High Performance Computing 2023 - Excercise 1

Marco Zampar - SM3800032

September 25, 2024

1 Problem Statement

A profiling study [3] reports that in average 80% of the total execution time of MPI applications is consumed by MPI collective operations. That is why significant research efforts have been invested in the design and implementation of efficient collective algorithms aimed to improve the performance of collective operations. Unfortunately, there is no single collective algorithm optimal in all situations, depending on many factors including the physical topology of the network, number of processes, message sizes.

With this framework in mind we introduce the aim of this project: estimate the latency of default openMPI implementation, varying the number of processes and the size of the messages exchanged and compare it with the values obtained using different algorithms and different mappings of the processes to the hardware.

Open MPI uses internal logic to decide which algorithm to apply for a given situation, choosing the one that it believes will provide the best performance without user intervention. Unfortunately this is not always true: as reported in [3], sometimes there is a remarkable difference between the latency of the default and the best algorithm; that is why the comparison with other algortithms and different mappings of the processes is important and informative.

2 Computational Resources

The computational resources employed are 2 THIN nodes of the ORFEO cluster, each node has 2 CPUs of 12 cores in 1 single NUMA region.

More informations about the ORFEO cluster can be found here [1].

3 Broadcast Algorithms

In the broadcast operation (MPI_Bcast) a process called root sends a message with the same data to all processes in the communicator. Messages can be segmented in transmissions. Segmentation of messages is a common technique used for increasing the communication parallelism, hence the performance. It consists of dividing up the message into segments and sending them in sequence.

There are many implementations of the algorithm, the ones I chose to analyse are the Flat Tree or Linear, Chain Tree and Binary Tree algorithms, well described in [3].

During the execution of the experiments, we noted that the Linear algorithm behaved very similarly to the Binomial, then I decided to keep only the Linear, which is more easy interpretable.

A brief description of the algorithms:

- Flat Tree: the algorithm employs a single level tree topology where the root node has P-1 children. The message is transmitted to child nodes without segmentation.
- Chain Tree: each internal node in the topology has one child. The message is split into segments and transmission of segments continues in a pipeline until the last node gets the broadcast message. i-th process receives the message from the $(i-1)$-th process, and sends it to $(i+1)$-th process.
- Binary Tree: each internal process has two children, and hence data is transmitted from each node to both children. Segmentation technique is employed in this algorithm. If we assume that the binary tree is complete, then $P = 2H - 1$ where H is the height of the tree, $H = \log_2(P + 1)$.

Those operations can be either blocking or non-blocking, as suggested in the excercise assingment, I run the tests with the executable `osu_bcast` in the `blocking` folder of the OSU benchmark, so the point-to-point message that constitute the program are actually blocking. This executable returns the latency of the communication in μs.

Fore every broadcast operation there are two parameters that regulate the final latency: the message size and the number of processes, moreover, I tried different mappings of the processes to `core-socket-node` through the flag of `mpirun -map-by`: to conduct this analysis I decided to fix two of these parameters to analyse the relation of the third with the latency. In the Appendix, Figure 13 shows a 3d plot with message size and number of processes on the x and y-axis.

3.1 Latency against message size, fixing the number of processes

In Figure 12 of the Appendix we can see that the scaling with respect to the message size is actually linear. This holds also for the other kinds of allocation of the processes, we didn't include the other plots for brevity.

3.2 Latency against number of processes, fixing the message size

3.2.1 Mapping by core

Setting the `--map-by` flag of `mpirun` to `core`, each process is mapped to the available cores in order, this means that the total 48 processes will be distributed in this way: process 0 is bound to core 0 of node 0 and so on, process 24 is bound to core 0 of node 1 and so on.

In Figure 1 we can have a glimpse of the performance of the algorithms for different message sizes, we notice two different behaviours: for small and medium message sizes the trend is more irregular, while for large message sizes there are less "jumps". It is interesting to note that the jumps don't occur at 12 or 24 processes as we may expect, which is the moment in which some processes are mapped to a new socket and to a new node respectively.

