Implementation of MPI collective communication algorithms

Edoardo Insaghi

1 Introduction

The openMPI library is a widely used implementation of the Message Passing Interface, offering a number of features for parallel programming. Among its possibilities, openMPI provides multiple algorithms to perform collective operations, which play a fundamental role in parallel computing applications. Collective operations involve communication and coordination between processes, facilitating data exchange and synchronization through algorithms that rely upon point to point communications.

This report presents an implementation of three different algorithms that perform the broadcast collective communication between MPI processes, and their comparison in terms of how much time is needed for the communication to be concluded. There are a number of degrees of freedom that can be taken into account to obtain a more detailed model for the comparison, they include the number of processes involved in the communication, the way the physical resources are allocated, and the size of the data. The evaluation of the algorithms is executed on two whole EPYC nodes of the ORFEO cluster. Each of the two nodes is equipped with two AMD EPYC 7H12 CPUs, each possessing 64 CPU cores, for a total of 256 cores across the two nodes.

To collect the data, the processes are synchronized through a barrier before and after the communication is executed, and then the time is collected by the root process. This synchronization introduces some overhead, but hopefully the results are still valid as the overhead should depend only on the number of processes, and be the same regardless of the algorithm. One other method that has been tried to collect the data consisted of taking the maximum time recorded by any process to estimate the end of the communication, but was abandoned as resulted in extremely variable estimates.

The algorithm of choice are the flat tree algorithm, in which a root process sends the buffer to every other process in the communicator, the chain tree algorithm, in which each process receives the message from the previous and sends it to the next, and the binary tree algorithm, in which each process receives the buffer from a parent process and sends it to two children processes. To obtain the time for each run the number of warmup iterations has been set to one thousand while the number of recorded iteration was five thousand. This numbers have been found after some fine tuning and they represent a good tradeoff between the reliability of the estimates and the resources needed to obtain them. While at first an attempt to use the mean as the estimator for the comparison was made, this choice led to extremely unreliable results even at a much higher number of iterations, and therefore the estimator chosen in the end was the median of the observation. This comes at a price in terms of computational resources, since it requires to sort a vector at each run, but the gains in terms of quality of the results made the change definitely worth it.

The next paragraphs shows two kind of comparisons of the algorithms, in the first one it is addressed the weak scaling of the algorithms, and therefore the size of the data is kept fixed while both the number of processes involved in the communication and the allocation of the computing resources vary. The second one, which gets the name of strong scaling, instead compares how the time needed to conclude the communication varies with the size of the buffer, keeping fixed the number of processes.

2 Weak Scaling

The following plots show how the total time needed to conclude the communication varies with the number of processes involved in the communication for the three algorithms and two different mapping of the resources, by node and by socket. The size of the message is fixed at 32 bytes, which is so small that the time could be interpreted as the latency of the communication (which measures the time for a zero sized message to be delivered). Other message sizes (up to 8KB, which is one fourth of the size of the L1 cache of EPYC nodes' cores) have been tried but did not add further insight to the analysis.

The first one shows the data collected for the flat tree algorithm. The data shows that, as expected, the total time increases with the number of processes in the communicator, but some differences emerge between the different algorithms and resource allocations. Both implementations of the algorithms perform better when resources are allocated by core rather than by node, and this is true especially for smaller numbers of processes. This is certainly expected as less time is needed to send the buffer to closer cores rather than distant ones. Also, when the computational resources are allocated by core there seems to be a jump in the total time at 128 processors, which is the point at which any two processes of the two nodes start communicating. As for the different implementations, they seem to be behaving similarly, with the default version being slightly faster with the allocation by core, and the opposite happens with the allocation by node.

