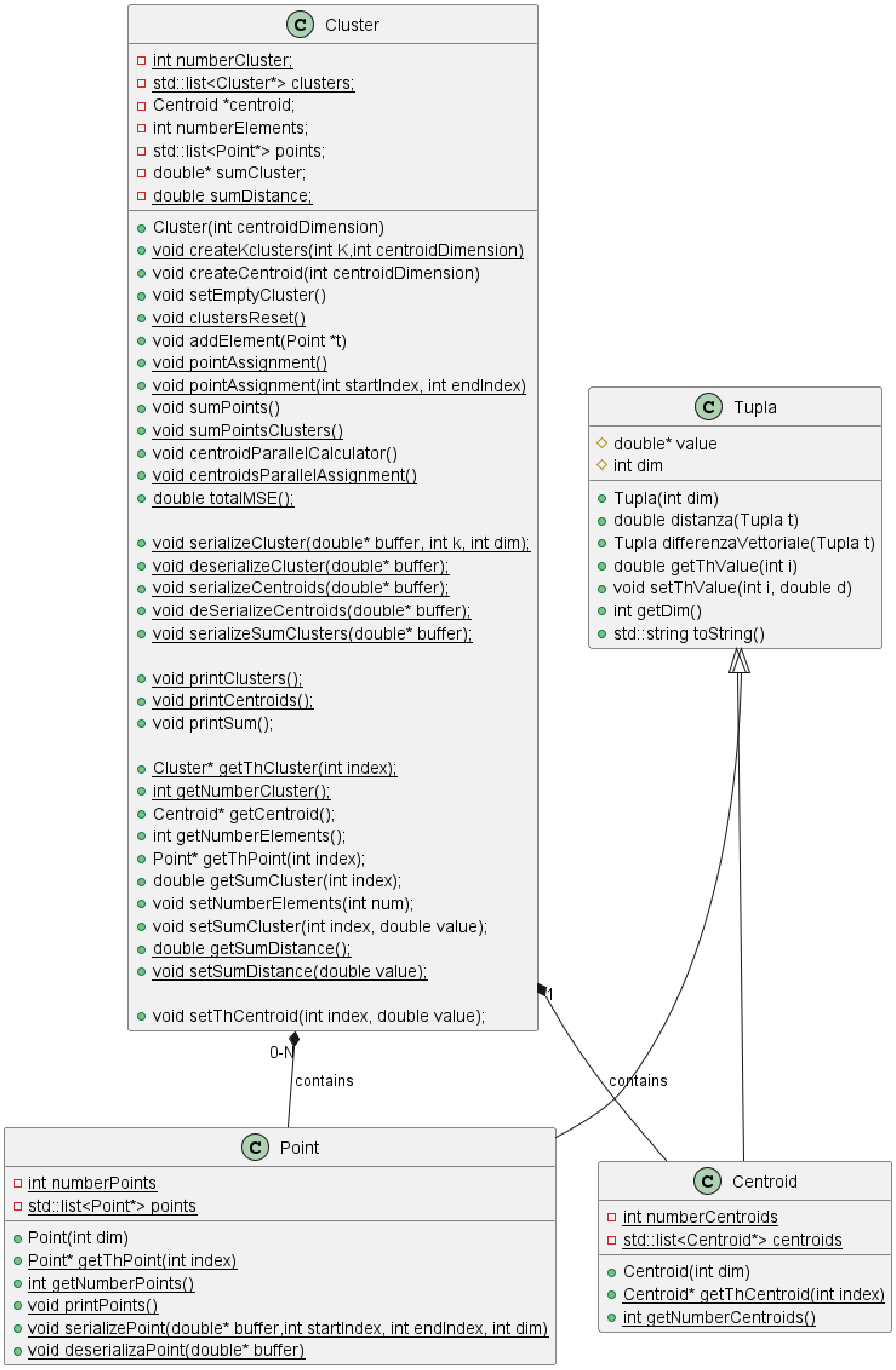

Cluster
int numberCluster;
std::list<Cluster*> clusters;
Centroid *centroid;
int numberElements;
std::list<Point*> points;
double* sumCluster;
double sumDistance;
Cluster(int centroidDimension)
void createKclusters(int K,int centroidDimension)
void createCentroid(int centroidDimension)
void setEmptyCluster()
void clustersReset()
void addElement(Point *t)
void pointAssignment()
void pointAssignment(int startIndex, int endIndex)
void sumPoints()
void sumPointsClusters()
void centroidParallelCalculator()
void centroidsParallelAssignment()
double totalMSE();
void serializeCluster(double* buffer, int k, int dim);
void deserializeCluster(double* buffer);
void serializeCentroids(double* buffer);
void deSerializeCentroids(double* buffer);
void serializeSumClusters(double* buffer);
void printClusters();
void printCentroids();
void printSum();
Cluster* getThCluster(int index);
int getNumberCluster();
Centroid* getCentroid();
int getNumberElements();
Point* getThPoint(int index);
double getSumCluster(int index);
void setNumberElements(int num);
void setSumCluster(int index, double value);
double getSumDistance();
void setSumDistance(double value);
void setThCentroid(int index, double value);
Tupla
double* value
int dim
Tupla(int dim)
double distanza(Tupla t)
Tupla differenzaVettoriale(Tupla t)
double getThValue(int i)
void setThValue(int i, double d)
int getDim()
std::string toString()
1
0-N
contains
contains
Point
int numberPoints
std::list<Point*> points
Point(int dim)
Point* getThPoint(int index)
int getNumberPoints()
void printPoints()
void serializePoint(double* buffer,int startIndex, int endIndex, int dim)
void deserializaPoint(double* buffer)
Centroid
int numberCentroids
std::list<Centroid*> centroids
Centroid(int dim)
Centroid* getThCentroid(int index)
int getNumberCentroids()

