Advanced Computer Architecture 2023-2024

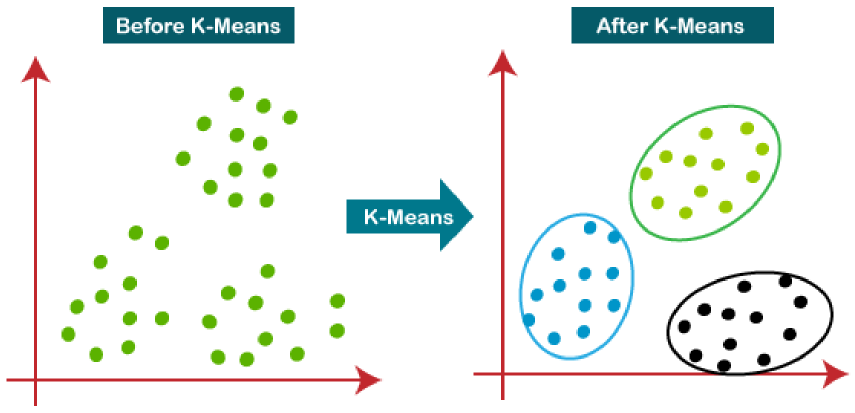
K-Means Clustering Parallelization

Introduction

K-means clustering is a popular unsupervised machine learning algorithm used to partition a dataset into distinct groups, or clusters. The algorithm aims to divide *n* data points into *k* clusters, where each data point belongs to a cluster according to a comparison made using a specific similarity function. This process minimizes the variance within each cluster, creating cohesive and well-separated groups.

The algorithm operates through a simple and iterative procedure:

1. Initialization: Select *k* initial cluster centroids randomly from a given dataset.
2. Assignment: Assign each data point to the nearest cluster centroid based on the similarity function output.
3. Update: Calculate the new centroids as the mean of all data points assigned to each cluster.
4. Convergence: Repeat the assignment and update steps until the cluster centroids stabilize and do not change significantly, or a predefined number of iterations is reached.



K-means clustering is widely used in various fields such as market segmentation, image compression, and pattern recognition due to its simplicity and efficiency. However, it requires to specify in advance the number of clusters *k* and may converge to a local optimum, which is sensitive to the initial placement of the centroids. Despite these limitations, K-means remains a fundamental and practical tool for exploratory data analysis and clustering tasks.

Analysis of the serial algorithm

The algorithm used for this analysis is a customized version of the K-Means Algorithm. In this case the algorithm is structured to accept in input:

- a dataset characterized by:
    - a first line, which can be considered a header line, providing information about the number of observations and the number of features per observation
    - several lines, each one defining a different observation with its related features
- the number of clusters desired in which observations will be divided
- the number of iterations the algorithm will iterate

All these 3 elements must be provided by command line to have the algorithm work properly.

For what it concerns the internal characteristics of the algorithm itself, the critical functions are:

- *read_data()*: the function processes the entire dataset file allocating each observation in a structure object *Point*. Such a structure provides information about the cluster the observation was assigned and a vector containing the feature values
- *km()*: this function represents the heart of the algorithm. At each iteration it computes the cluster centroid position, obtained as the average of the point positions belonging to a cluster, and then it assigns each point to the closer cluster
- *euclidean_distance()*: it represents the similarity function used in the algorithm to define the point closeness to each cluster

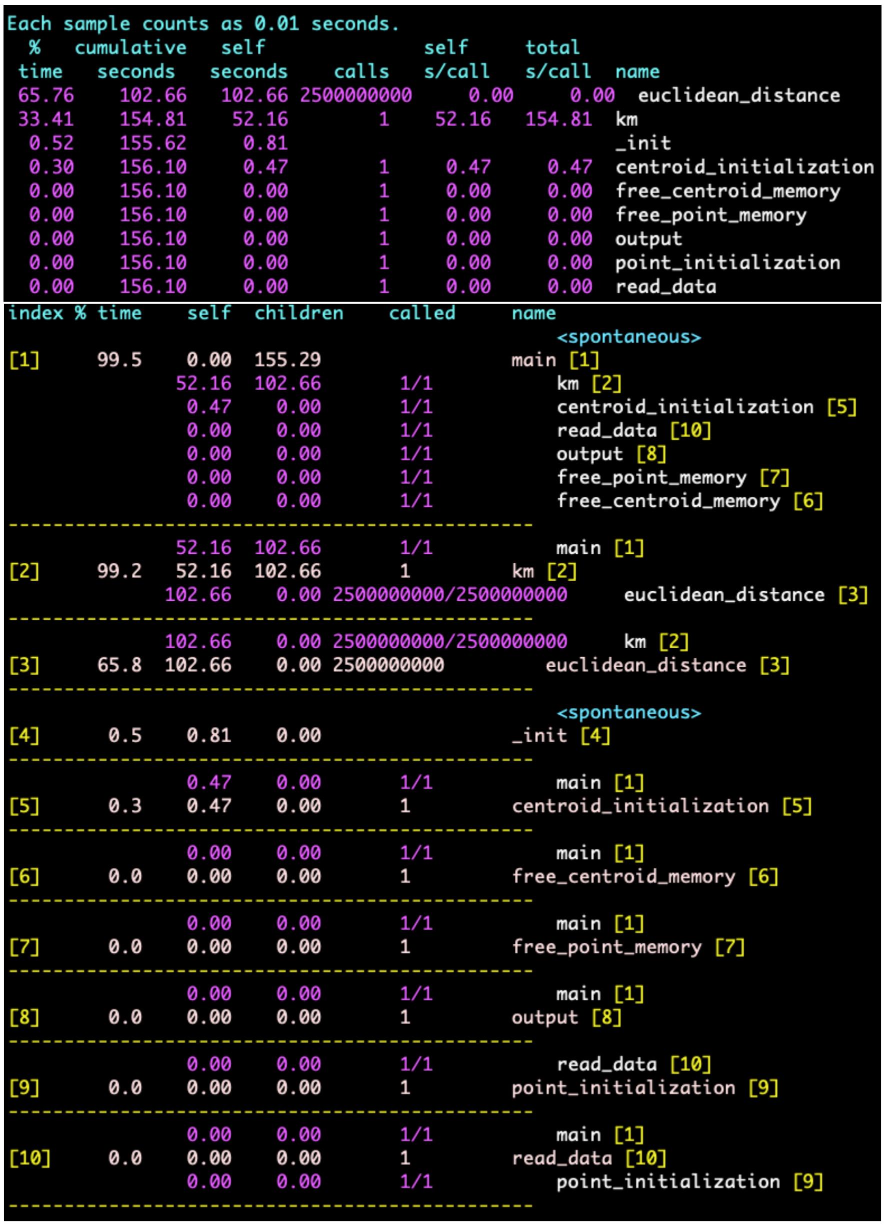
A-priori study of available parallelism

Looking at the code we can observe that the only interesting parts, which could be the target of a parallelization effort, concern:

- The data extrapolation of a dataset file in the *read_data()* function. Indeed, in this case it could be possible perform a parallel reading of different lines exploiting different processes
- The observation assignment to a specific cluster in *km()* function. In this case, instead, it could be possible to find a to perform the assignment for different group of points handled by the different processes

The remaining part of the code is essentially characterized by initialization functions and by the function providing the results after the entire execution of the algorithm, which implies the write operation and which must be performed only by a single process.