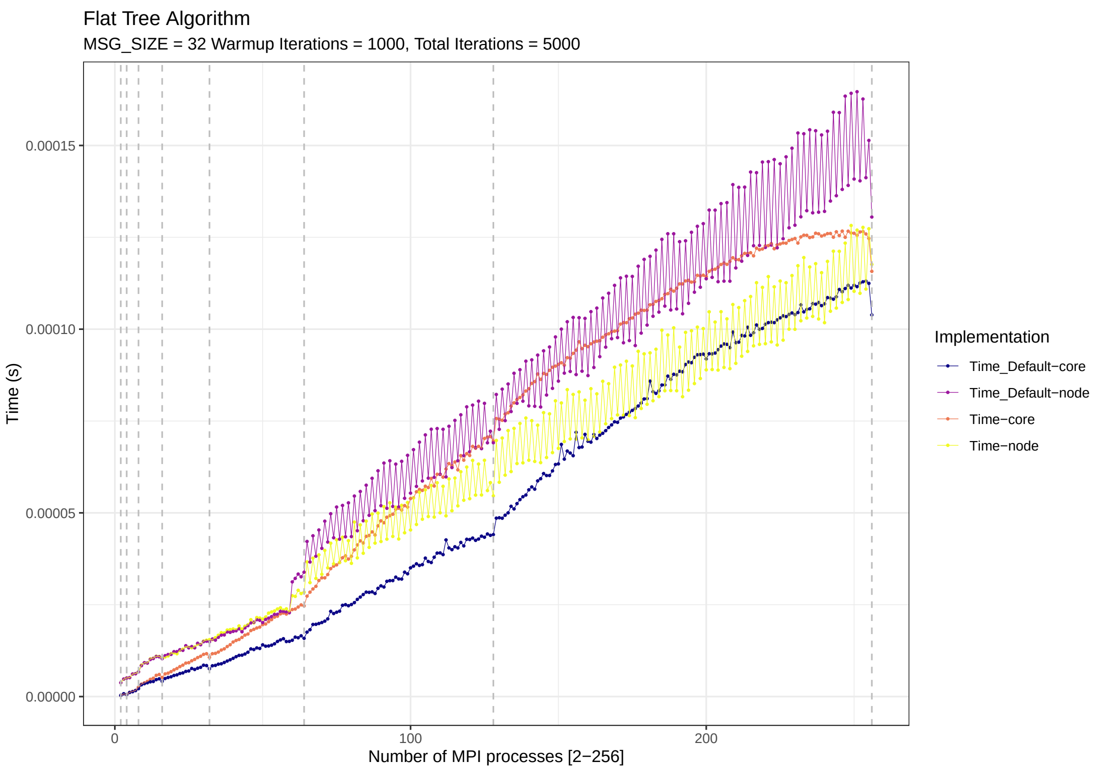

Figure 1: Weak Scaling for the Flat Tree Algorithm

The same data is collected for the chain tree algorithm, for which the results are somewhat dissimilar from the previous. Here clear differences emerge both with respect to the implementation of the algorithm and the allocation of the resources. In the chain tree algorithm each process sends the buffer to the next in line, if the mapping of the cores is made by nodes the message has to travel the longest possible distance and this explains a huge portion of the variability of the data. With respect to the implementation, the default ones here perform much better than their counterparts. Compared to the flat tree algorithm, the chain tree seems to be performing worse on average, but the default implementation is by far the fastest when the resources are mapped by core, which is to be expected since with this algorithm the distance travelled by the buffer is shorter compared to the flat tree case.