From these plots we can note that for the core mapping the default algorithm seems to work quite weel, expecially for large message sizes and many processes, while for small message sizes we can't see a big difference between the algorithms, expect for the chain, which performs well only for large message sizes: this can be explained by the fact that the chain algorithm leverages fragmentation to transfer the data with a pipeline, leveraging better the bandwith of the network.

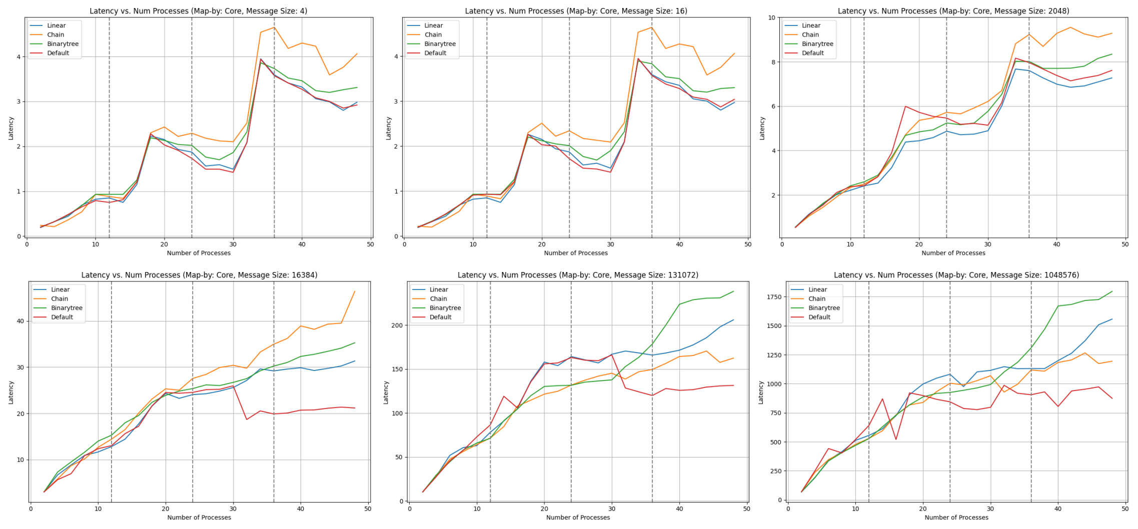

Figure 1: Latency against processes for a fixed message size, mapping by `core`.

3.2.2 Mapping by socket

Setting the `--map-by` flag of `mpirun` to `socket`, each process is mapped evenly to the sockets of the first node, this means that the total 48 processes will be distributed in this way: process 0 is bound to socket 0 of node 0, process 1 is bound to socket 1 of node 0, process 24 is bound to socket 0 of node 1, process 25 is bound to socket 1 of node 1.

In Figure 2 we see a similar trend as the core mapping, what is interesting here is that the default algorithm for $P = 48$ is always the best.

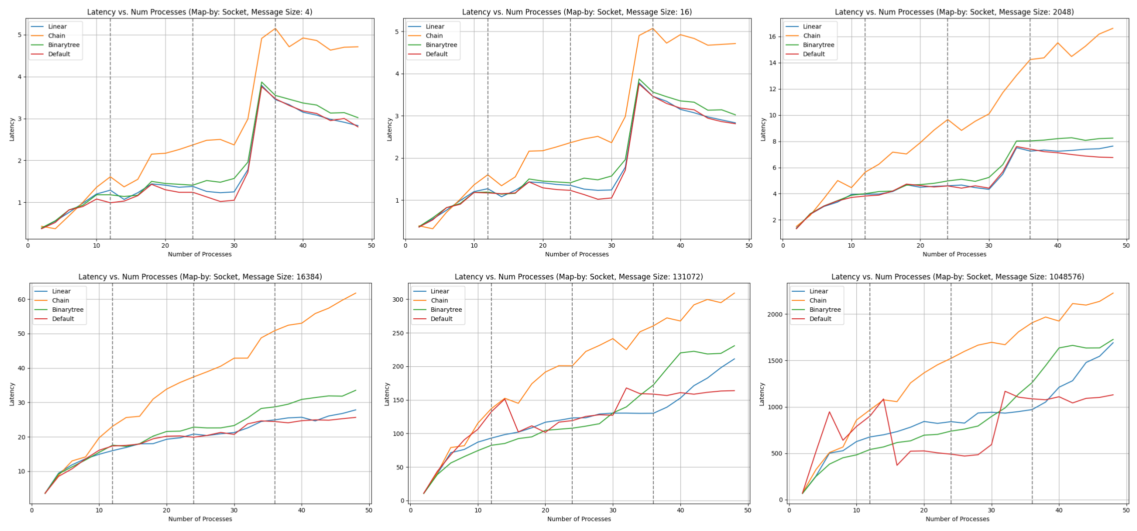
Figure 2: Latency against processes for a fixed message size, mapping by `socket`.