To better detect what the crucial part are for the parallelization process, the performances of the different functions of the serial algorithm were assessed using the profiling tool *GPROF*.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 65.76    102.66   102.66 2500000000     0.00     0.00  euclidean_distance
 33.41    154.81    52.16         1    52.16   154.81  km
  0.52    155.62     0.81                             _init
  0.30    156.10     0.47         1     0.47     0.47  centroid_initialization
  0.00    156.10     0.00         1     0.00     0.00  free_centroid_memory
  0.00    156.10     0.00         1     0.00     0.00  free_point_memory
  0.00    156.10     0.00         1     0.00     0.00  output
  0.00    156.10     0.00         1     0.00     0.00  point_initialization
  0.00    156.10     0.00         1     0.00     0.00  read_data
index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.5    0.00  155.29                 main [1]
               52.16  102.66       1/1           km [2]
                0.47    0.00       1/1           centroid_initialization [5]
                0.00    0.00       1/1           read_data [10]
                0.00    0.00       1/1           output [8]
                0.00    0.00       1/1           free_point_memory [7]
                0.00    0.00       1/1           free_centroid_memory [6]
-----------------------------------------------
               52.16  102.66       1/1           main [1]
[2]     99.2   52.16  102.66       1         km [2]
              102.66    0.00 2500000000/2500000000     euclidean_distance [3]
-----------------------------------------------
              102.66    0.00 2500000000/2500000000     km [2]
[3]     65.8  102.66    0.00 2500000000         euclidean_distance [3]
-----------------------------------------------
                                                 <spontaneous>
[4]      0.5    0.81    0.00                 _init [4]
-----------------------------------------------
                0.47    0.00       1/1           main [1]
[5]      0.3    0.47    0.00       1         centroid_initialization [5]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[6]      0.0    0.00    0.00       1         free_centroid_memory [6]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[7]      0.0    0.00    0.00       1         free_point_memory [7]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[8]      0.0    0.00    0.00       1         output [8]
-----------------------------------------------
                0.00    0.00       1/1           read_data [10]
[9]      0.0    0.00    0.00       1         point_initialization [9]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[10]     0.0    0.00    0.00       1         read_data [10]
                0.00    0.00       1/1           point_initialization [9]
-----------------------------------------------
```

The test was done providing in input a dataset of 50.000 observations and requiring a number of iterations equal to 5000. The output, represented by the two pictures above, shows the behavior of the algorithm:

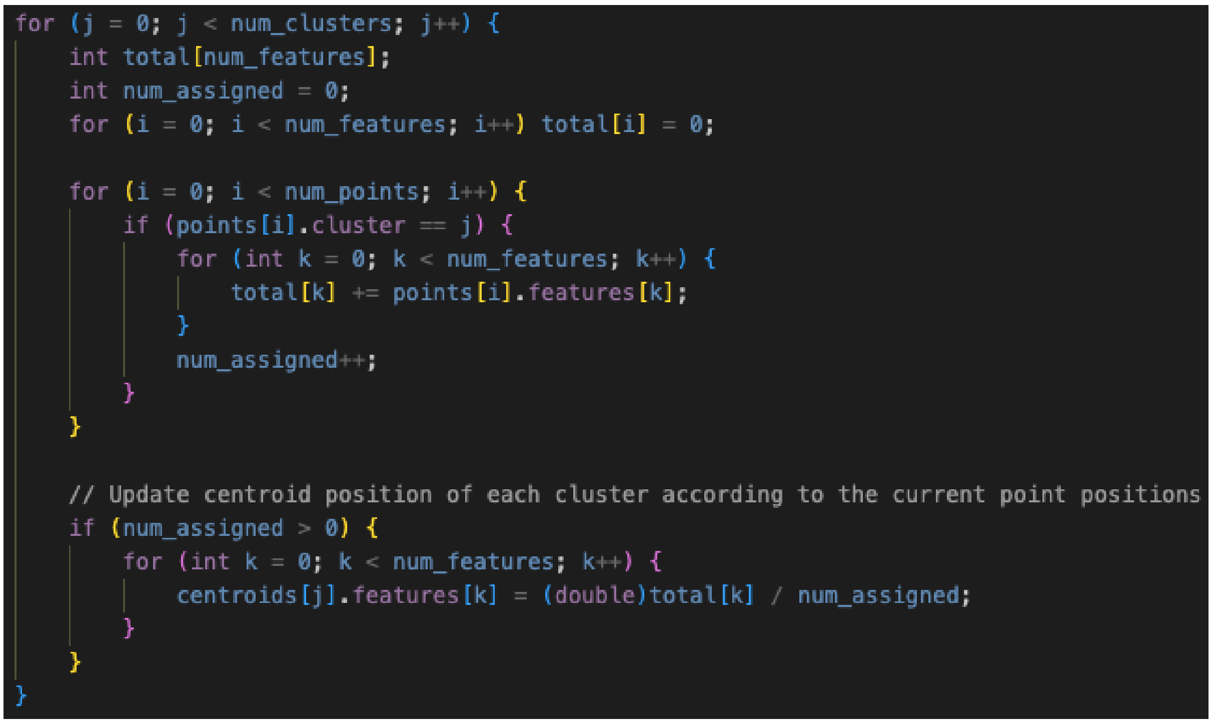
- In the first one we have a recap of the amount time required by each function according to different aspects.
- In the second one it is represented a table describing the call tree of the program sorted by the total amount of time spent in each function and its children.

It immediately appears obvious from the analysis the function taking most of the percentage of the total running time of the program is the *euclidean_distance()* function, however with a closer look to the results we can notice such a function is called in each single iteration per each single observation. Indeed, the call tree of the program reveals how the *euclidean_distance()* function is a child of the *km()* function and this explains also how *km()* is the function having the largest average number of milliseconds spent both in itself (*%self s/call*) and its descendents (*%total s/call*) per call. So, given this further assessment most of the parallelization effort was focused on the *km()* function.

‘km’ function

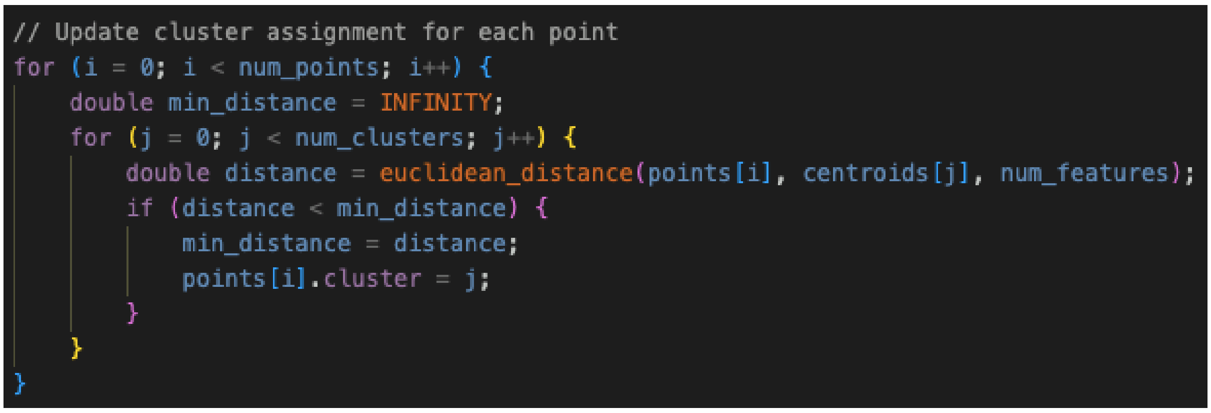
The function is structured in 2 main blocks:

- The first one focused on the computation of centroid positions according to the observation assignments done up to the current iteration



```c
for (j = 0; j < num_clusters; j++) {
    int total[num_features];
    int num_assigned = 0;
    for (i = 0; i < num_features; i++) total[i] = 0;

    for (i = 0; i < num_points; i++) {
        if (points[i].cluster == j) {
            for (int k = 0; k < num_features; k++) {
                total[k] += points[i].features[k];
            }
            num_assigned++;
        }
    }

    // Update centroid position of each cluster according to the current point positions
    if (num_assigned > 0) {
        for (int k = 0; k < num_features; k++) {
            centroids[j].features[k] = (double)total[k] / num_assigned;
        }
    }
}
```

- The second one focused on computing the distance between each point and each cluster, with the aim of updating the cluster assignment made for each point



```
// Update cluster assignment for each point
for (i = 0; i < num_points; i++) {
    double min_distance = INFINITY;
    for (j = 0; j < num_clusters; j++) {
        double distance = euclidean_distance(points[i], centroids[j], num_features);
        if (distance < min_distance) {
            min_distance = distance;
            points[i].cluster = j;
        }
    }
}
```

Both blocks of code are repeated iteratively for a several amount of times and even if it could seem easy to parallelize them just splitting the iteration executions on different processes, such approach is not possible since there is a direct dependence between the results of an iteration and the previous one. Another constraint, which cannot be violated, concerns the sequence in which these two blocks are executed. Indeed, it is mandatory to update the position of the cluster centroids, before being able to assign an observation in the dataset to a specific cluster.

A valid parallelization approach, instead, can be faced considering the individual parallelization of the two blocks.

First block - Computing centroid positions

The first block of code of the *km()* function aggregates all the features of the points assigned to a particular cluster and finally computes the cluster centroid as the average of the aggregated features. In this situation there can be 2 possible strategies for parallelization:

1. Dividing the workload per cluster
2. Dividing the workload per observation

Exploiting the first approach, the approximate number of instructions which can be parallelized is:

$$N\_Instr_{cluster} \approx \frac{NC\left(2 + 2NF + ND\left(1 + \frac{NF+1}{NC}\right)\right)}{P} \approx \frac{NC(2+2NF)}{P} + \frac{ND(NC+NF+1)}{P}$$

Instead, exploiting the second approach:

$$N\_Instr_{obs} \approx \frac{ND(NC + NF + 1)}{P} + NC(2 + 2NF)$$

$$P = number\ of\ processes, \quad ND = number\ of\ data, \quad NC = number\ of\ clusters$$

From this approximate calculus, it looks clear how following the strategy of parallelizing the block, subdividing the workload among the processes according to the number of clusters, results in a stronger reduction of the amount of instructions handled by each single process. However, this strategy has an intrinsic scalability problem strictly correlated to the number of clusters. In fact, it occurs very often that the number of clusters required is smaller than the number of available processes, so increasing the number of processes does not necessarily imply a performance improvement. On the contrary it will reach a number after which the speedup obtained will stop increasing because processes won't help anymore in the parallel execution.

Following these observations and considering that a possible parallelization strategy considering the process workload division according to the number of features would result in a situation similar to the one just seen, the last plausible option remains to apply a workload division distributing the observations among the processes. Moreover, if we consider that in most cases the number of observations in a dataset is much larger than the number of features or clusters, we would observe that:

$$N\_Instr_{cluster} \approx N\_Instr_{obs}$$

Second block – Cluster assignments

For what it concerns the second block, a similar analysis can be performed. However, if we compute the approximate number of instructions following the two main strategies presented in the previous paragraph, we would obtain:

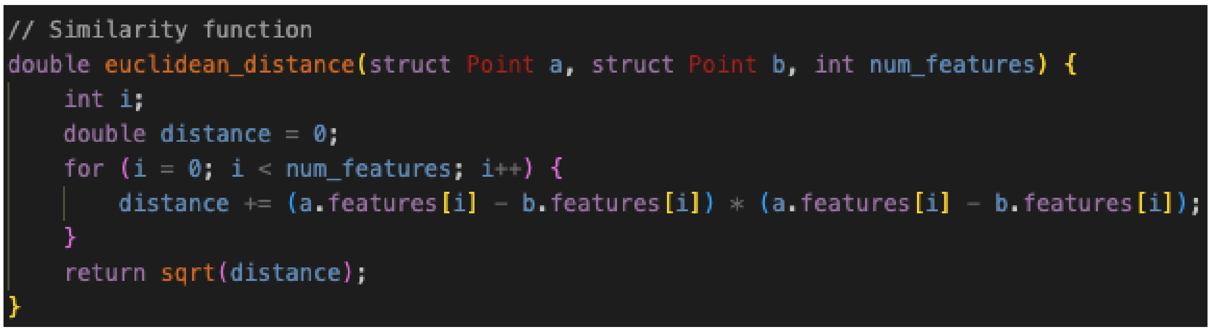
$$N\_Instr_{obs} \approx \frac{ND\left(1 + NC\left(N\_Instr_{eucl} + \frac{1}{2}(2)\right)\right)}{P} = \frac{ND(1 + NC(N\_Instr_{eucl} + 1))}{P}$$

$$N\_Instr_{cluster} \approx ND\left(1 + \frac{NC(N\_Instr_{eucl} + 1)}{P}\right)$$

Considering the *if* condition is true half of the times and that:

$$N\_Instr_{eucl} \approx 3 + 4NF$$

$$N\_Instr_{eucl} = number\ of\ instructions\ in\ the\ euclidean\ distance\ function$$