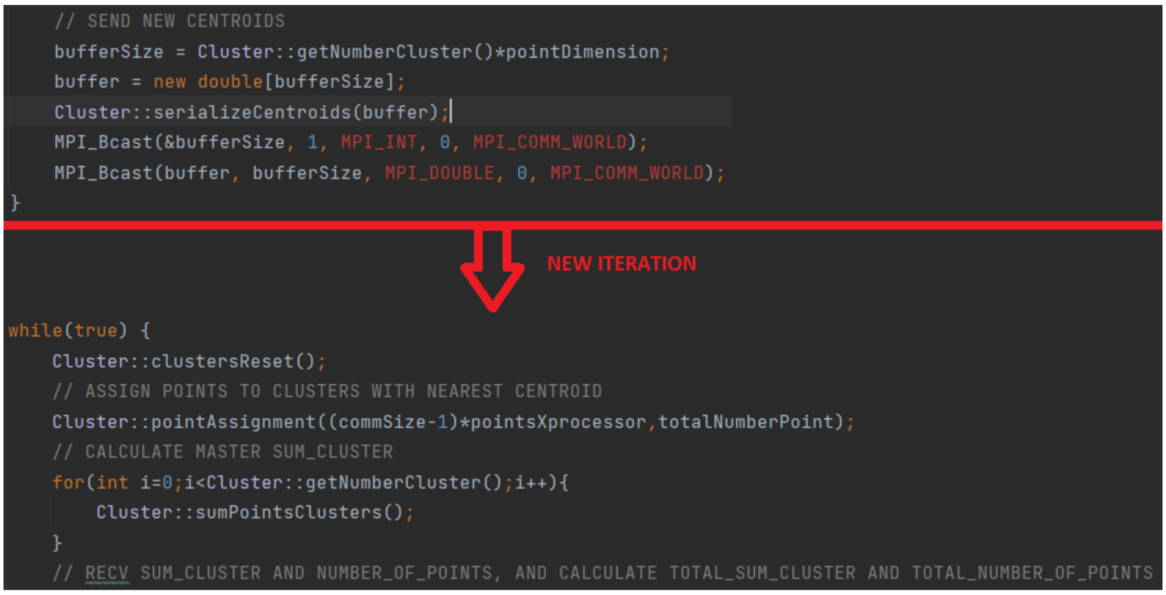
Figure 12 - Class Diagram

Why C++?

C++ was chosen as the programming language due to its support for object-oriented programming and its ability to handle large and complex data structures. It also provides a wide range of libraries and tools for parallel computing, making it a good choice for implementing MPI.
Moreover, it is faster than python in execution.
On the other hand, the price to pay is that of having a larger number of lines with respect to higher level languages.

Why synchronous communication?

The reason for our choice is that the benefits of using an asynchronous method would be minimal in our situation. This is the case in which there is a possibility to use MPI_Ibcast:



```
    // SEND NEW CENTROIDS
    bufferSize = Cluster::getNumberCluster()*pointDimension;
    buffer = new double[bufferSize];
    Cluster::serializeCentroids(buffer);
    MPI_Bcast(&bufferSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(buffer, bufferSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

```
while(true) {
    Cluster::clustersReset();
    // ASSIGN POINTS TO CLUSTERS WITH NEAREST CENTROID
    Cluster::pointAssignment((commSize-1)*pointsXprocessor,totalNumberPoint);
    // CALCULATE MASTER SUM_CLUSTER
    for(int i=0;i<Cluster::getNumberCluster();i++){
        Cluster::sumPointsClusters();
    }
    // RECV SUM_CLUSTER AND NUMBER_OF_POINTS, AND CALCULATE TOTAL_SUM_CLUSTER AND TOTAL_NUMBER_OF_POINTS
```

Figure 13 - Code Snapshot

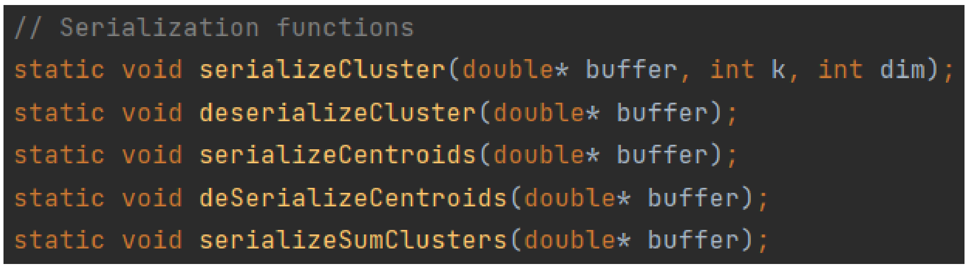
This is the section of code in which the master sends the new centroids to slaves and then goes on with its computation. However, the slave, in parallel, computes the same task on its own data. So, we can expect that both will be completed at the same time.
For this reason, we decided to use MPI_Bcast which allowed us for simpler debugging.

Serialization

Since we defined new structures do not present in the standard MPI data types, we implemented some serialize and deserialize functions to manage communication.
Each function creates a buffer of doubles containing different information.

- serializeCluster(): it contains the number of clusters (K), the centroids' dimension and values. Used by Master;
- deserializeCluster(): it generates the cluster from buffer's information. Used by Slave;
- serializeCentroids(): it contains the centroids' information. Used by Master;

- deserializeCentroids(): used to update the centroid's information; Used by Slave;
- serializeSumClusters(): it contains the partial sum of the points in each cluster and their number; Used by Slave;

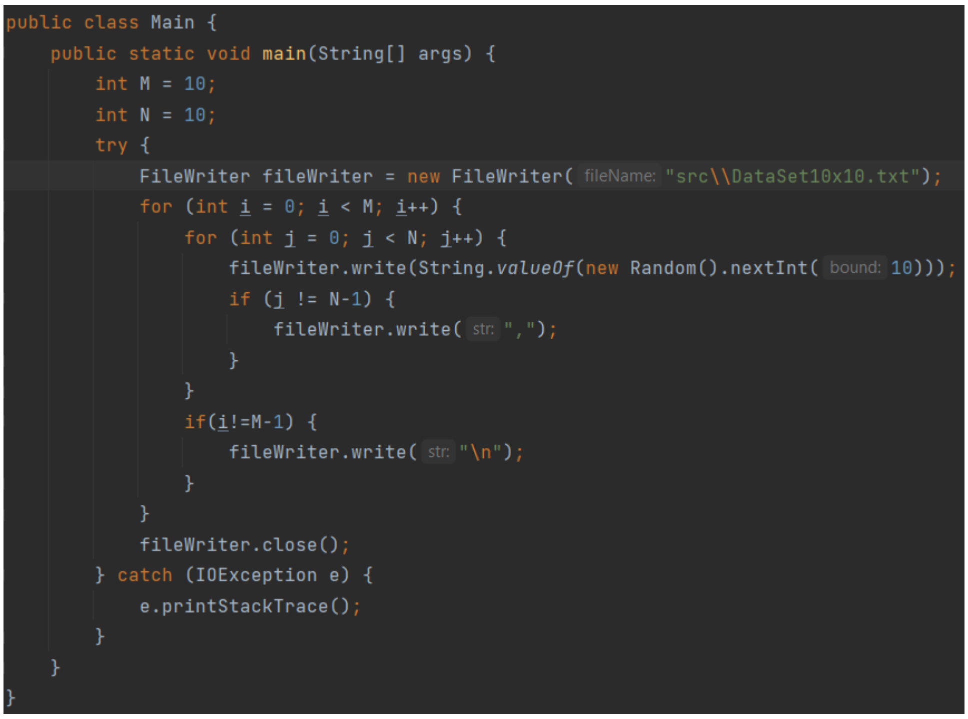


