High Performance Computing assignment
Exercise 1

Piero Zappi

March 2024

1 Introduction

The performance of collective operations is critical to High Performance Computing and it's therefore crucial to design efficient implementations of them. This report focuses on the evaluation and the comparison of the performance of some of the algorithms implemented by the *OpenMPI* library to perform collective operations, specifically the broadcast and the barrier. In particular, the main objective of this study is to estimate the latency of the default *OpenMPI* implementations of these operations, by building prediction models based on the number of processes involved in the communication and, only in the case of the broadcast operation, on the size of the messages exchanged.

2 Experimental setup

The analysis of the broadcast and the barrier collective operations has been carried out using the *OSU benchmark* [4], executed on up to two whole EPYC nodes of the ORFEO cluster [5]. Each EPYC node is equipped with two AMD EPYC 7H12 CPUs, which are based on the "Rome" architecture and composed of 64 cores each organized in 4 NUMA regions, each having 64 GiB of memory; the total number of cores across the two nodes is therefore 256.
Both the default values of the number of warmup iterations and the total number of iterations have been increased to obtain more consistent and stable results.
In order to streamline the analysis, the data collection process has been automated by using some `bash` scripts, which are available on the author's GitHub repository [1].
Before starting to collect the data, we first conducted some test measures in order to establish which was the most proper task mapping setup to deploy during the following experiments. We observed that the `--map-by core` option of `mpirun` provided the lowest average latency values generally across all the employed algorithms, so we selected it as allocation of the *MPI* processes for our entire analysis.
For each selected algorithm of the broadcast and the barrier operations, we then collected many different measures of the average latency, by varying the number of processes from 2 to 256 and, only in the case of the broadcast operation, the size of the messages from 1 to 2^{17} bytes.

3 Broadcast

The broadcast operation (`MPI_Bcast`) is designed to broadcast some data from one process, called root, to all the others: the root sends the same message to all the processes in the

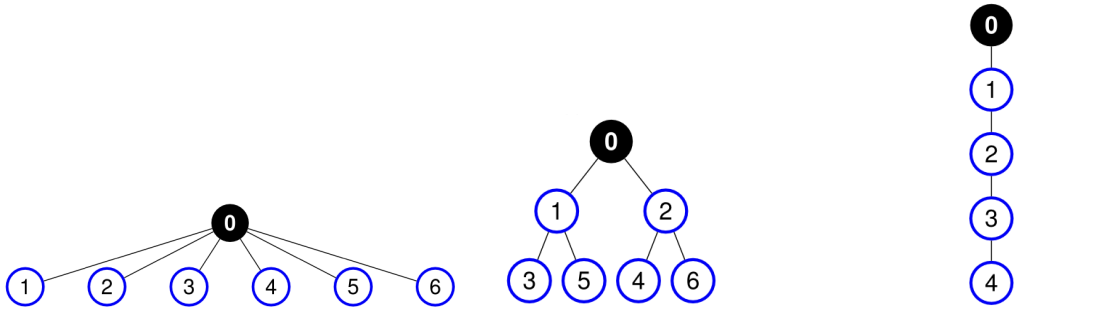
communicator; at the end of the call, the content of the root's communication buffer is copied to all the other processes.
As for all the other collective operations, its performance depends on many factors, such as the number of processes involved, the size of the exchanged message and the architecture of the system on which it is executed.

3.1 Algorithms

The *OpenMPI* library implements several algorithms to perform the broadcast operation; each algorithm defines a communication graph with a specific topology between the ranks in the communicator, which are the nodes in the graph.
In the specific case of this study, a total of four different algorithms have been selected:

- **Default**: it has not a specific implementation, since the library dynamically selects at runtime the most proper algorithm among the many available ones, basing its decision on aspects like the number of processes, the size of the message and the architecture of the system.
- **Flat tree**: also known as **basic linear**; the root node sends an individual message to all the other processes in the communicator. The data is transmitted to the child nodes without segmentation (figure 1 (a)).
- **Binary tree**: each process has two children and transmits the data to both of them employing segmentation; assuming for simplicity that the binary tree is complete, the height of the tree is $\log_2(P+1)$, where P is the number of ranks in the communicator (figure 1 (b)).
- **Chain**: each node has one child and the message is split into segments; the transmission of segments continues in a pipeline where the ith process receives from the $(i-1)$th and sends to the $(i+1)$th, until the last node gets the message (figure 1 (c)).