```
// Similarity function
double euclidean_distance(struct Point a, struct Point b, int num_features) {
    int i;
    double distance = 0;
    for (i = 0; i < num_features; i++) {
        distance += (a.features[i] - b.features[i]) * (a.features[i] - b.features[i]);
    }
    return sqrt(distance);
}
```

Resulting even better a parallelization strategy based on the distribution of observations across processes than a parallelization based on the subdivision of the workload of clusters.

Conclusion

The better parallelization approach from a-priori study results to be the distribution of the dataset observations across several processes. A possible drawback it needs to keep in consideration concerns the communication overhead caused by the necessity for processes to communicate intermediate results.

A-priori theoretical assessment

The conclusion we arrived can be supported also performing a-priori theoretical assessment of gainable speedups. Considering the total number of executed instructions in *km()* function results to be:

$$\begin{aligned} Tot_{N\_Instr} &\approx ND(NC + NF + 1) + NC(2 + 2NF) + ND\left(1 + NC\left(N_{Instr_{eucl}} + 1\right)\right) \\ &= ND(NC(4NF + 5) + NF + 2) + NC(2 + 2NF) \end{aligned}$$

- *Observation distribution strategy*

Accepting the distribution of observations across the set of available processes as the main parallelization strategy, the number of instructions which will be carried out in parallel and sequentially are:

$$N\_Instr_{parallel} \approx ND(NC(4NF + 5) + NF + 2)$$

$$N\_Instr_{serial} \approx NC(2 + 2NF)$$

If we consider some realistic values for ND, NC and NF like:

$$ND = 10000 \quad NC = 5 \quad NF = 2$$

The resulting a-priori theorical assessment of the speed-up obtained using such parallelization strategy with *P* processors would be:

$$\%Code_{parallel} = \frac{690000}{690030} \approx 0{,}9995$$

$$\%Code_{serial} = \frac{30}{690030} \approx 0{,}0004$$

This concludes that:

$$Speedup(P) = \frac{1}{\%Code_{serial} + \frac{\%Code_{parallel}}{P}} \approx P$$

- *Cluster distribution strategy*

Accepting the distribution of cluster workloads across the set of available processes as the main parallelization strategy, the number of instructions which will be carried out in parallel and sequentially are:

$$N\_Instr_{parallel} \approx NC(ND((4NF + 5)) + 2 + 2NF)$$

$$N\_Instr_{serial} \approx ND(2 + NF)$$

The resulting a-priori theorical assessment of the speed-up obtained using such parallelization strategy with *P* processors would be:

$$\%Code_{parallel} = \frac{650030}{690030} \approx 0{,}942$$

$$\%Code_{serial} = \frac{40000}{690030} \approx 0{,}06$$

This concludes that also in this case the speedup would be good enough, however as the number of processes involved increases, such strategy would result to be less performing.

So, also with a-priori theoretical assessment of the speedup, we can conclude that the observation distribution strategy looks being more solid. The gap of the speedup obtained using the two strategies becomes more and more evident as the number of processors applied in the execution is increased.

MPI Parallel Implementation

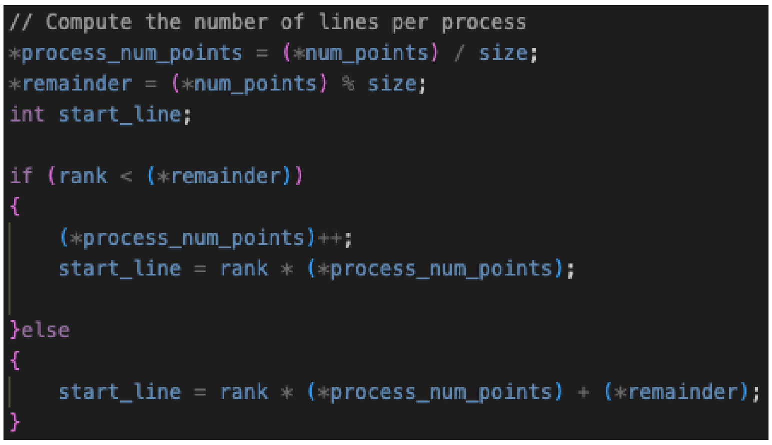
A possible parallel solution of the previous presented serial algorithm, based on the observation distribution strategy, was developed through 3 main steps:

1- An initial distribution of the observation data in a dataset to the different processors during the call of the *read_data()* function

2- An MPI parallel approach to update centroid positions starting from the partial contributions of each point handled by the different processors and, afterwards, an update of cluster assignment for each point in *km()* function

3- An MPI parallel approach to gather the results computed by each processor

1. Since all the parallelization effort is based on the distribution of observations in a dataset across processors, the *read_data()* function was modified allowing each processor to read only a partial part of the dataset file, i.e. the lines containing the observations the processor must be interested in.

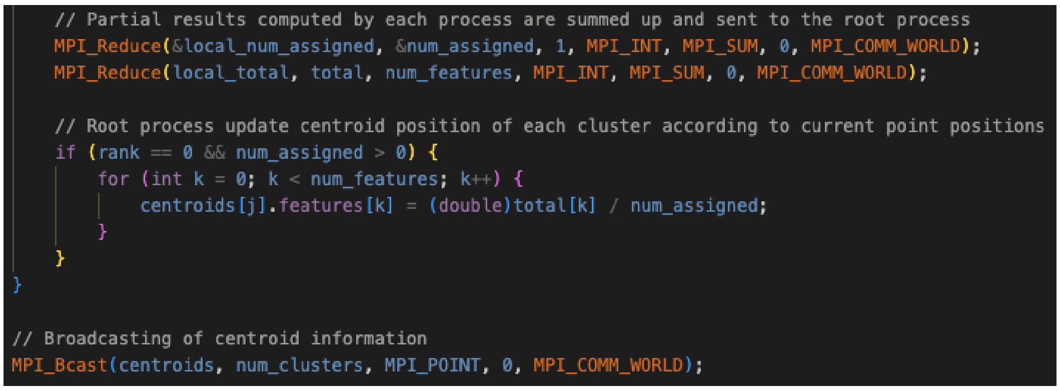
   ```
   // Compute the number of lines per process
   *process_num_points = (*num_points) / size;
   *remainder = (*num_points) % size;
   int start_line;

   if (rank < (*remainder))
   {
       (*process_num_points)++;
       start_line = rank * (*process_num_points);

   }else
   {
       start_line = rank * (*process_num_points) + (*remainder);
   }
   ```

   The calculus concerning the number of observations assigned to each process is simply obtained considering the number of processors used and the total number of observations in the dataset (the number of lines). Moreover, some few additional lines of code were added to cope the case the dataset size was not equally divisible to each processor. In this case, not to overcharge a single processor, an additional line was assigned to all the processes whose rank was smaller than the remainder dataset number of lines.
   To make each processor start reading from the correct line of the dataset file, it was also computed the start line for each one of them. Thereby, all processors, except the root one, can jump to target line and start reading the amount of lines previously established.

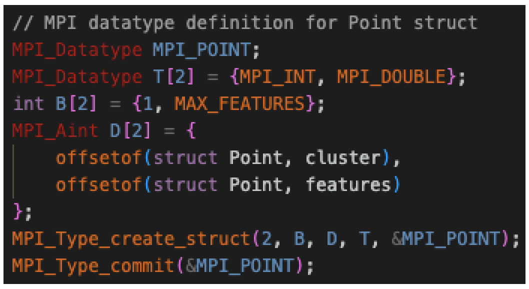
   Another possible approach to menage the observation distribution could be the one of performing the scanning of the dataset file using only one process and then scattering the data to all other processes using the *Scatterv()* MPI function.

2. Accomplished the call to the *read_data()* function, each process proceeds entering in the *km()* function. The execution of such function, as seen in previous paragraphs, can be split in two parts:

a. The update of cluster centroid positions. Here each process computes the partial contribution, given by the observation it handles, in determining each centroid position. Afterwards, such partial contributions are summed together and sent to the root process, which uses the current results to update the centroid position and communicate the new centroid positions to all other processes.