3.2.3 Mapping by node

Setting the `--map-by` flag of `mpirun` to `node`, each process is mapped evenly to the 2 nodes, this means that the total 48 processes will be distributed in this way: process 0 is bound to socket 0 of node 0, process 1 is bound to socket 0 of node 1, process 2 is bound to socket 1 of node 0, process 3 is bound to socket 1 of node 1.

From Figure 3 we can see that that the chain algorithm behaves badly, probably because of the pipeline in the communication: process 0 has to send a message to process 1, which is mapped to another node, increasing the latency; as the number of processes grows this problem is accentuated.

On the contrary, the other algorithms behave similarly, but the best is the linear, where half of the communications are on the same node and the other half to the other. Also the binary tree algorithm behaves similarly and it's interesting to note that the default algorithm for large message sizes chooses the wrong strategy.

It is nice to see that with this kind of allocation the binarytree and the linear algorithms are quite regular, with a linear grow of the latency.

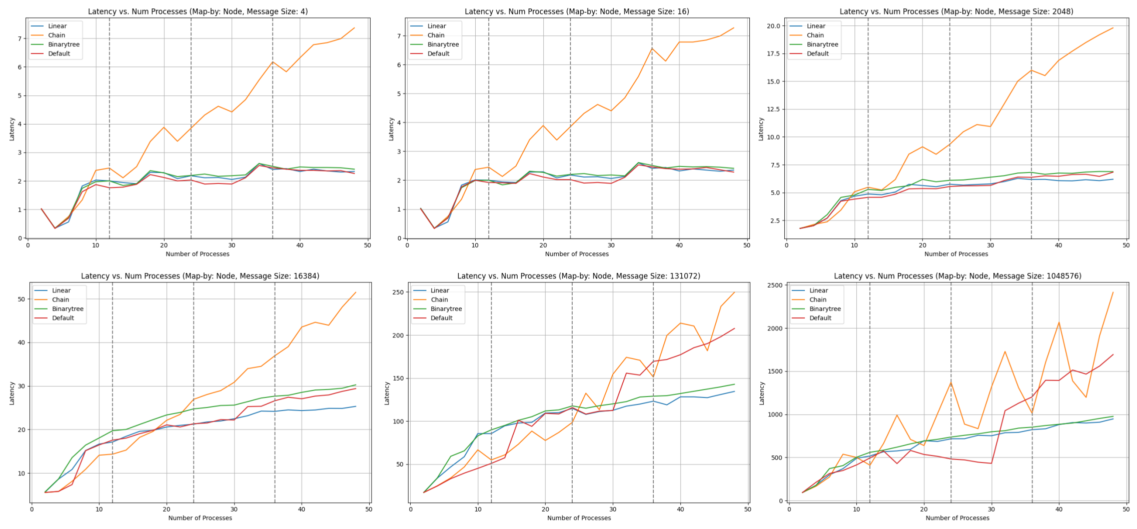
Figure 3: Latency against processes for a fixed message size, mapping by `node`.

3.3 Latency against number of processes, fixing the algorithm and the size

In figure 4 we can see a comparison of the different allocation strategies, for different message sizes.