In our analysis, we assess and compare the performances of these four possible implementations for the broadcast operation.



(a) Flat tree algorithm (b) Binary tree algorithm (c) Chain algorithm

Figure 1: broadcast algorithms' topologies

3.2 Average latency analysis

We first provide, for each algorithm, a 3D heatmap of the average latency against the number of processes and the size of the message, in order to visualize and understand some general behaviours of the broadcast collective operation.

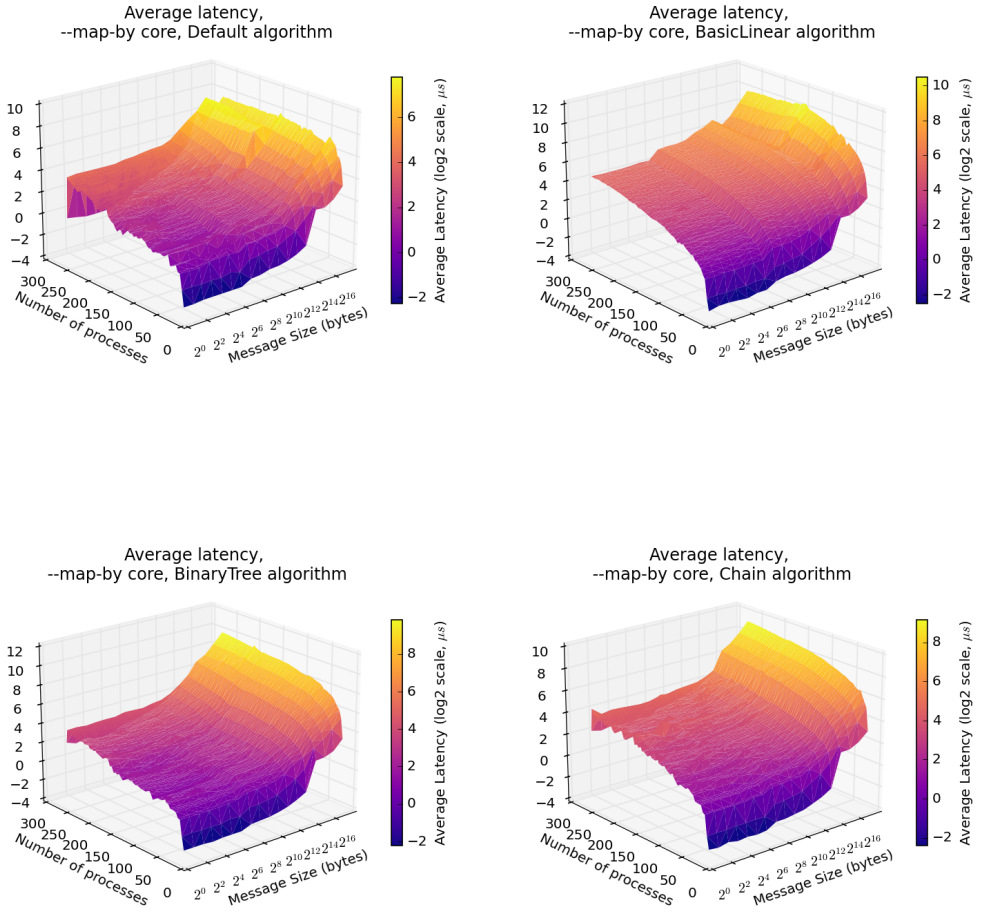


Figure 2: 3D heatmaps for each considered algorithm of the broadcast operation

By looking at figure 2, we can make some first observations: as expected, the average latency increases with the number of processes in the communicator and the size of the messages exchanged. In addition, we can notice comparable trends across all the four selected algorithms; only the default algorithm showcases some spikes, which are probably associated to changes of the algorithm selected at runtime. Finally, the basic linear algorithm seems to be the worst performing one; this is also in agreement with our expectations, since the flat tree algorithm is the simplest one as it does not distribute the communication among the processes and it does not exploit segmentation.
We then continued our analysis comparing the average latency values against the number of processes for the four algorithms, using three different fixed message sizes. By observing figure 3, we can draw the following conclusions: as expected and as already suggested by the previous plot, the basic linear algorithm is by far the