```
    // Partial results computed by each process are summed up and sent to the root process
    MPI_Reduce(&local_num_assigned, &num_assigned, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(local_total, total, num_features, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Root process update centroid position of each cluster according to current point positions
    if (rank == 0 && num_assigned > 0) {
        for (int k = 0; k < num_features; k++) {
            centroids[j].features[k] = (double)total[k] / num_assigned;
        }
    }
}

// Broadcasting of centroid information
MPI_Bcast(centroids, num_clusters, MPI_POINT, 0, MPI_COMM_WORLD);
```

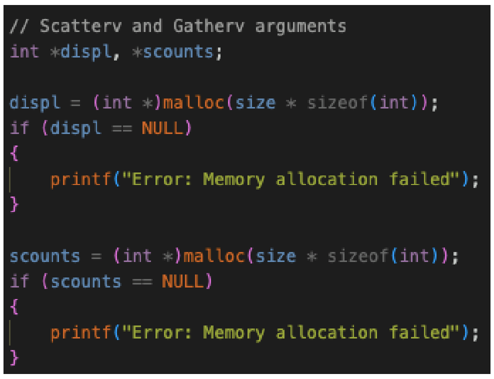


```
// MPI datatype definition for Point struct
MPI_Datatype MPI_POINT;
MPI_Datatype T[2] = {MPI_INT, MPI_DOUBLE};
int B[2] = {1, MAX_FEATURES};
MPI_Aint D[2] = {
    offsetof(struct Point, cluster),
    offsetof(struct Point, features)
};
MPI_Type_create_struct(2, B, D, T, &MPI_POINT);
MPI_Type_commit(&MPI_POINT);
```

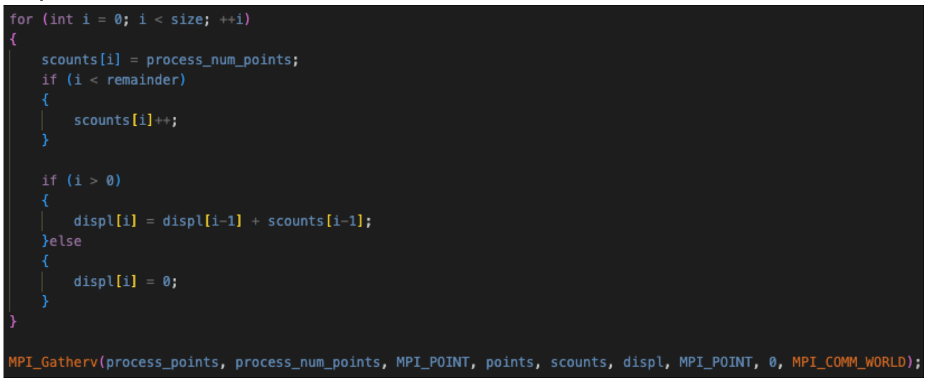
The completion of the centroid positions broadcasting needed to use a specific MPI datatype created ad hoc to allow such operation. The new MPI datatype (*MPI_POINT*) was created with the aim of representing the C data structure used to encapsulate observation information.

b. The update of cluster assignments. This part of the function didn't require any change from the original algorithm, indeed, once each process has knowledge of the new centroid positions, it can perform the update of the cluster assignments separately on the set of observations it handles.

3. Finally, once the algorithm worked for the number of iterations specified, all the processes computed cluster assignments for each of the observations they handled. To gather such results and write them in a single file, processes need to send the information of each observation handled to the root process and then the root process provides an output file.



```
// Scatterv and Gatherv arguments
int *displ, *scounts;

displ = (int *)malloc(size * sizeof(int));
if (displ == NULL)
{
    printf("Error: Memory allocation failed");
}

scounts = (int *)malloc(size * sizeof(int));
if (scounts == NULL)
{
    printf("Error: Memory allocation failed");
}
```

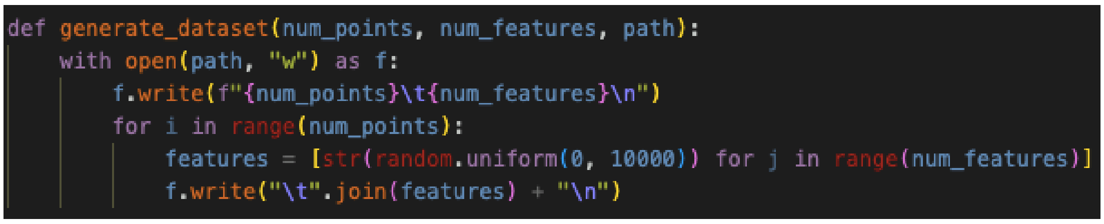


```
for (int i = 0; i < size; ++i)
{
    scounts[i] = process_num_points;
    if (i < remainder)
    {
        scounts[i]++;
    }

    if (i > 0)
    {
        displ[i] = displ[i-1] + scounts[i-1];
    }else
    {
        displ[i] = 0;
    }
}

MPI_Gatherv(process_points, process_num_points, MPI_POINT, points, scounts, displ, MPI_POINT, 0, MPI_COMM_WORLD);
```

Testing and debugging

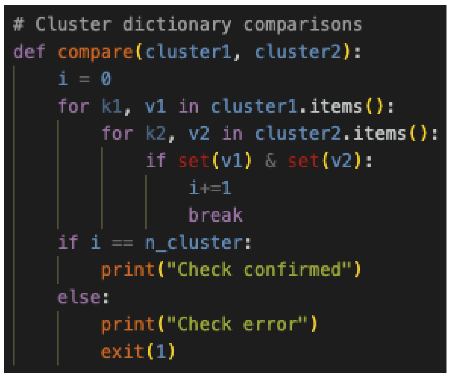
The test and debugging of the parallelized version of the K-Means algorithm was achieved with the support of two additional Python scripts:

- *dataset_generator.py* : such script was used to generate different possible sets of observations, specifying the desired number of data and feature per data, and where to locate the set. The heart of observation generation is based on the picking random values from a uniform distribution, used as feature values for data.



```python
def generate_dataset(num_points, num_features, path):
    with open(path, "w") as f:
        f.write(f"{num_points}\t{num_features}\n")
        for i in range(num_points):
            features = [str(random.uniform(0, 10000)) for j in range(num_features)]
            f.write("\t".join(features) + "\n")
```

- *check.py* : such script was mainly used to speed up the test phase and to provide also a way to check that the results of the serial and parallel algorithm were compatible. The script main functions comprise: the *generate_dataset()* function and the *compare()* function, a function used to check results.