- About the linear algorithm: we can see that it performs well with the node allocation where half of the messages is sent from the root to processes in the same node, while the other half is to processes in the other node.
- About the chain algorithm: the core allocation seems to be the best strategy, this is in line with the fact that the communication is performed in a pipeline with the order of the rank of the processes: if the processes are mapped to cores close to each other, the latency of the messages is reduced.
- About the binary tree algorithm: its behaviour is similar to the linear algorithm, with the allocation by node as the best allocation strategy and a remarkable difference in the latency for $P = 48$ and message size $= 1048576$.

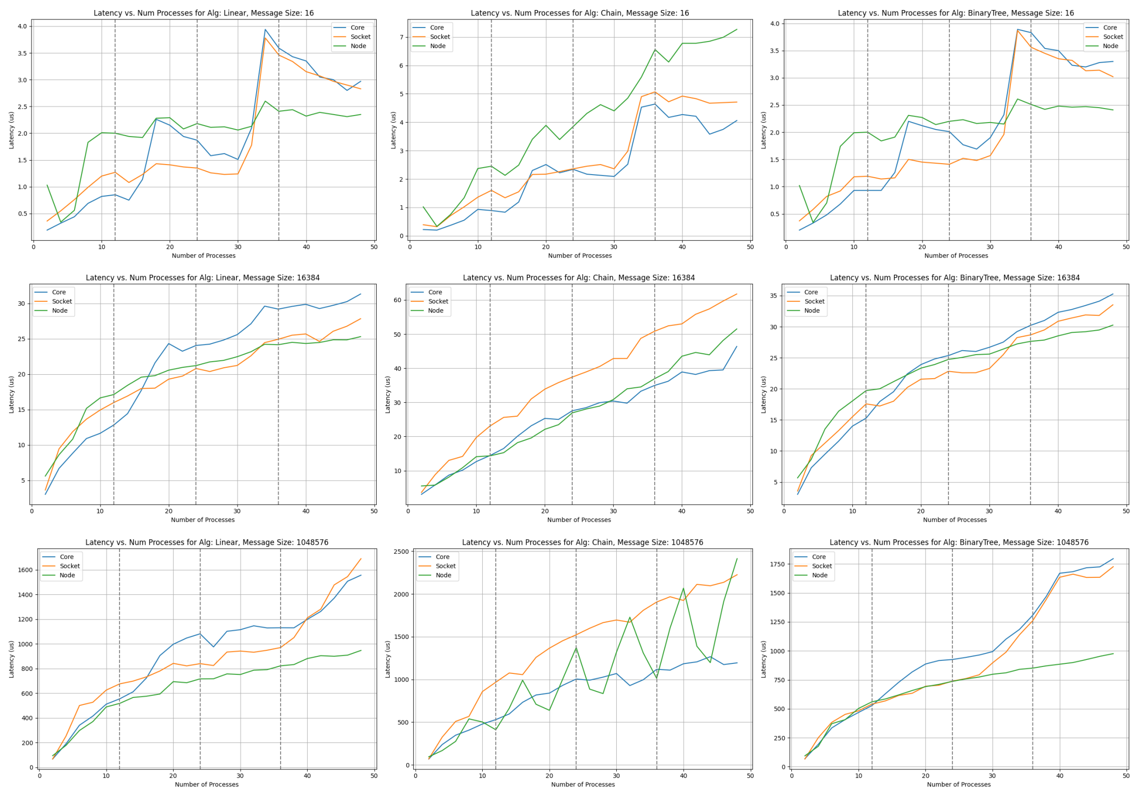

Figure 4: Latency vs Processes fixing the algorithm: in order Linear, Default, BinaryTree.

3.4 Default against other algorithms

To have a glimpse of the performance of the default algorithm, I grouped the processes and the message sizes in 3 groups (Small size $= [1, 2^7]$, Medium size $= [2^8, 2^{14}]$, Large Size $= [2^{15}, 2^{20}]$; Few processes $= [2, 16]$, Medium processes $= [18, 32]$, Many processes $= [34, 48]$) and then I computed the mean of the difference of the latency of the best algorithm (default excluded) and the default. In table 3.4 we can see the results I got.

The column "Optimal Coun" refers to the times the default algorithm resulted the best for a fixed allocation, size and number of processes.