worst performing one for all the three different message sizes we considered; this is due to the reasons related to the algorithm's simplicity that we already discussed. On the other hand, the binary tree and the default implementations showcase very similar and comparable behaviours, being the best performing algorithms for compact message sizes; however, the chain algorithm's performance gets progressively better as the size of the messages increases, outperforming also the binary tree and the default algorithms when the size of the exchanged message gets sufficiently large. This is probably related to the chain algorithm's ability of splitting the messages into segments.

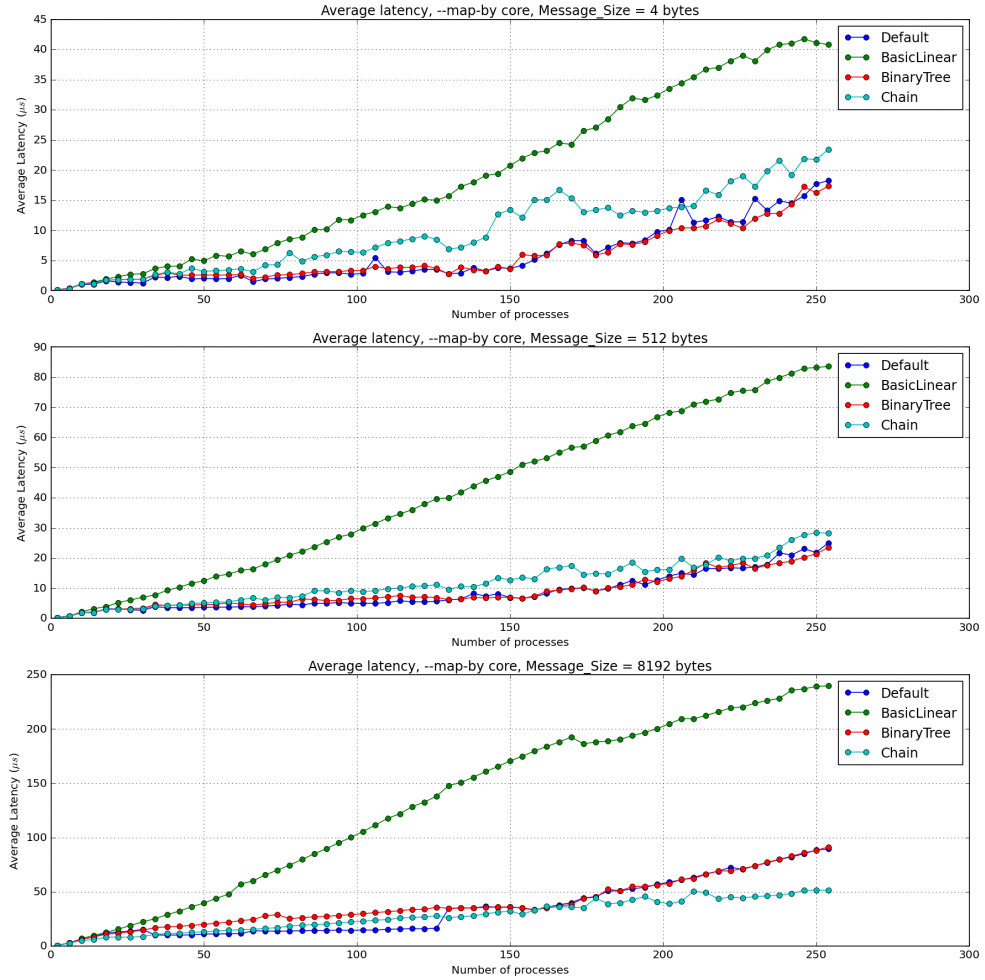


Figure 3: comparison of the broadcast algorithms varying the size of the message

3.3 Performance models

We built a prediction model in order to estimate the average latency (`avg_lat`) of the broadcast operation; in particular, we decided to implement a linear regression model for each algorithm, by fitting the previously collected data. The number of processes in the communicator (`proc_num`) and the size of the exchanged message (`mess_size`) were selected as predictors for the model, while we decided to not insert any intercept in it: in fact, we assumed the latency to be 0 in the case of a message of size 0 bytes transmitted among 0 processes.
All the algorithms' models share the same basic structure, which can be represented by the following relation:

$$\log_2(\texttt{avg\_lat}) = \beta_1 * \texttt{proc\_num} + \beta_2 * \log_2(\texttt{mess\_size}) + \beta_3 * \log_2(\texttt{mess\_size})^2$$