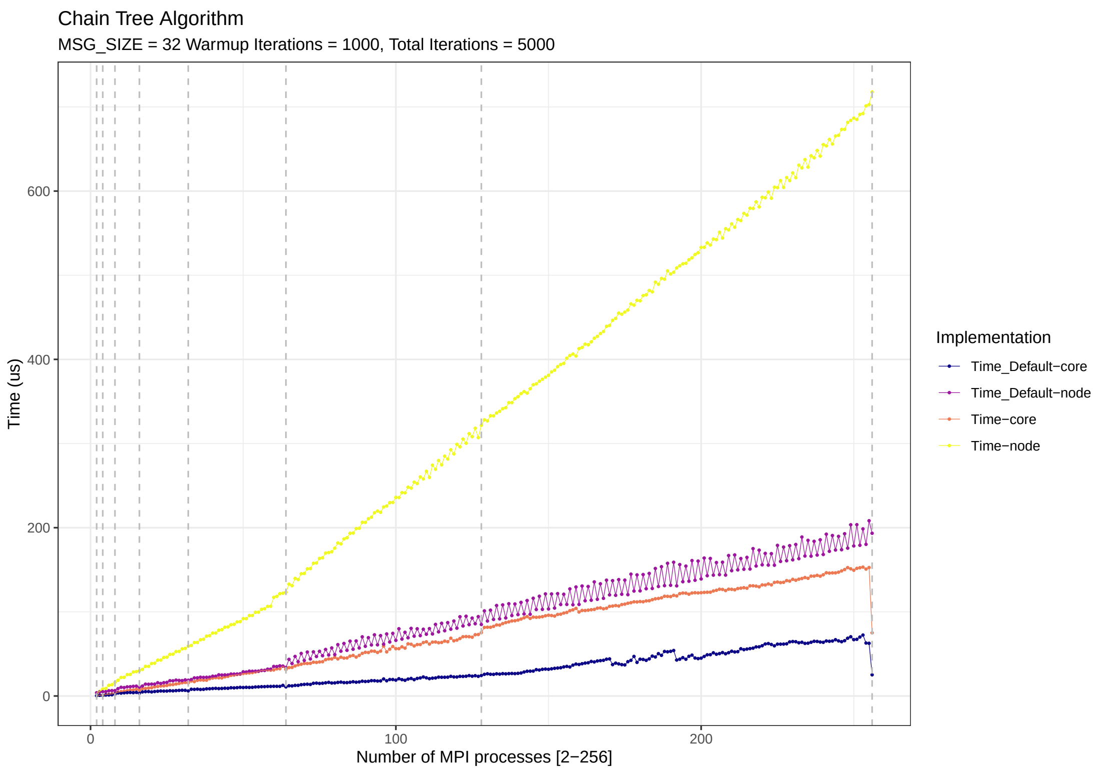

Figure 2: Weak Scaling for the Chain Tree Algorithm

Once again, the following plot shows the same data for the binary tree algorithm. This time, huge differences emerge between communications involving an even and odd number of processes when the computational resources are mapped by node. This pattern was actually visibly evident from the data of the flat tree algorithm, but this time this phenomenon seems to explain a larger portion of variance. This is mostly due to the fact that the binary tree algorithm outperforms the linear, and thus the even-odd discrepancy here looks bigger. Actually they are rather similar across all three algorithms, spanning from $10\mu s$ to $50\mu s$ depending on the algorithm and the number of processes involved in the communication. With respect to the different implementations of the algorithms instead, when the cores are mapped by core they seem to perform pretty much identically, while the default implementation becomes faster when the resources are allocated by node. Once again with the mapping by core a jump can be observed when the number of processes crosses 128, which is when the two nodes start communicating.