The column "Mean Difference" is the mean of the difference between the best of the other algorithm and the default, grouping by the columns "Allocation", "Message Size" and "Processes". If the Mean Difference is positive, it means that in general the default algorithm is not the best algorithm.

We can see that for small and medium sizes of the messages the difference is not remarkable, but for large sizes of the messages we get some interesting insights about the performance of the default algorithm: for socket allocation, large message size and few processes; for node allocation, large message size and many processes; for core allocation, large message size and few or medium processes the default algorithm performs worse than the best of the other algorithms, on the contrary in all other cases the difference is not remarkable or the default is better.

It is difficult to infer on the reasons why this happens, but it's good to know the cases in which the default algorithm is not the best choice.

Allocation	Message Size	Processes	#Optimal	Optimal %	Mean Diff	Std Diff
core	Small	Few	10	17.86	0.07	0.08
core	Small	Medium	41	73.21	-0.05	0.07
core	Small	Many	30	53.57	0.09	0.61
core	Medium	Few	10	17.86	0.17	0.21
core	Medium	Medium	7	12.50	0.39	0.35
core	Medium	Many	26	46.43	0.77	3.92
core	Large	Few	16	28.57	17.28	55.45
core	Large	Medium	9	16.07	17.22	62.55
core	Large	Many	56	100.00	-65.71	83.83
socket	Small	Few	32	57.14	0.02	0.13
socket	Small	Medium	52	92.86	-0.11	0.09
socket	Small	Many	34	60.71	-0.01	0.05
socket	Medium	Few	19	33.93	0.15	0.36
socket	Medium	Medium	12	21.43	0.12	0.16
socket	Medium	Many	33	58.93	-0.30	0.80
socket	Large	Few	6	10.71	66.04	131.11
socket	Large	Medium	15	26.79	-7.08	98.31
socket	Large	Many	33	58.93	-33.76	125.58
node	Small	Few	28	50.00	0.04	0.18
node	Small	Medium	55	98.21	-0.15	0.09
node	Small	Many	31	55.36	0.00	0.06
node	Medium	Few	24	42.86	0.22	0.50
node	Medium	Medium	41	73.21	-0.03	0.23
node	Medium	Many	17	30.36	0.66	1.12
node	Large	Few	30	53.57	-2.62	28.18
node	Large	Medium	19	33.93	-12.58	89.62
node	Large	Many	0	0.00	146.32	192.01

Table 1: Difference of the default algorithm and the minimum latency of the orhers, fixing allocation, message size and number of processes.

4 Linear Models

The first approach I tried in order to infer the underlyign performance models of the various algorithms was to estimate the latency of between the cores of a node and of two contiguous THIN nodes with the program `osu_latency`. I tried to code a function that computes the total latency of the collective operation leveraging the measurements obtained, but I didn't obtain any good results.

Therefore I tried a different approach, fitting a plane to approximate the surface generated by the latency, varying the number of processes and the size of the message.

A different approach may be to perform a linear regression on a log transormation of the surface: the fit I obtained is not bad, with a medium $R^2 = 85\%$.

In figure 5 I show an example of the regression, while in table 4 a summary of the coefficients.

The plot in numbers:

Algorithm	Allocation	Intercept	Coeff. Log(size)	Coeff Processes	R^2
Binary Tree	core	-1.96	0.433	0.047	0.838%
Binary Tree	socket	-1.62	0.427	0.036	0.853%
Binary Tree	node	-1.092	0.405	0.023	0.844%
Linear	core	-2.02	0.44	0.045	0.831%
Linear	socket	-1.594	0.432	0.032	0.844%
Linear	node	-1.100	0.402	0.022	0.835%
Chain	core	-1.95	0.42	0.051	0.835%
Chain	socket	-1.518	0.44	0.045	0.870%
Chain	node	-1.09	0.370	0.047	0.838%