We applied the logarithmic transformation to `avg_lat` and `mess_size` consistently with the fact that both these variables showcased a skewed distribution; moreover, we added the final quadratic term because the plots in figure 2 showed that the relationship between the logarithm of the average latency and the one of the size of the messages was not linear.

Finally, it is important to note that the estimates of the parameters β_i, $i = 1, 2, 3$, of each algorithm's model are platform dependent.
The following table 1 reports the various models' summaries, which include the parameters' estimates and the models' adjusted R^2, which we used as a metric to assess the goodness of the models.

Algorithm	β_1	β_2	β_3	Adj. R^2
Default	0.0147	−0.1461	0.0278	0.970
Basic Linear	0.0215	0.0980	0.0148	0.970
Binary Tree	0.0155	−0.2100	0.0358	0.979
Chain	0.0182	−0.1212	0.0262	0.969

Table 1: summaries of the performance models for the broadcast algorithms

All the models' coefficients are statistically significant; in addition, by looking at the adjusted R^2 values reported in table 1, we can affirm that all the performance models we developed are able to predict with very good accuracy the average latency of their associated algorithm. Finally, regarding the models' interpretation, each estimated coefficient can be interpreted as the expected change in the response variable $\log_2(\texttt{avg\_lat})$ when its corresponding predictor increases by one unit and all the others are kept constant.
For the sake of clarity and completeness, we provide the following plots to visualize the models and their performances, by plotting the measured and the predicted data.

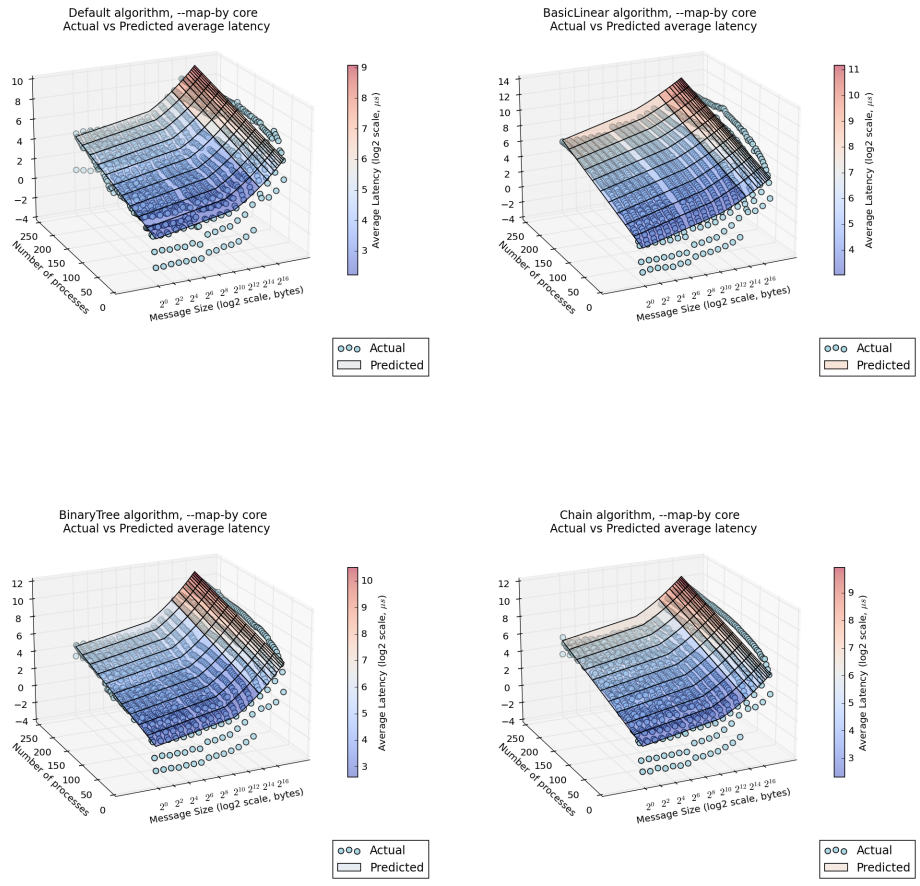


Figure 4: comparison between the actual and the predicted data for each considered algorithm of the broadcast operation