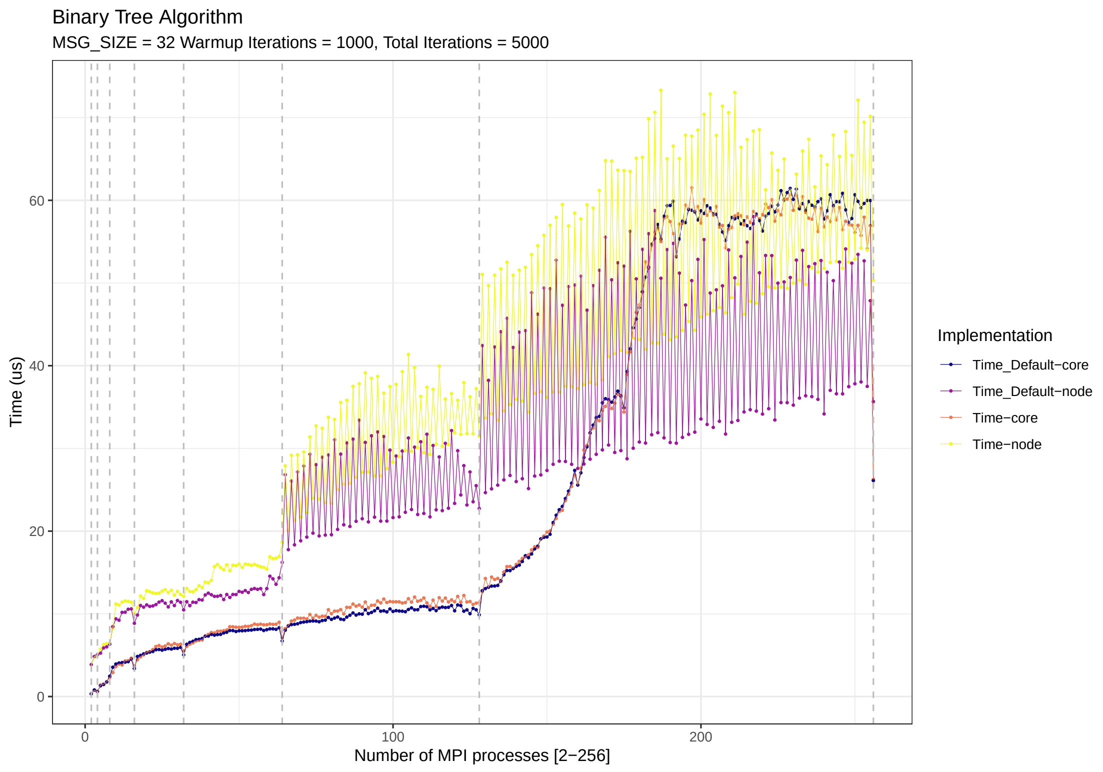

Figure 3: Weak Scaling for the Binary Tree Algorithm

3 Strong Scaling

As mentioned before, this section addresses the problem of the strong scaling for this problem, the following plots show how the total time needed to conclude the communication varies when the size of the data changes. This data has been collected for different number of processes, 32, 64 and 128. This aims at extracting information about possible interactions between the number of processes, the size of the data and the different implementations of the algorithms.

From the plots it emerges that when the size of the data increases it becomes the major factor in explaining the variability of the data, but when performance becomes critical the choice of the algorithm remains crucial, in fact, with only 32 processes involved in the communication a factor of four separates the best from the worst algorithm when the size of the data reaches the size of 1MB. This number increases to a factor of twelve with 128 processes.

Data of this size does not fit into the L2 cache of the EPYC nodes' cores (512KB), and while increasing the size of the buffer up to the size of the L3 cache (16MB) could have revealed some more interaction possibly due to the more time required to fetch data from a more distant cache, it would have also required too much time and computational resources on the nodes. Also, each algorithm seems to follow a pretty clear linear trend for each number of processors, and it sounds reasonable to assume that it would continue like this doubling the size of the data a few more times.

What clearly emerges from the plots is that the default implementations of the algorithms perform better that their counterparts, which is expected. The default implementations of the flat tree and chain tree algorithms reach similar levels of performance, while the default implementation of the binary tree is by far the fastest of all. This difference becomes more prominent when the number of processes also increases as the binary tree is characterized by a better weak scaling.