```python
# Cluster dictionary comparisons
def compare(cluster1, cluster2):
    i = 0
    for k1, v1 in cluster1.items():
        for k2, v2 in cluster2.items():
            if set(v1) & set(v2):
                i+=1
                break
    if i == n_cluster:
        print("Check confirmed")
    else:
        print("Check error")
        exit(1)
```

Performance and scalability analysis

To perform some analysis about the performance of the parallel algorithm it was exploited the cloud infrastructure of Google Cloud Platform. The goal of the analysis was that to assess performances in terms of execution time and speedup, weak and strong scalability capabilities, in clusters of machines located in the same region and located in different regions.

The cluster configurations of choice, for both intra- and infra-regional clusters, are:

- light cluster: 8 2-vcores machines, each equipped with 8GB of RAM
- fat cluster: 2 8-vcores machines, each equipped with 32GB of RAM

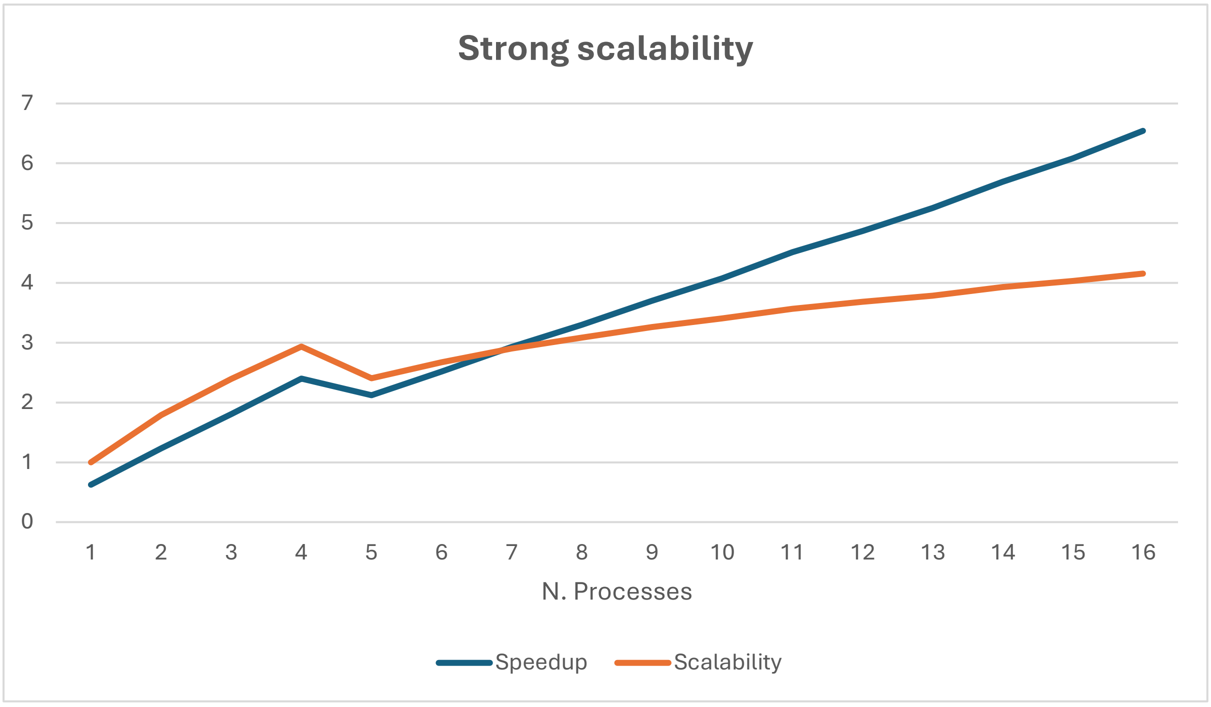
To study the strong scaling performances of the algorithm it was used a dataset with size 500.000 of observations, a number of clusters equal to 10 and a number of iterations equal to 100. Whereas, to study the weak scaling performances of the algorithm it was used a dataset with size varying from 50.000 to 800.000 of observations as more cores where added, a number of clusters equal to 10 and a number of iterations equal to 100.