4 Barrier

The barrier operation (`MPI_Barrier`) is used to synchronize a group of processes; it guarantees that no processor continues to the next stage of the code until all the processors have finished the current stage.
Unlike the broadcast operation, the barrier does not involve the exchange of messages of varying sizes, therefore its performance only depends on factors like the number of processes involved and the architecture of the system on which it is executed.

4.1 Algorithms

As in the case of the broadcast operation, the *OpenMPI* library implements several algorithms to perform the barrier; for this study, a total of four of these algorithms have been selected:

- **Default**: similarly to the broadcast default algorithm, it has not a specific implementation and the library dynamically selects at runtime the most proper algorithm to use among the available ones.
- **Linear**: all the processes report to a pre-selected root; once all the processes in the communicator have reported to the root, the latter sends a releasing message to all the nodes (figure 5).
- **Tree**: it implements a tree-like hierarchical structure in order to speed up the operation by lowering the number of required steps (figure 6).
- **Double ring**: a 0 bytes message is sent from a pre-selected root circularly to the right; a node can continue to the next stage of the code only after it receives the message for the second time.

In our analysis, we assess and compare the performances of these four possible implementations for the barrier operation.

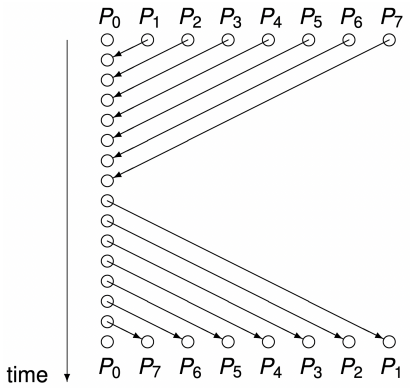


Figure 5: barrier linear algorithm

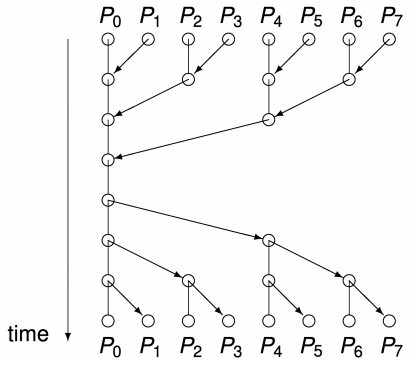


Figure 6: barrier tree algorithm

4.2 Average latency analysis

In order to assess the performances of the four algorithms, we plot the average latency values we collected for each algorithm against the number of processes in the communicator.