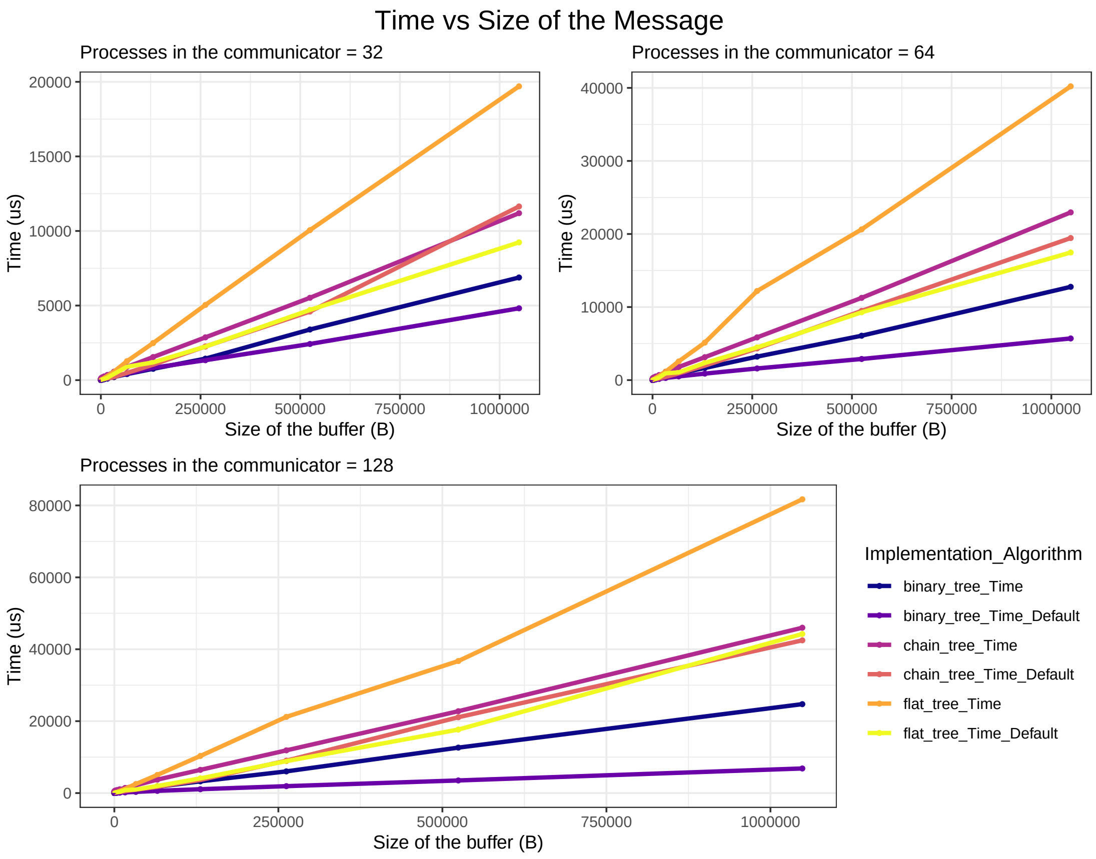

Figure 4: Strong Scaling of the Algorithms, Allocation by Node

In order to better understand the data, it might be useful to resort to a simple linear model that predicts the total time of the communication using the other variables at play. After a number of attempts, the model that worked best on the data uses the product of the size of the data and the number of processes as a covariate for the total time of the communication. This new variable represents the total amount of data to be moved during the communication. The intercept has been removed from the model as it is reasonable to assume that a communication involving no processes and a message of size zero takes no time to complete. Hereafter are reported the results of the model.

Table 1: Results of the Linear Model

	Estimate	Std. Error	t value	Pr(>|t|)
binary_tree_Time	1.867e-04	7.594e-06	24.591	<2e-16
binary_tree_Time_Default	5.328e-05	7.594e-06	7.016	8.01e-12
chain_tree_Time	3.490e-04	7.594e-06	45.951	<2e-16
chain_tree_Time_Default	3.160e-04	7.594e-06	41.611	< 2e-16
flat_tree_Time	6.004e-04	5.010e-06	119.834	<2e-16
flat_tree_Time_Default	3.828e-04	5.010e-06	76.398	<2e-16

The numbers do not seem to add much information, as they mostly confirm what the plots showed, but the model shows that it is actually able to explain the majority of the variance of the data [$adj R^2 = 0.98$], which suggests strong evidence in favour of the multiplicative effect of number of processes and size of the data on the total time of the communication. In particular, the multiplicative model outperformed the additive model, whose $adjR^2$ only reached 0.57.

The same procedure was used to gather and analyse the data with the computational resources allocated by core, and they yielded interesting results. As before, the default implementation outperform the others, but this time the performance of the chain tree algorithm implemented by openMPI are comparable with the results obtained for the binary tree. Also, the difference between the default implementation of the binary tree and its counterpart seems to be less pronounced than before, and the default flat tree algorithm also becomes more competitive.