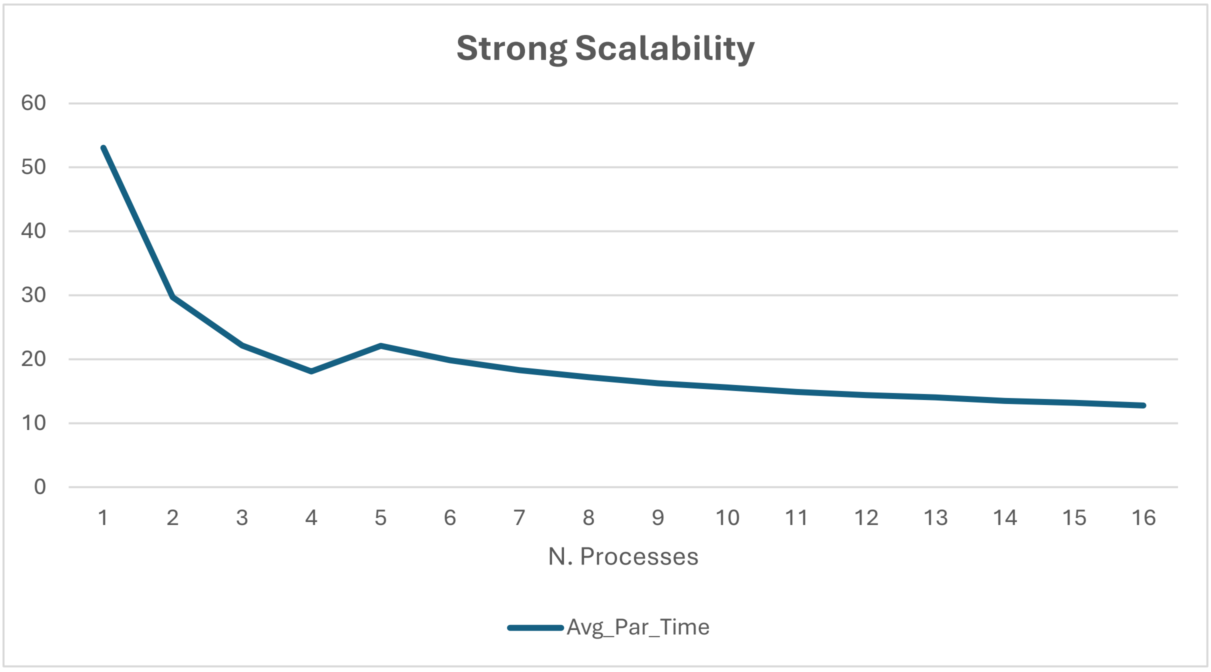
In all the test cases, the execution time was computed exploiting the *sys/time.h* C library.

Fat cluster

Intra-regional

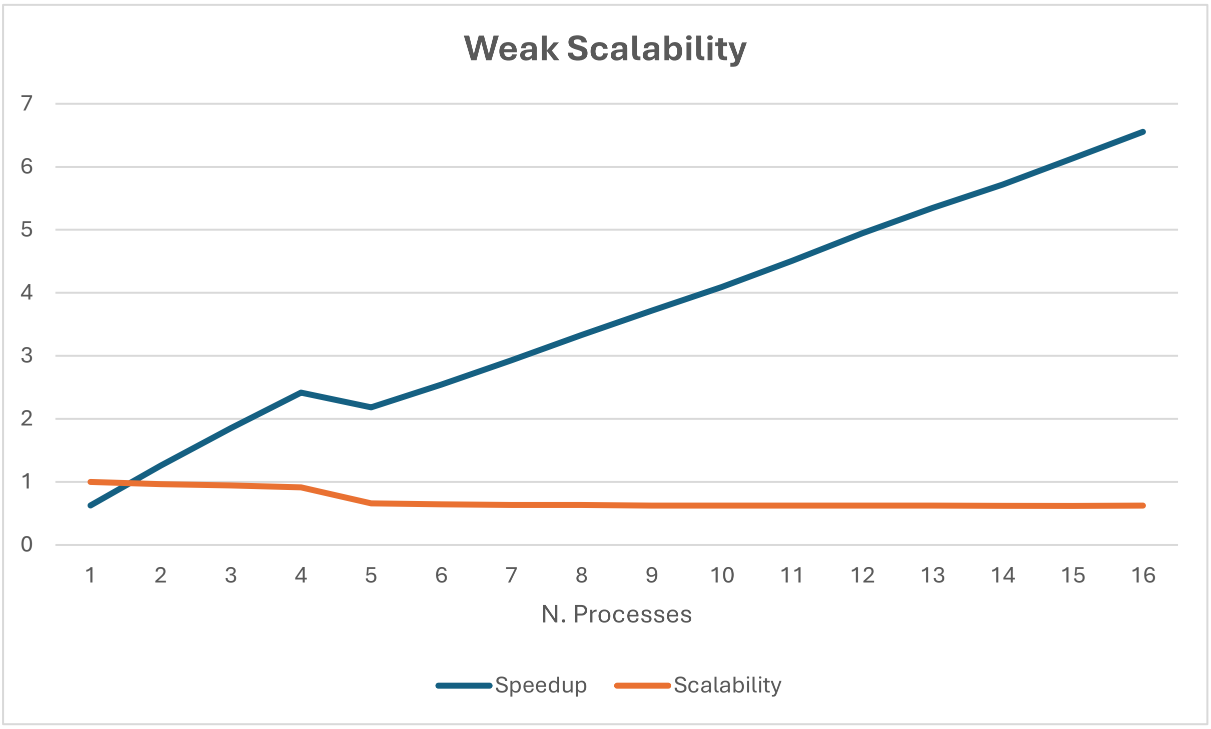
Strong scaling test

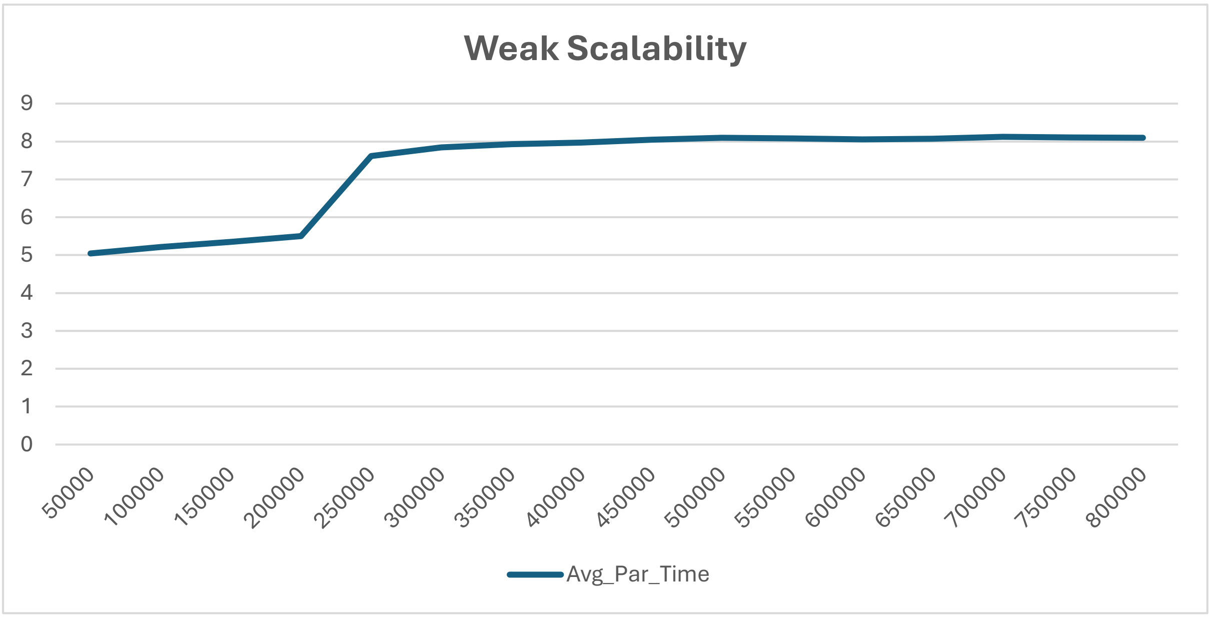




As we can see from the first graph above, the configuration of the fat cluster, located in the same region, performs particularly well both in terms of speedup and scalability. As expected, this is also confirmed by the time graph, where the timeline decreases as more processes are involved in the computation. The only critical point looks to be after 6 processes are involved, indeed the average parallel time starts decreasing in a smoother way and also the level of scalability of the problem doesn't follow very well the speedup behavior.

Weak scaling test

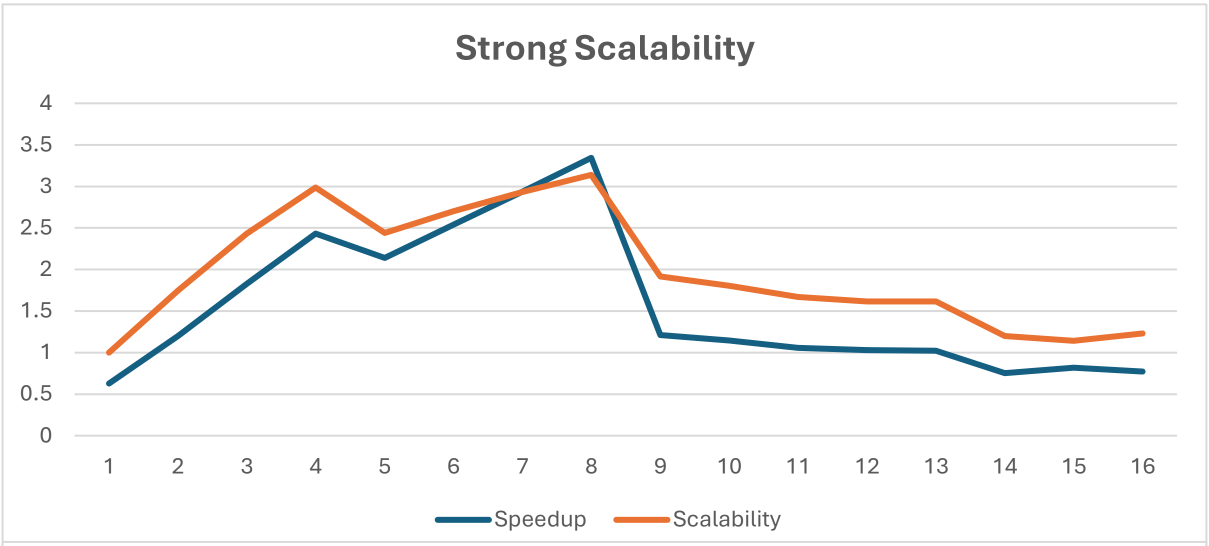


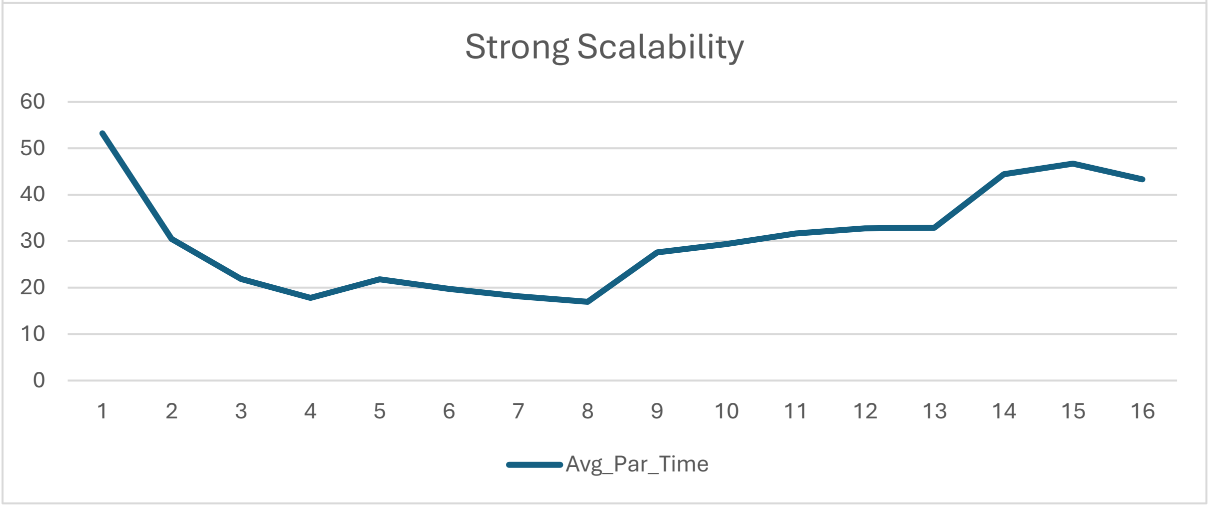


The same cluster configuration acts quite well also in the case the amount of workload increases as more processes are involved in the computation. The speedup slightly increases as the number of processes increases and reaches almost the same value as in a strong scale test, while scalability and average parallel time required stay almost constant, similarly to an ideal case after the 5th process enters in execution.