Table 2: Times the Default Algorithm is Not Optimal

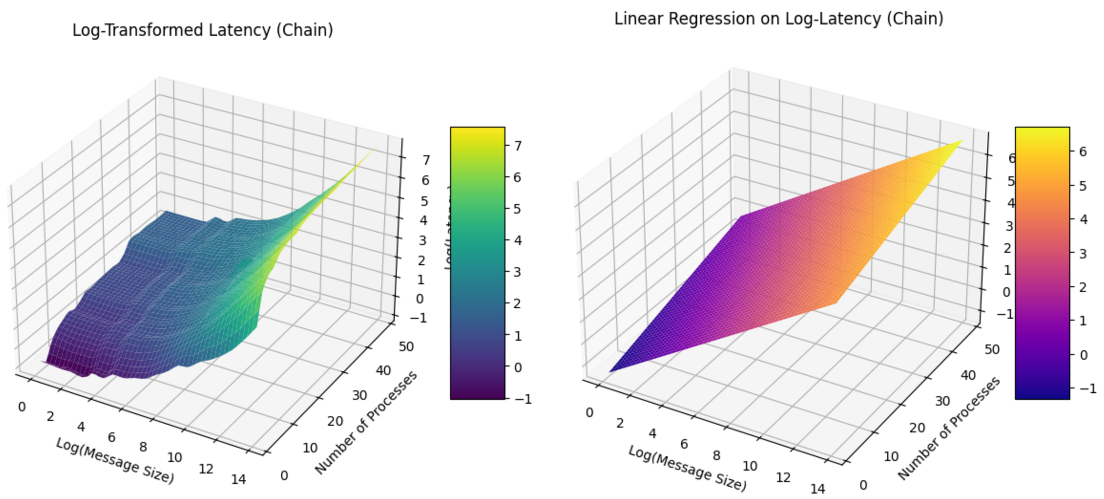

Figure 5: Log of the surface for socket allocation, Chain algorithm and linear fit of the surface

5 Barrier Algorithms

Barrier algorithms' primarily aim is the synchronization of the processes: this can be helpful in spotting some imbalances in the workload, moreover, an implicit barrier is present at the end of Gather, Scatter and Broadcast operations. It is clear that the size of the messages exchanged during a Barrier operation is zero, that is why we will only discuss the evolution of the latency with respect to the number of processes.

The algorithms I chose to compare with the Default are: the Linear, the RecursiveDoubling and the Tree algorithm.

A brief description of the algorithms, taken from [2], assuming P is the number of processes:

- Linear algorithm: all nodes report to a preselected root; once everyone has reported to the root, the root sends a releasing message to everyone. It requires P communication steps.
- Tree algorithm: as the name suggests, utilizes a hierarchical structure where processes synchronize in a tree-like fashion.
- Recursive Doubling algorithm: it requires $\log_2 P$ steps if P is power of 2, and $\lfloor \log_2 P \rfloor + 2$ steps if not. At step k, node r exchanges message with node (r XOR $2k$). If the number of nodes P is not a power of 2, we need two extra steps to handle extra nodes.

5.1 Latency against number of processes, fixing the allocation

The first analysis we perform is a comparison of the performance of the different algorithms, fixing the allocation of the processes.

From Figure 6 we can see many interesting aspects of the behaviour of the algorithms:

- The Linear algorithm is actually linear and mapping the processes by `core` we get an almost straight line.

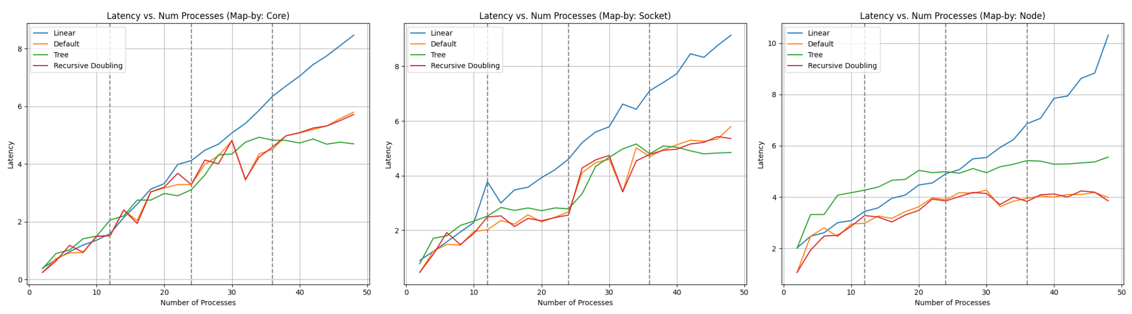

Figure 6: Latency vs Processes fixing the allocation of the processes.