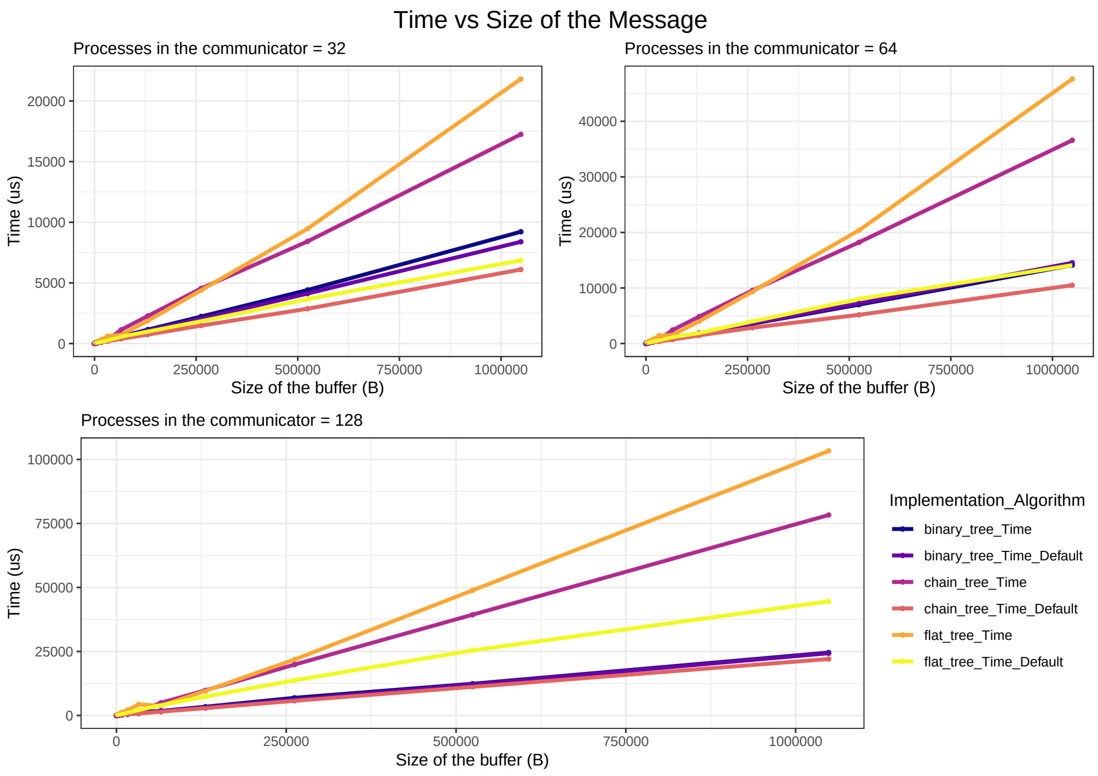

Figure 5: Strong Scaling of the Algorithms, Allocation by Core

As before, here are presented the results for the linear model in which the total amount of data is used as the covariate while the time for the communication to be concluded plays the role of the response variable.

	Estimate	Std. Error	t value	Pr(>|t|)
binary_tree_Time	1.939e-04	0.0000	40.13	<2e-16
binary_tree_Time_Default	1.914e-04	0.0000	39.62	<2e-16
chain_tree_Time	5.736e-04	0.0000	118.73	<2e-16
chain_tree_Time_Default	1.642e-04	0.0000	34.00	<2e-16
flat_tree_Time	7.332e-04	0.0000	151.77	<2e-16
flat_tree_Time_Default	3.152e-04	0.0000	65.23	<2e-16

In accordance to the plots, the chain tree algorithm is the most robust when the computational resources are allocated by core. This makes sense as for 128 allocated by core all communication happen within a single node. Also in this case the $adjR^2$ of the model indicates an excellent adaptation to the collected data as it rises at the value of 0.992.