```
// Serialization functions
static void serializeCluster(double* buffer, int k, int dim);
static void deserializeCluster(double* buffer);
static void serializeCentroids(double* buffer);
static void deSerializeCentroids(double* buffer);
static void serializeSumClusters(double* buffer);
```

Figure 14 - Code snapshot 2

Testing and Debugging

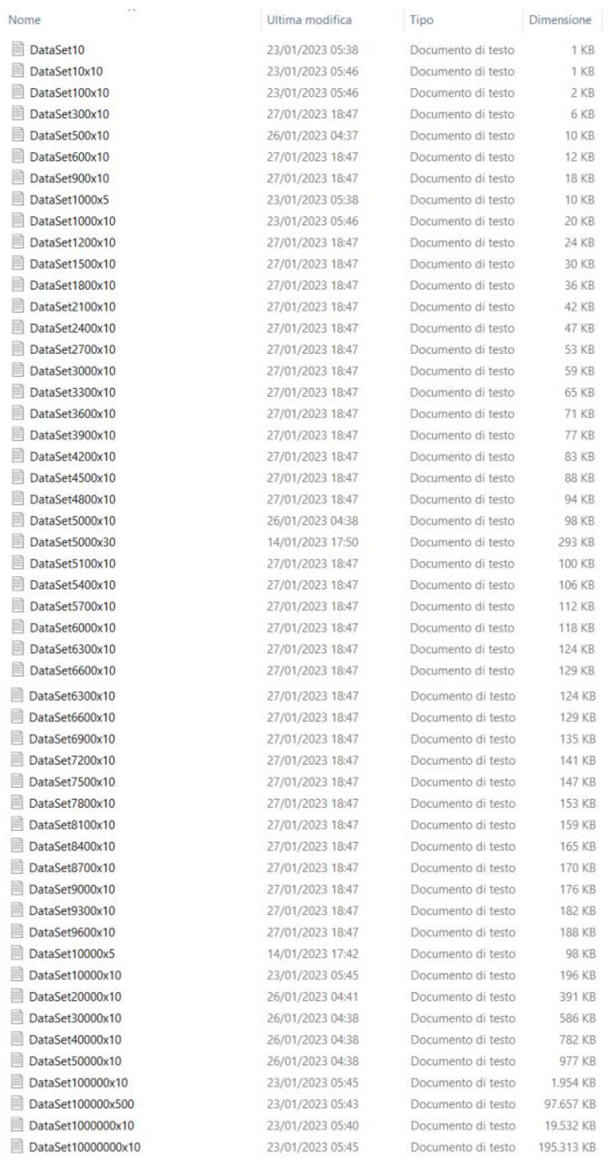
To test that our program was working properly we created a small program in Java with the goal of generating a dataset.



```
public class Main {
    public static void main(String[] args) {
        int M = 10;
        int N = 10;
        try {
            FileWriter fileWriter = new FileWriter( fileName: "src\\DataSet10x10.txt");
            for (int i = 0; i < M; i++) {
                for (int j = 0; j < N; j++) {
                    fileWriter.write(String.valueOf(new Random().nextInt( bound: 10)));
                    if (j != N-1) {
                        fileWriter.write( str: ",");
                    }
                }
                if(i!=M-1) {
                    fileWriter.write( str: "\n");
                }
            }
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 15 - Java program

We generated and tested our program using the following different datasets. The name of each dataset is "DataSet m x n" where m is the number of data and n is their dimension.
Due to the high computational time, we could not perform the test "DataSet1.000.000x10" and "DataSet10.000.000x10".