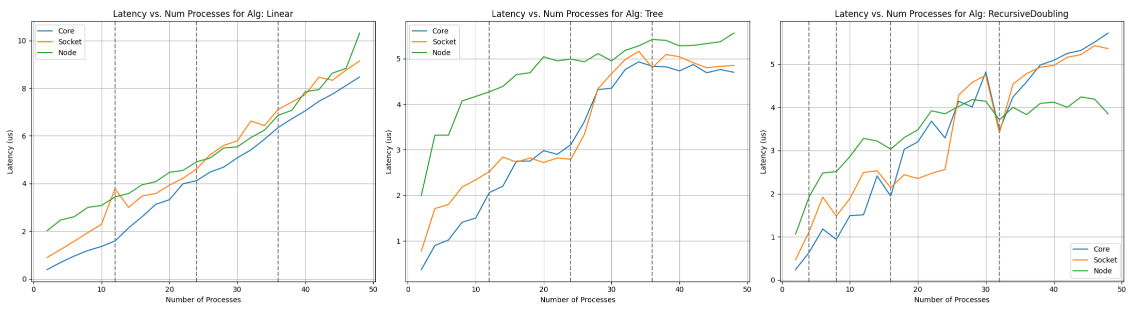

Figure 7: Latency vs Processes fixing the algorithm.

- In general, the Linear algorithm performs badly if the number of processes is high, with a remarkable difference with respect to the others when more than 24 processes are running (24 is the number of cores in a THIN node).
- It seems that the internal logic of MPI opts often for the Recursive Doubling algorithm, in fact they behave very similarly.

5.2 Latency against number of processes, fixing the algorithm

Another interesting analysis is the comparison of the same algorithm, varying the processes allocations (Figure 7).

- As far as the Linear algorithm, what we stated earlier still holds, noting that mapping by `core` is more efficient
- About the Tree algorithm, we note that, mapping by `socket`, we find two different behaviours: approximately constant in the intervals of [12-24] and [36-48] processes, approximately linear in the intervals of [2-12] and [24-36] processes. We remember that each socket in a THIN node has 12 cores. In any case it performs better if the mapping of the processes is set to `core` or `socket`.
- The Recursive Doubling shows an interesting behaviour: for $P = 8, 16, 32$ there is a local minimum in the latency. This is because when P is a power of 2 the algorithm requires only $\log_2 P$ communications steps and not $\lfloor \log 2P \rfloor + 2$. Although some irregulariteis, we can see that for the core and socket allocation the algorithm behaves quite linearly.

5.3 Default against other algorithms

As stated at the beginning of this report, the internal logic of OpenMPI doesn't always choose the optimal algorithm in terms of latency, the plot in Figure 8 demonstrates this fact.

To generate this plot we computed the difference between the latency of the default algorithm and the minimun latency of the algorithms analysed. If a point lays under the x-axis it means that the Default algorithm is optimal.

The plot in numbers:

We can see that in many cases the Default algorithm is not optimal, after all there is no absolute optimal algorithm, but we can see that if the allocation is set to `node`, the RecursiveDoubling

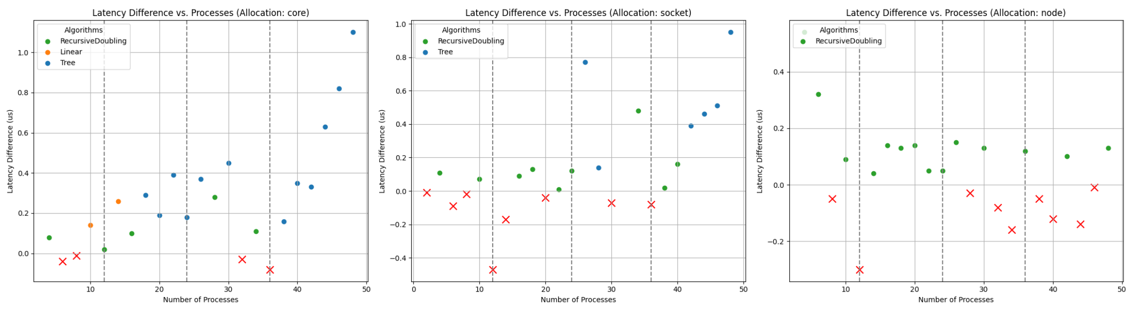

Figure 8: Difference of the Default algorithm latency and the min of the other algorithms latency.