Infra-regional

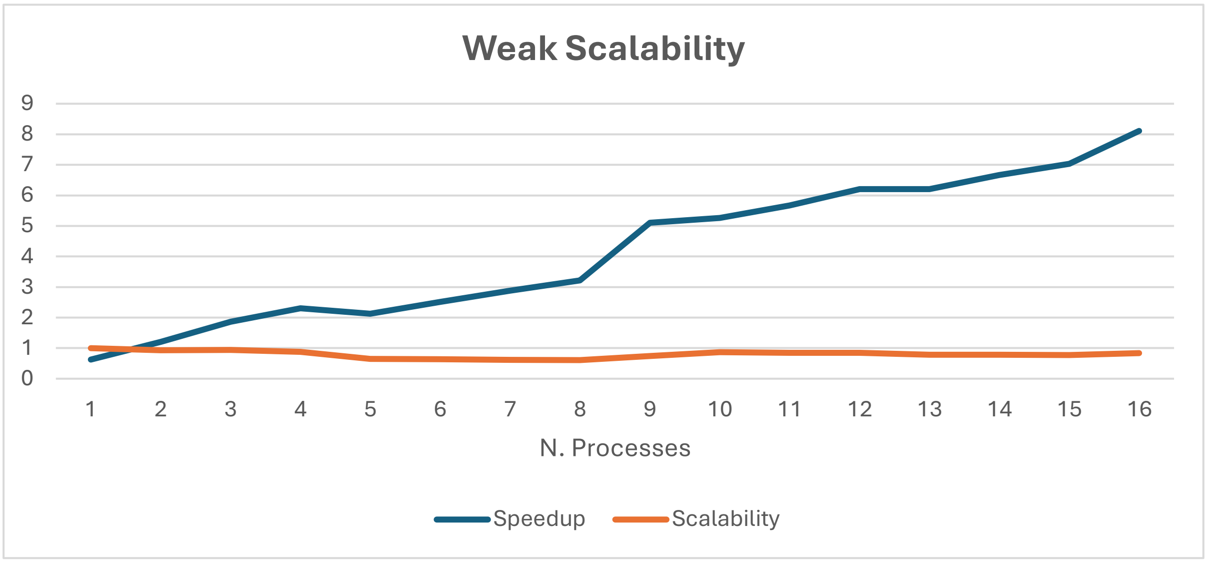
Strong scaling

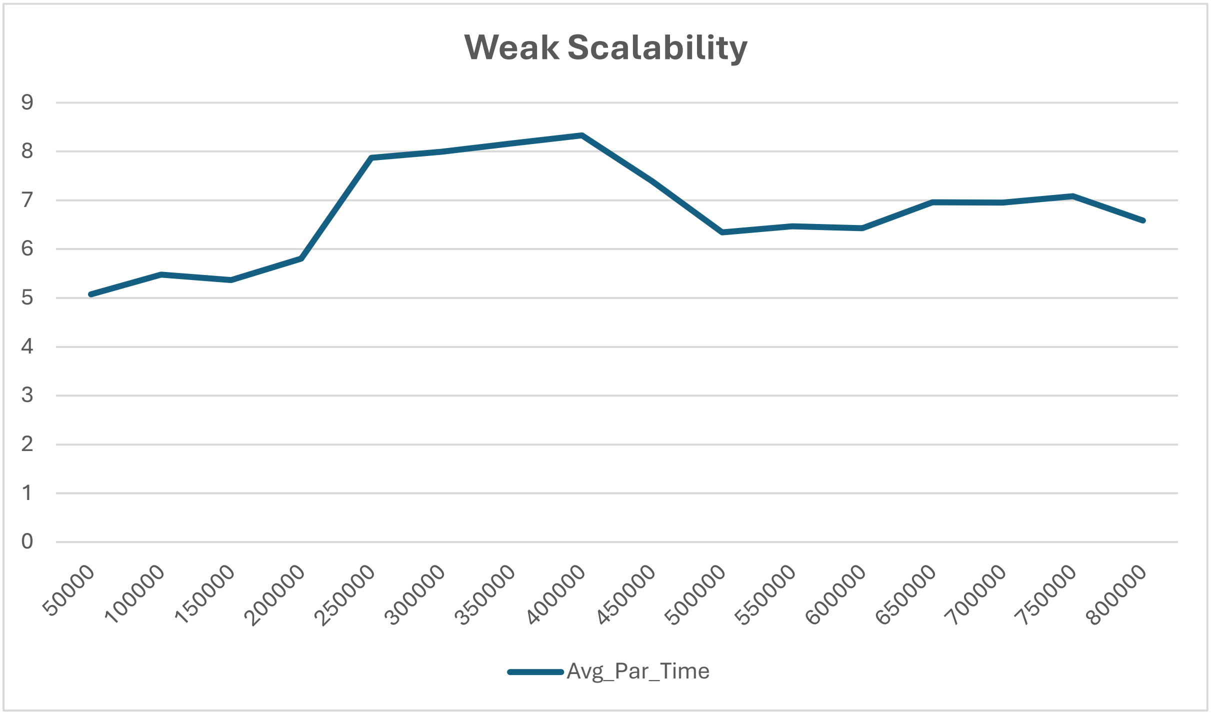




In the infra-regional test case, it results immediately obvious as the location of machines plays a fundamental role in communication. Machines were located in different states and the traffic data caused some delay in the exchange of information and a significant dropdown in speedup and scalability capacities.

Weak scaling



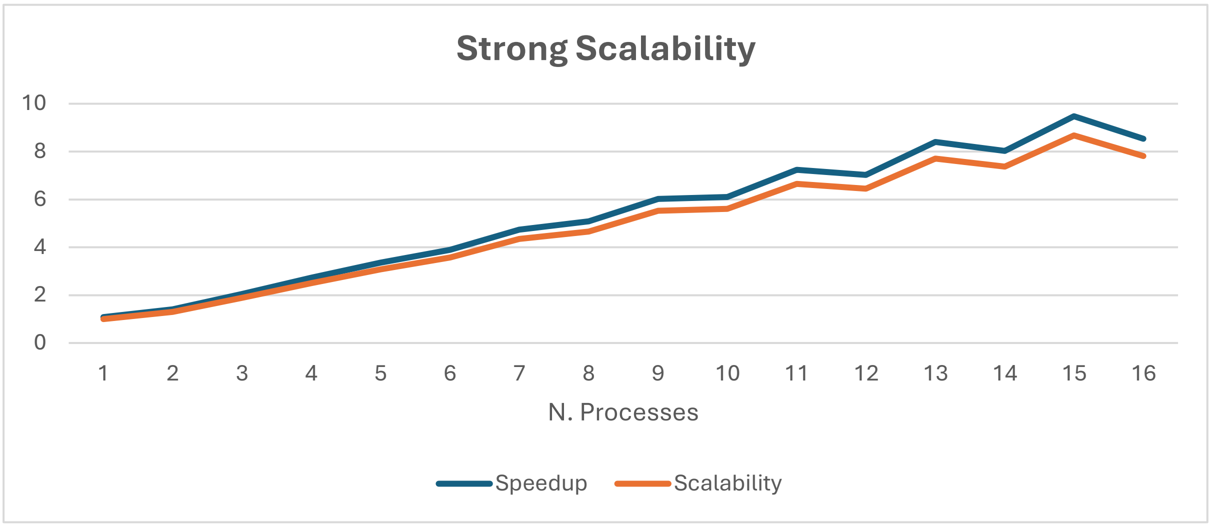


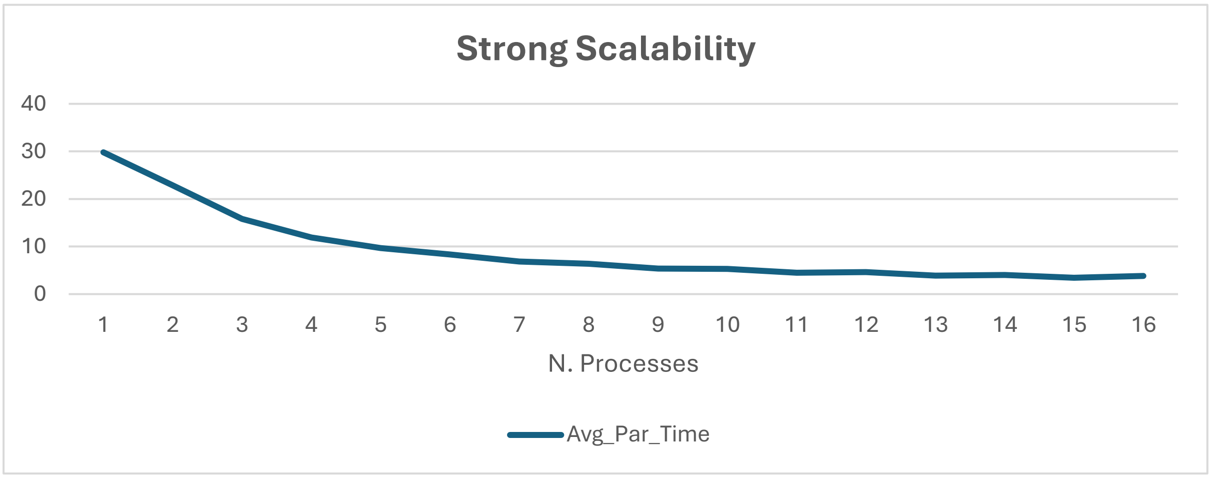
During the weak scalability test the proportional increase of workload together with the moment the test was performed (late night) probably helped in obtaining better results and maintaining an average parallel time, which was approximately constant.

Light cluster

Intra- regional

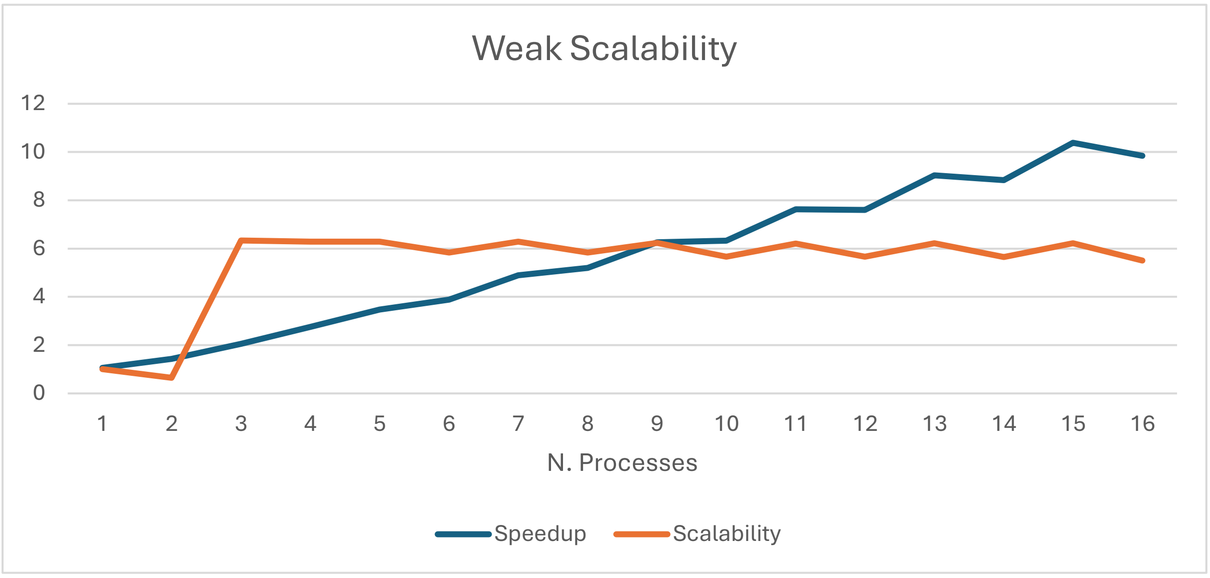
Strong scaling test

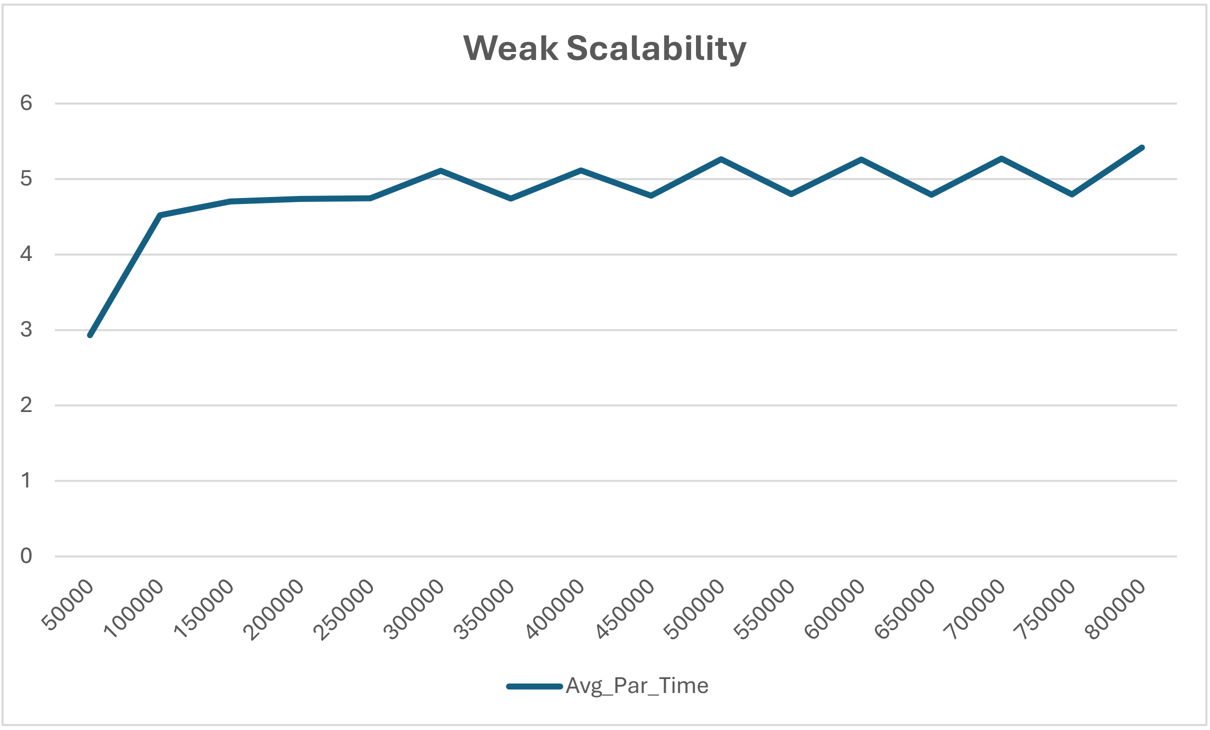




The light cluster test, running the algorithm in a cluster of machines located in the same region, shows very good results. Even if the speedup is not equal to the ideal because of the overhead in process communication, it tends to increase as more processes are involved. However, it is possible to notice how the advantage provided in including several processors tends to decrease and to flatten after the 8th process is considered.