Nome	Ultima modifica	Tipo	Dimensione
DataSet10	23/01/2023 05:38	Documento di testo	1 KB
DataSet10x10	23/01/2023 05:46	Documento di testo	1 KB
DataSet100x10	23/01/2023 05:46	Documento di testo	2 KB
DataSet300x10	27/01/2023 18:47	Documento di testo	6 KB
DataSet500x10	26/01/2023 04:37	Documento di testo	10 KB
DataSet600x10	27/01/2023 18:47	Documento di testo	12 KB
DataSet900x10	27/01/2023 18:47	Documento di testo	18 KB
DataSet1000x5	23/01/2023 05:38	Documento di testo	10 KB
DataSet1000x10	23/01/2023 05:46	Documento di testo	20 KB
DataSet1200x10	27/01/2023 18:47	Documento di testo	24 KB
DataSet1500x10	27/01/2023 18:47	Documento di testo	30 KB
DataSet1800x10	27/01/2023 18:47	Documento di testo	36 KB
DataSet2100x10	27/01/2023 18:47	Documento di testo	42 KB
DataSet2400x10	27/01/2023 18:47	Documento di testo	47 KB
DataSet2700x10	27/01/2023 18:47	Documento di testo	53 KB
DataSet3000x10	27/01/2023 18:47	Documento di testo	59 KB
DataSet3300x10	27/01/2023 18:47	Documento di testo	65 KB
DataSet3600x10	27/01/2023 18:47	Documento di testo	71 KB
DataSet3900x10	27/01/2023 18:47	Documento di testo	77 KB
DataSet4200x10	27/01/2023 18:47	Documento di testo	83 KB
DataSet4500x10	27/01/2023 18:47	Documento di testo	88 KB
DataSet4800x10	27/01/2023 18:47	Documento di testo	94 KB
DataSet5000x10	26/01/2023 04:38	Documento di testo	98 KB
DataSet5000x30	14/01/2023 17:50	Documento di testo	293 KB
DataSet5100x10	27/01/2023 18:47	Documento di testo	100 KB
DataSet5400x10	27/01/2023 18:47	Documento di testo	106 KB
DataSet5700x10	27/01/2023 18:47	Documento di testo	112 KB
DataSet6000x10	27/01/2023 18:47	Documento di testo	118 KB
DataSet6300x10	27/01/2023 18:47	Documento di testo	124 KB
DataSet6600x10	27/01/2023 18:47	Documento di testo	129 KB
DataSet6300x10	27/01/2023 18:47	Documento di testo	124 KB
DataSet6600x10	27/01/2023 18:47	Documento di testo	129 KB
DataSet6900x10	27/01/2023 18:47	Documento di testo	135 KB
DataSet7200x10	27/01/2023 18:47	Documento di testo	141 KB
DataSet7500x10	27/01/2023 18:47	Documento di testo	147 KB
DataSet7800x10	27/01/2023 18:47	Documento di testo	153 KB
DataSet8100x10	27/01/2023 18:47	Documento di testo	159 KB
DataSet8400x10	27/01/2023 18:47	Documento di testo	165 KB
DataSet8700x10	27/01/2023 18:47	Documento di testo	170 KB
DataSet9000x10	27/01/2023 18:47	Documento di testo	176 KB
DataSet9300x10	27/01/2023 18:47	Documento di testo	182 KB
DataSet9600x10	27/01/2023 18:47	Documento di testo	188 KB
DataSet10000x5	14/01/2023 17:42	Documento di testo	98 KB
DataSet10000x10	23/01/2023 05:45	Documento di testo	196 KB
DataSet20000x10	26/01/2023 04:41	Documento di testo	391 KB
DataSet30000x10	26/01/2023 04:38	Documento di testo	586 KB
DataSet40000x10	26/01/2023 04:38	Documento di testo	782 KB
DataSet50000x10	26/01/2023 04:38	Documento di testo	977 KB
DataSet100000x10	23/01/2023 05:45	Documento di testo	1.954 KB
DataSet100000x500	23/01/2023 05:43	Documento di testo	97.657 KB
DataSet1000000x10	23/01/2023 05:40	Documento di testo	19.532 KB
DataSet10000000x10	23/01/2023 05:45	Documento di testo	195.313 KB

Figure 16 - Dataset used in testing

For the debug of the serial code, we used Clion, however it was impossible to use it for the parallel algorithm as well. For this latter we carefully used many prints to check the proper execution of the program.

As final test we executed the program on the Google Cloud platform, using different clusters:

- Fat Cluster – Infra Regional: 2 machines, 16 cores, 64Gb RAM each one;
- Fat Cluster – Intra Regional: 3 machines, 8 cores, 32 Gb RAM each one;
- Light Cluster – Infra Regional: 16 machines, 2 cores, 4 Gb RAM each one
- Light Cluster – Intra Regional: 12 machines, 2 cores, 4 Gb RAM each one;

Experimental Results

Used formulae

$$speedup = \frac{Execution\ time\ old}{Execution\ time\ new}$$

$$Scaleup\ = \frac{Time\ for\ parallel\ execution\ with\ P = 1}{Time\ for\ parallel\ execution\ varying\ P\ and\ keeping\ costant\ work\ per\ P}$$

Premises:
All the following tests were conducted using a dataset composed by 10.000 x 10 data, where m=10.000 and n=10. Moreover, since changing the number of iterations would not have changed the ratio between the time of different processors, the number of iterations was fixed to 5 to allow us performing all the tests in a reasonable amount of time.