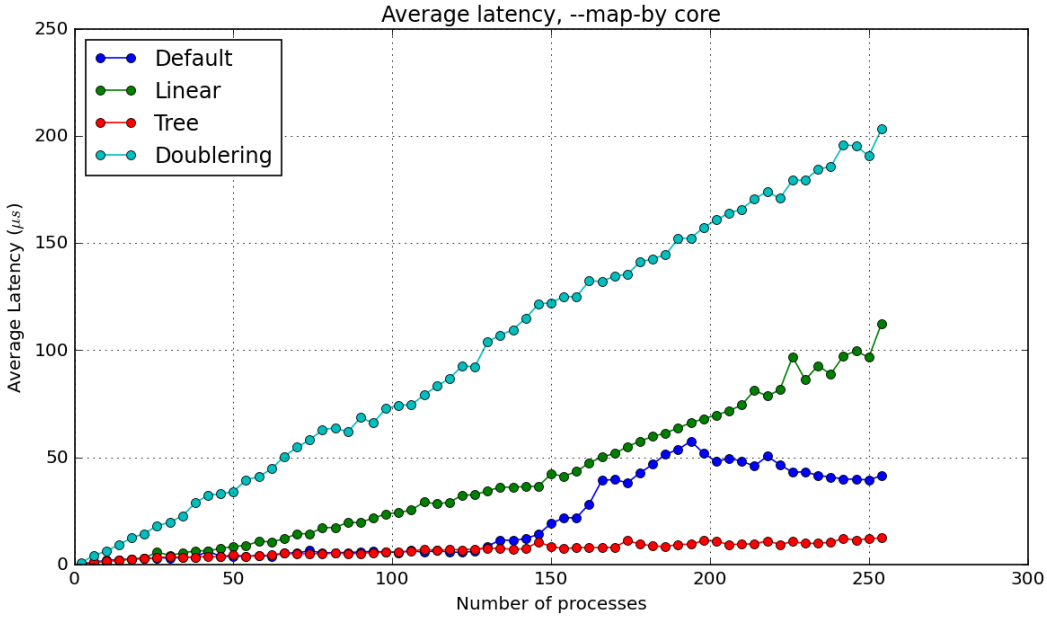


Figure 7: comparison of the barrier algorithms' performances

By looking at figure 7, we can immediately notice that, as expected, the average latency increases with the number of processes for all the four considered algorithms. The tree algorithm stands out as the best performing one; this is also in agreement with our expectations, because of its ability of reducing the number of steps required by the barrier operation. On the other hand, the double ring algorithm seems to be by far the worst performing one. In addition, we can see that in the case of the linear algorithm the slope of the measured data changes at around 128 processes, which is the number of cores of an EPYC node of the ORFEO cluster. In fact, this increment of the slope corresponds to the allocation of a new node in the architecture and the communication between two different nodes generally takes longer than the one between processes allocated within the same node. Finally, we can observe that the default algorithm showcases a very strange and unexpected trend: in fact, after a certain point the average latency starts to decrease with the number of processes in the communicator. We were not able to provide a clear explanation for this peculiar behaviour; however, the probable cause could be related to a change of the algorithm selected at runtime by the library.

4.3 Performance models

We developed a prediction model in order to estimate the average latency (`avg_lat`) of the barrier operation; specifically, also in this case we decided to implement a linear regression model for each algorithm, except for the default one, by fitting the previously collected data. We selected the number of processes in the communicator (`proc_num`) as the only predictor for the models and we opted to not insert any intercept in them: in fact, we

assumed the latency to be 0 when the number of processes is 0. Moreover, in the case of the barrier operation it was not necessary to apply the logarithmic transformation to `avg_lat`, since its distribution was not skewed.
Differently from the case of the broadcast operation, each algorithm's model has its unique structure. The linear algorithm's one can be expressed by the following relation, where we imposed $x \coloneqq$ `proc_num`:

$$\texttt{avg\_lat} = \beta_1 * x + \beta_2 * I(x > 128) + \beta_3 * (x * I(x > 128))$$

with:

$$I(x > 128) = \begin{cases} 1 & \text{if} \quad x > 128 \\ 0 & \text{otherwise} \end{cases}$$

This model, which introduces an indicator variable I and an interaction term along with the usual predictor `proc_num`, tries to reflect and to reproduce the specific behaviour of the data we collected for the linear algorithm, which we described in the previous paragraph: the slope of the measured data increases at around 128 cores, due to the allocation of a new node.
In the case of the tree algorithm, which is non-linear, we enhanced the model by adding a quadratic term:

$$\texttt{avg\_lat} = \beta_1 * \texttt{proc\_num} + \beta_2 * (\texttt{proc\_num})^2$$

Finally, the prediction model for the double ring algorithm has the following simple structure:

$$\texttt{avg\_lat} = \beta_1 * \texttt{proc\_num}$$

As for the broadcast algorithms, it is important to underline the fact that the estimates of all the parameters β_i of each algorithm's model are platform dependent.
The following table 2 reports the different models' summaries, which also in this case include the parameters' estimates and the models' adjusted R^2.

Algorithm	β_1	β_2	β_3	Adj. R^2
Tree	0.0735	-0.0001	$--$	0.987
Linear	0.2301	-47.0331	0.3585	0.997
Double Ring	0.7853	$--$	$--$	0.999

Table 2: summaries of the performance models for the barrier algorithms

All the models' coefficients are statistically significant and, as shown by the adjusted R^2 values reported in table 2, the performance models we built for the barrier operation are all able to predict with high accuracy the average latency of their associated algorithm.
The following plots enable to visualize the various models and their performances, by plotting the collected and the predicted data.