Weak scaling test

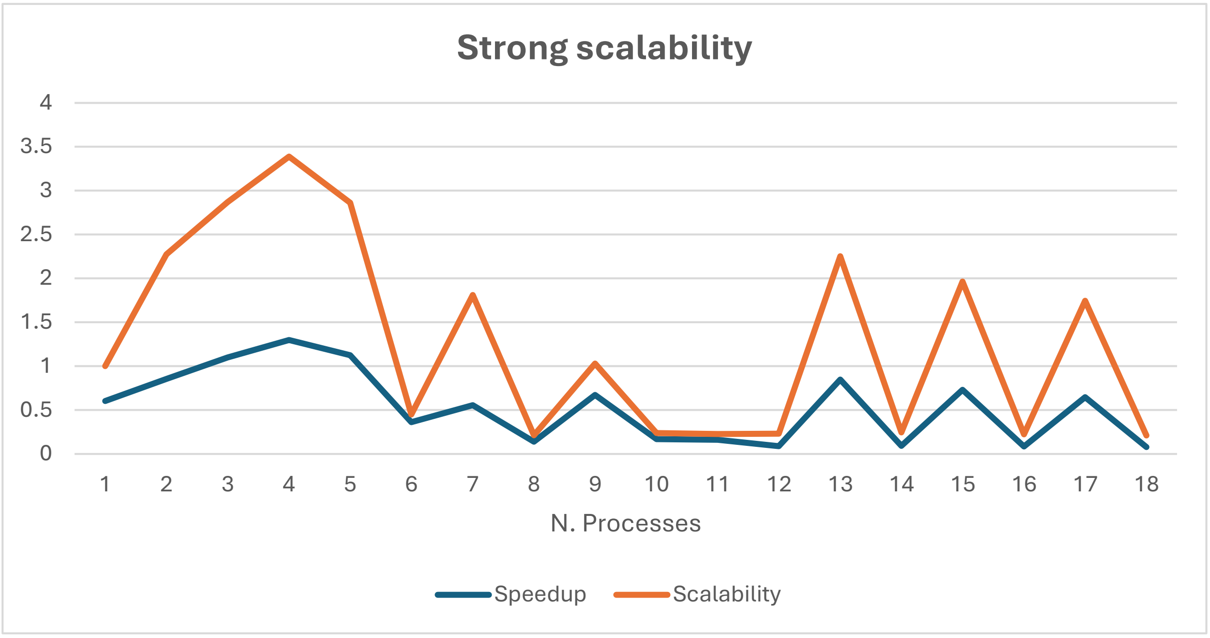


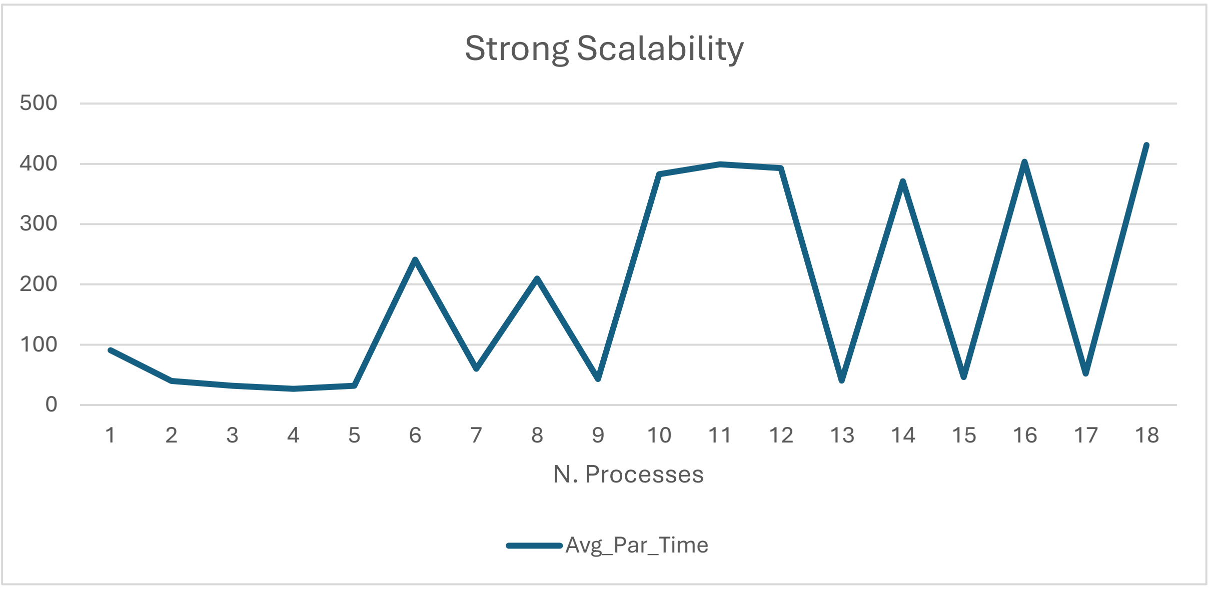


The average parallel time indicates a good response by the cluster to a proportional increase of the workload. The time increases slowly and this let think that this can be mainly due to the communication overhead among the machines guaranteeing a large speedup.

Infra-regional

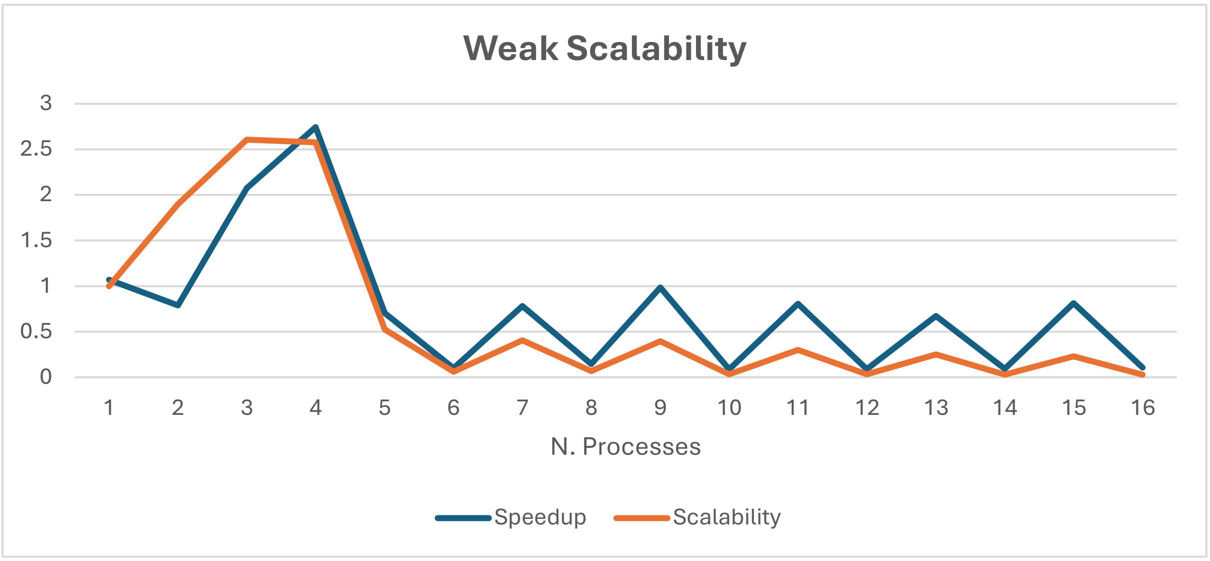
Strong Scaling test

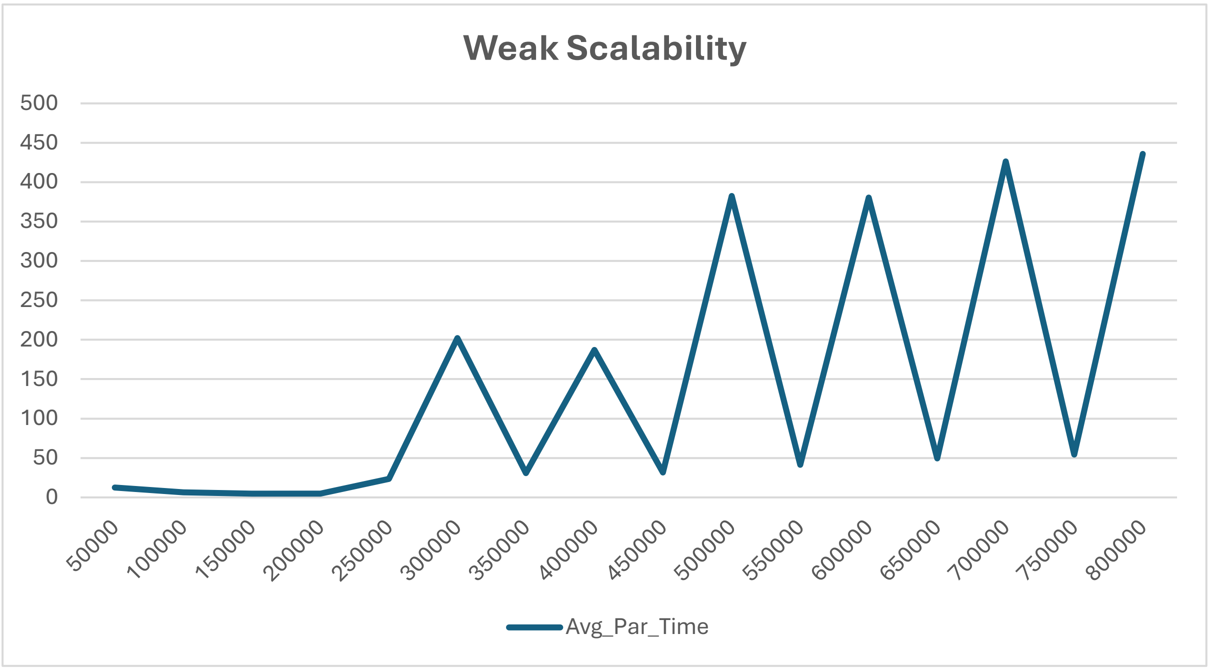




If the discontinuity of machines communications influenced the performances of a fat cluster when the two machines were located in different states, in the case of a light cluster of 8 machines this is even more true. The location in different states and continent make the communication very inconsistent, influencing a lot the response time of the different machines.

Weak scaling test





The same behavior occurs also in the weak scaling test. Performances tends to be similar to the ideal ones up to the involvement of the 4th process, while subsequently they tend to have a hiccup behavior. This is probably because machines were coupled and located in the same state and pairs of machines were located in different states or continents. However, performances continue to get worse as more processes from far countries are involved in the communication.