Fat Cluster

<u>Intra-Regional</u>

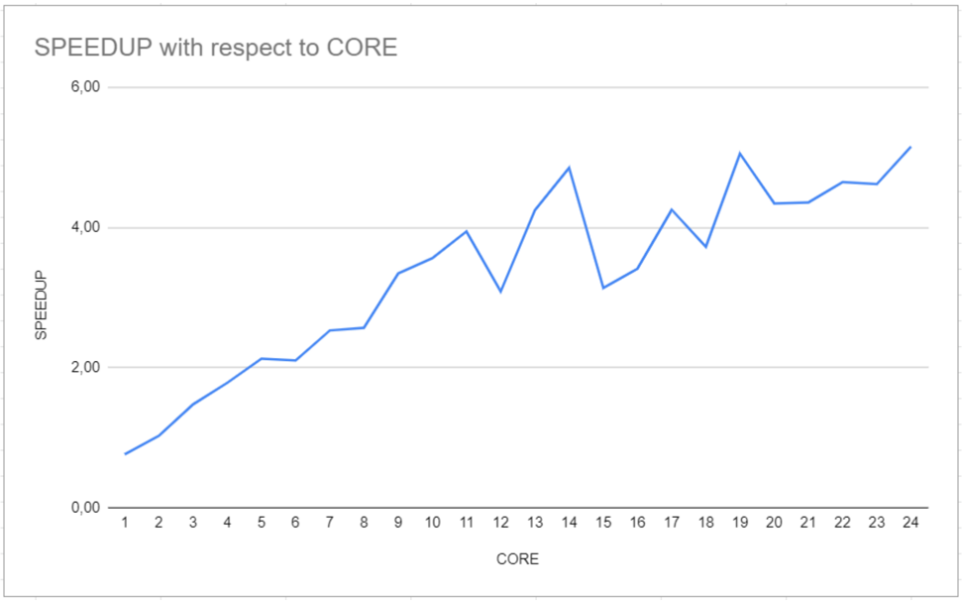


Figure 17 - Fat, Intra, Speedup

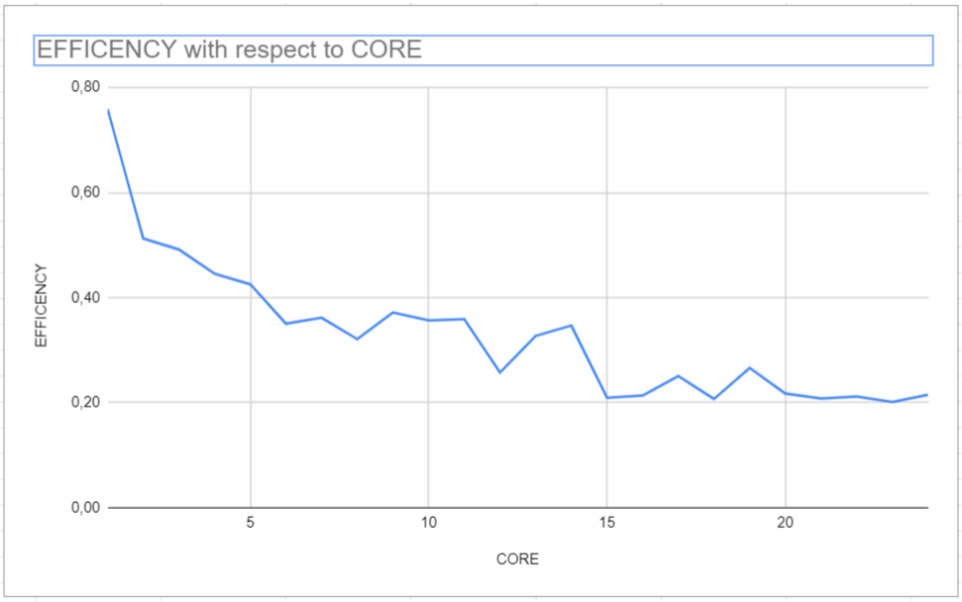


Figure 18 - Fat, Intra, Efficiency

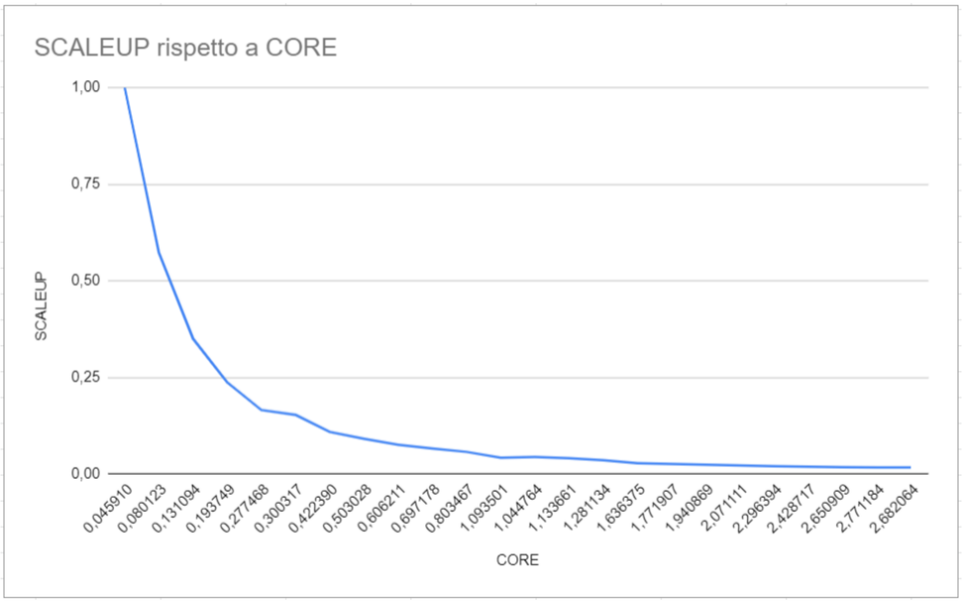


Figure 19 - Fat, Intra, Scaleup

Infra Regional

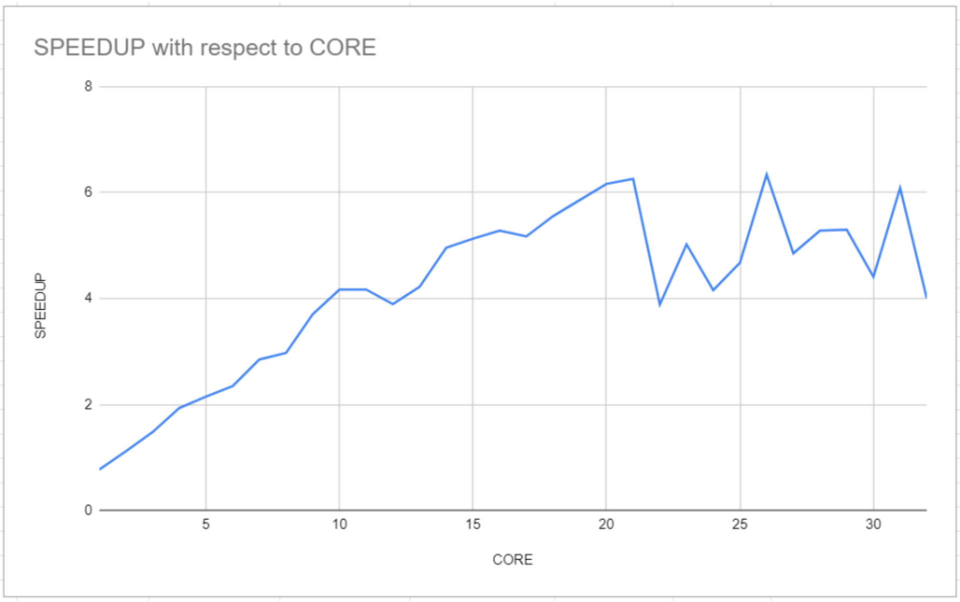


Figure 20 - Fat, Infra, Speedup

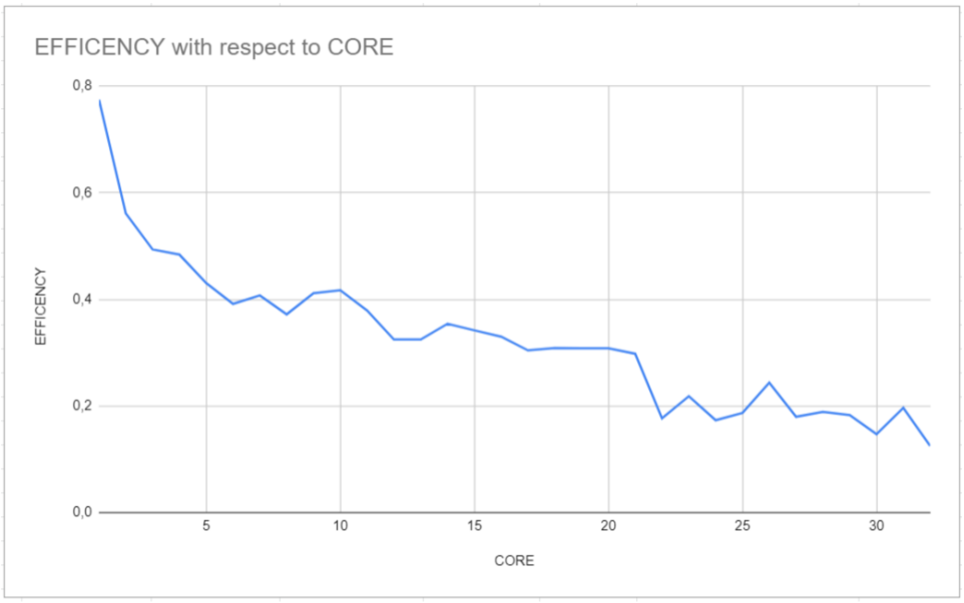