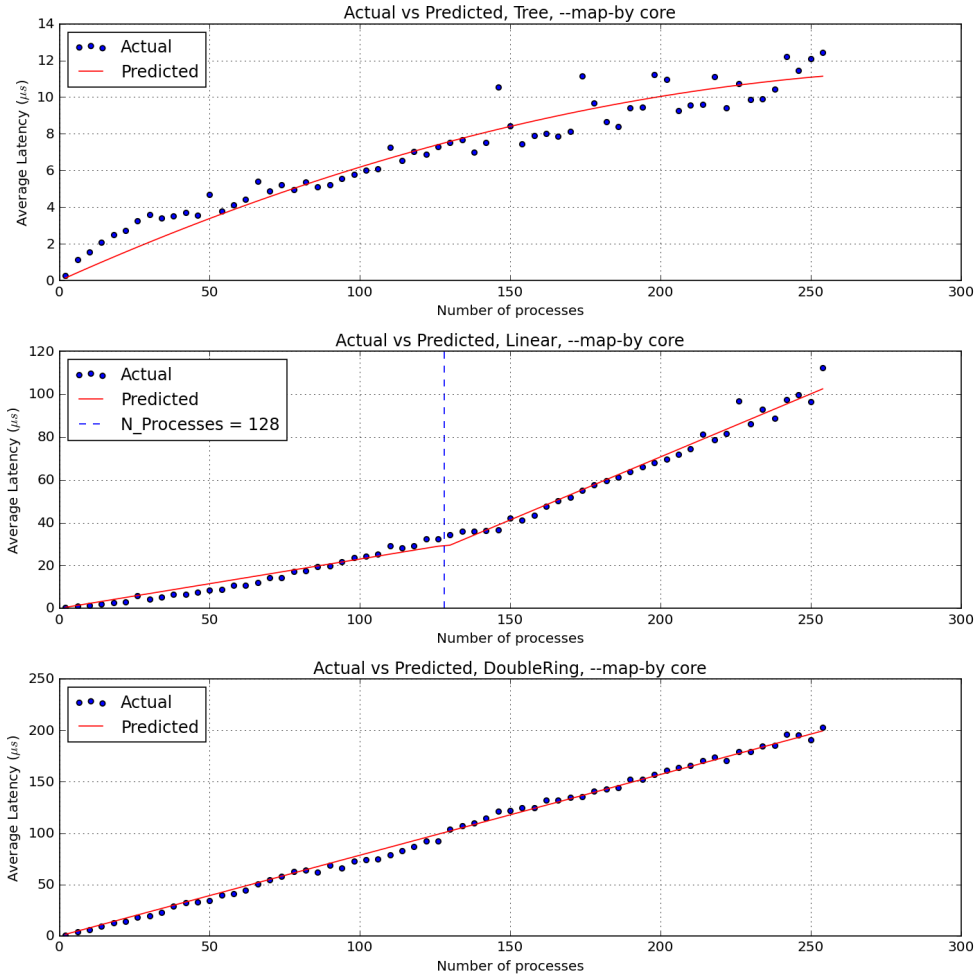


Figure 8: comparison between the actual and the predicted data for each considered algorithm of the barrier operation

5 References

1. Piero Zappi, `High_Performance_Computing_Assignment` GitHub repository. URL: `https://github.com/PieroZ01/High_Performance_Computing_Assignment/tree/master`

2. Emin Nuriyev, Juan-Antonio Rico-Gallego, Alexey Lastovetsky, *Model-based selection of optimal MPI broadcast algorithms for multi-core clusters*, Journal of Parallel and Distributed Computing, 165 (2022) 1-16

3. Pješivac-Grbović, et al., *Performance analysis of MPI collective operations*, Cluster Computing 10 (2007): 127-143

4. The Ohio State University, OSU Micro-Benchmarks 7.3, 2023. URL: `https://mvapich.cse.ohio-state.edu/benchmarks/`

5. AREA Science park, ORFEO documentation, 2023. URL: `https://orfeo-doc.areasciencepark.it`