Figure 21 - Fat, Infra, Efficiency

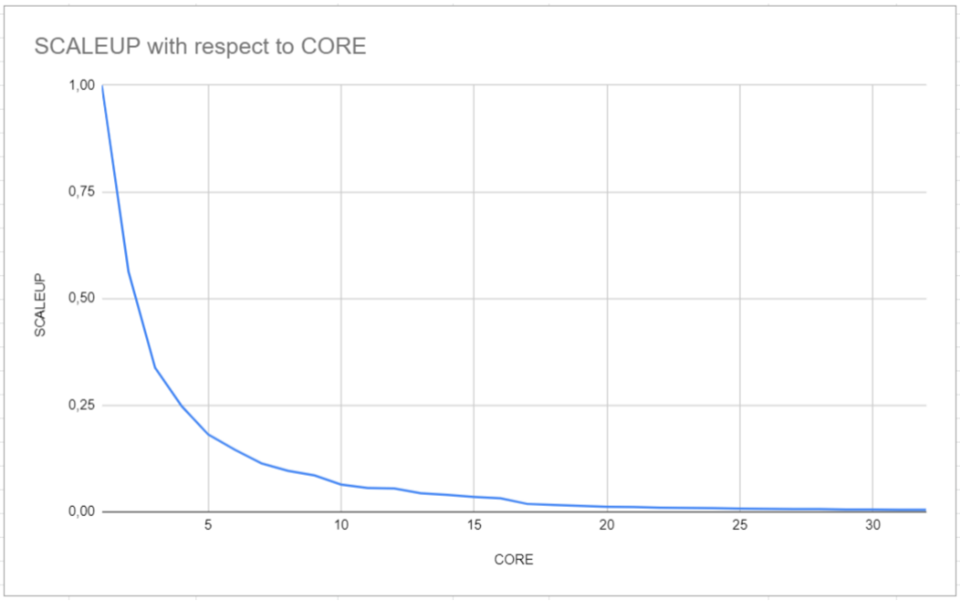


Figure 22 - Fat, Infra, Scaleup

Light Cluster

<u>Intra-Regional</u>

Since Google cloud only allowed us to keep 8 actives IP in the same region, we asked for a higher number of IP, up to 12. The goal was that of generating 12 light weighted vcores machines, to expand our testing possibility.

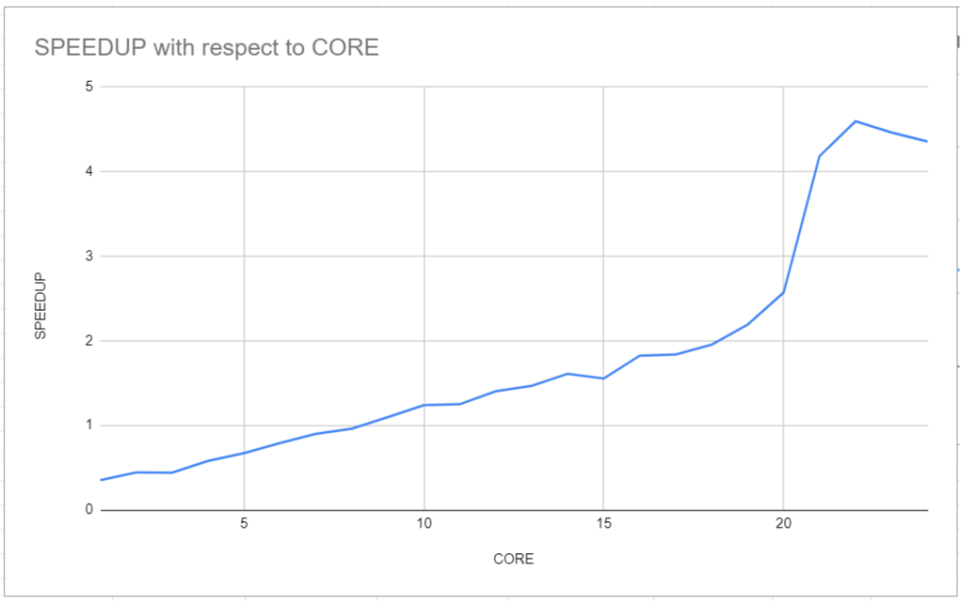


Figure 23 - Light, Intra, Scaleup

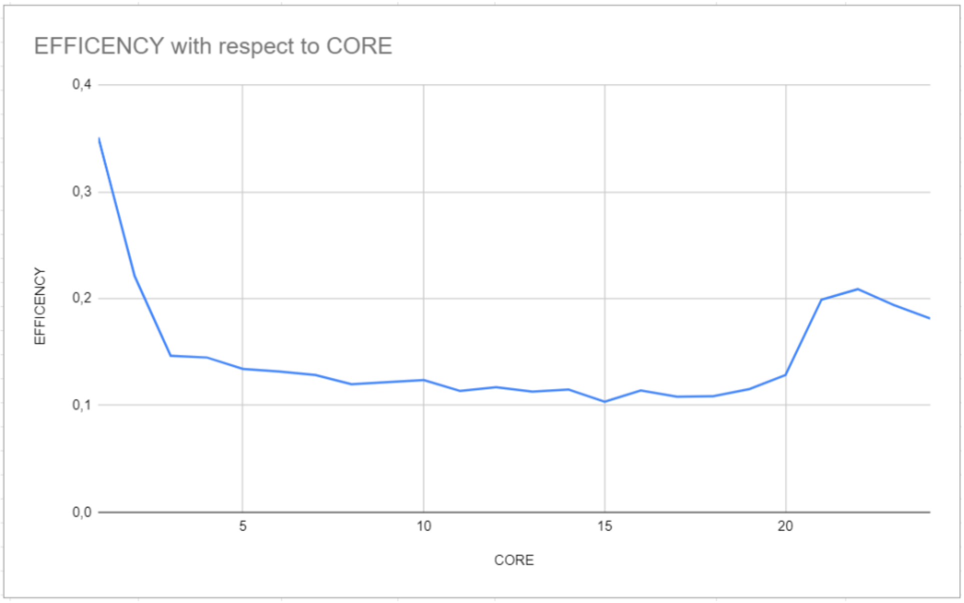


Figure 24 - Light, Intra, Efficiency

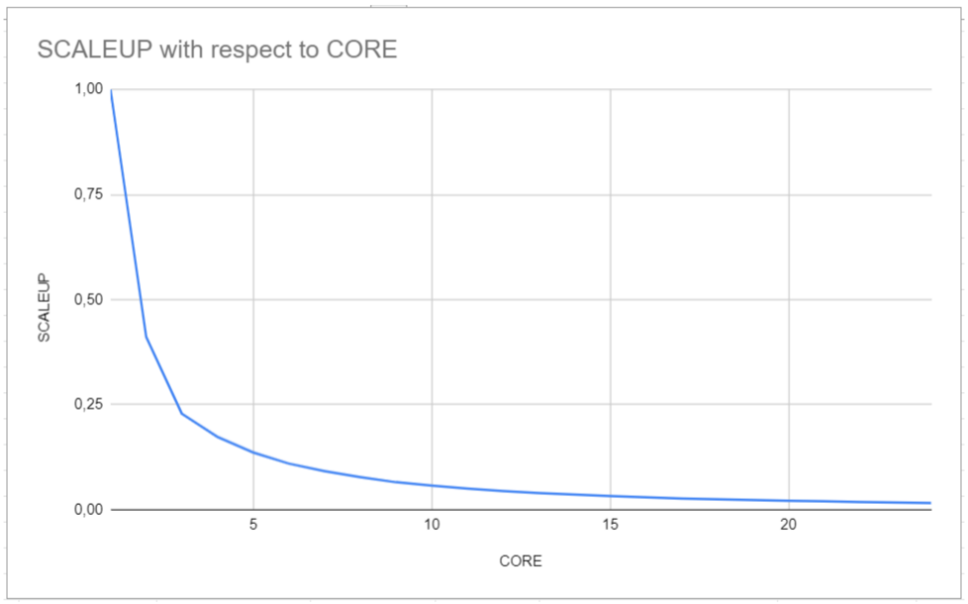


Figure 25 - Light, Intra, Scaleup

Infra Regional

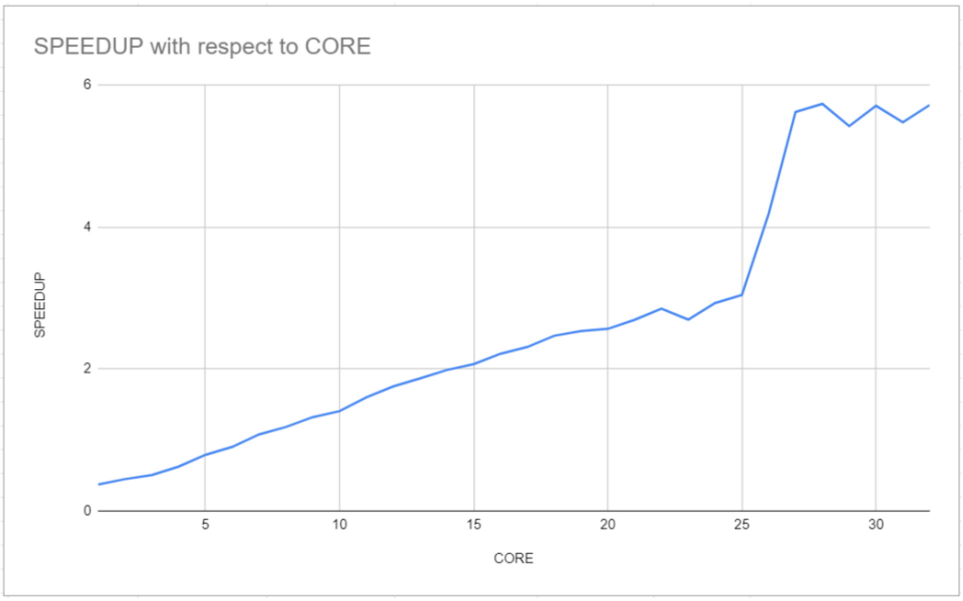


Figure 26 - Light, Infra, Speedup

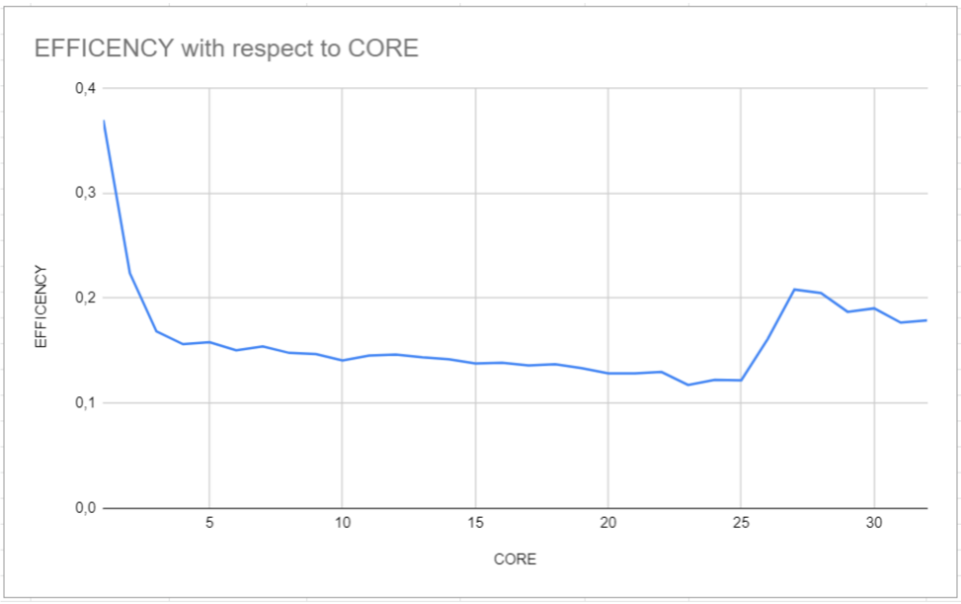


Figure 27 - Light, Infra, Efficiency

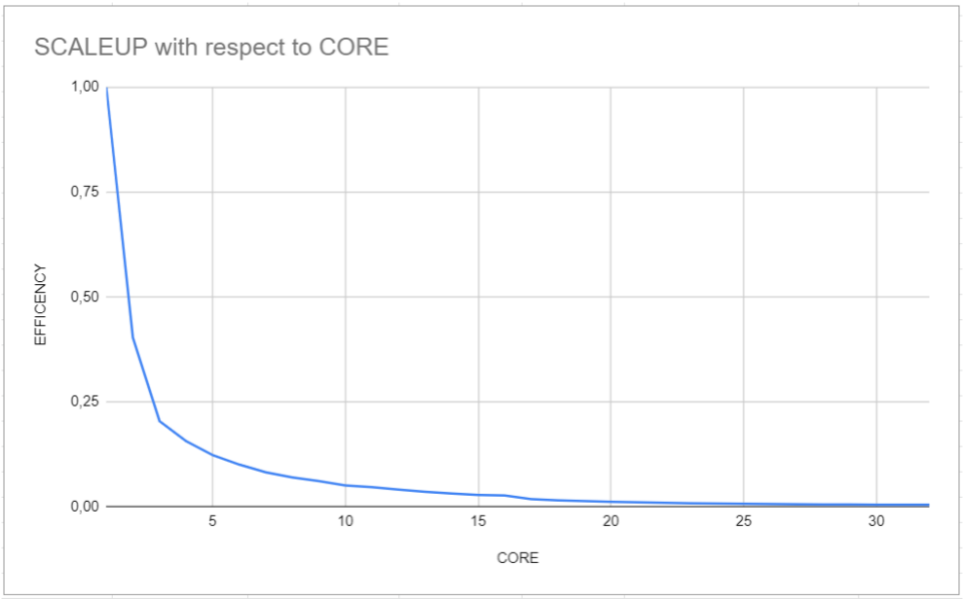


Figure 28 - Light, Infra, Scaleup

CONCLUSION and FUTURE WORKS

As we can see from the experimental results the scalability (strong and weak) is different from the theoretical values.

As regard the speedup we have a similar pattern between the four machine's configuration we used. In all cases the speedup grows almost linearly, however the growth is significantly lower than the growth in the processors' number. This could be caused by lots of factors, for example the interaction with the memory subsystem can change between the serial algorithm and the parallel.

As regard the efficiency we have a decreasing pattern, and it is obviously related to the different growth rate between speedup and the number of processors. This means that each new processor is exploited at its maximum capacity.

The scaleup decreases in a higher than linear manner. This means that when we increase the size of the problem, our algorithm struggles to well distribute the workload on other processors. Future works can concentrate more on this point and increasing the scaleup performances.

Moreover, we identified three possible upgrades to the algorithm:

- To avoid running out of RAM, using very large datasets, only a portion of the data could be loaded at a time:

    1. Read and load a portion of the data into RAM;
    2. Perform calculations on it;
    3. Unload the data from RAM;
    4. Repeat from 1 until we processed all data.

- We also have the tail problem: in some cases, the master may have to perform almost twice the load of the slaves. To avoid this problem, the tail load could be divided among all the processors.

- Reduce the number of times the buffersize is sent, since its size doesn't change.

INDIVIDUAL CONTRIBUTION

To decide about the division of work, we firstly listed all the main points the project should have touched and then we have identified those sections that could have been distributed between us in the best possible way. The two main parts we have identified are the preliminary study of the algorithm with its possibility of parallelization and the algorithm implementation. The first part was mainly cured by Andrea while the second one was mainly cured by Cristian. The entire work has been conducted in a cooperative way, with the help of videocalls to discuss and possibly improve the work done by the other. As regard the last section of the report (experimental results) it was conducted together to evaluate the entire project.