Allocation	Default Optimal	Default not Optimal	Zero Difference	Not Optimal %
Core	4	19	1	79.17%
Socket	8	15	1	62.50%
Node	9	14	1	58.33%

Table 3: Times the Default Algorithm is Not Optimal

algorithm performs well. On the contrary if the allocation is set to `core`, the Tree algorithm seems to be the best choice. Finally, if the allocation is on the `socket`, the choice of the optimal algorithm is more challenging.

We also have to note that the difference between the latency of the default algorithm and the minimun latency of the algorithms analysed is not huge ($< \mu$s, maybe the Default algorithm shows some overhead for the decision of the best strategy to adopt), then probably the best choice is to stick to the Default algorithm and spend time and programming effort in optimizing the code; instead of choosing the most suitable Barrier algorithm for the specific case.

6 Linear Models

In this section we try to build some performance models of the algorithms. It is clear that we won't analyse the Default algorithm since it would require the knowledge and understanding of the internal MPI logic.

I decided to regress the Latency in two different ways: for the Linear algorithm with a simple linear regression, for the Tree algorithm with splines of degree 1, placing the knots to $[12, 24, 36]$ and for the Recursive Doubling with a logarithmic and linear regression. The coefficients of the splines of Figure 6 are reported in table 4.

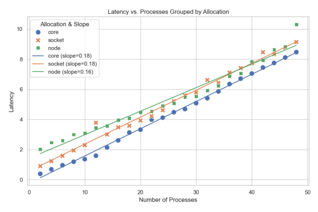

Figure 9: Linear Algorithm.

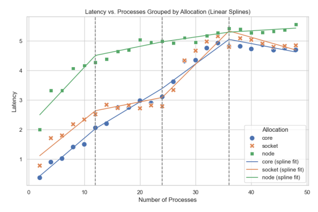

Figure 10: Tree Algorithm.

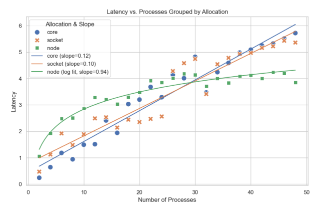

Figure 11: Recursive Doubling.

Allocation Type	Slopes Between Knots
Core	[0.1603, 0.1136, 0.1383, -0.0355]
Socket	[0.1523, 0.0373, 0.1877, -0.0537]
Node	[0.2010, 0.0384, 0.0274, 0.0114]

Table 4: Slopes Between Knots for Different Allocations

7 Appendix

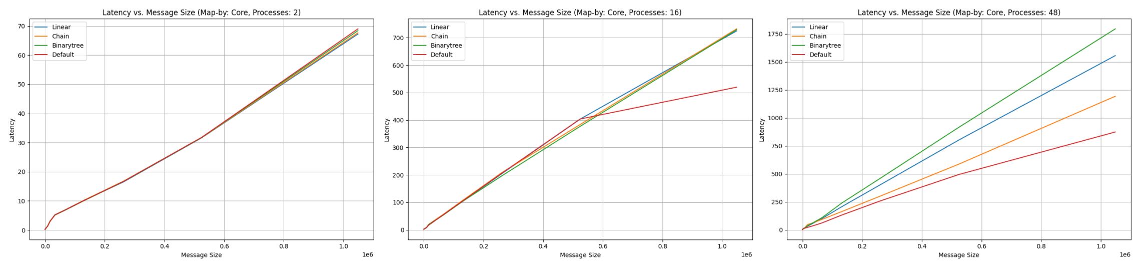

Figure 12: Latency vs Processes fixing the allocation of the processes.

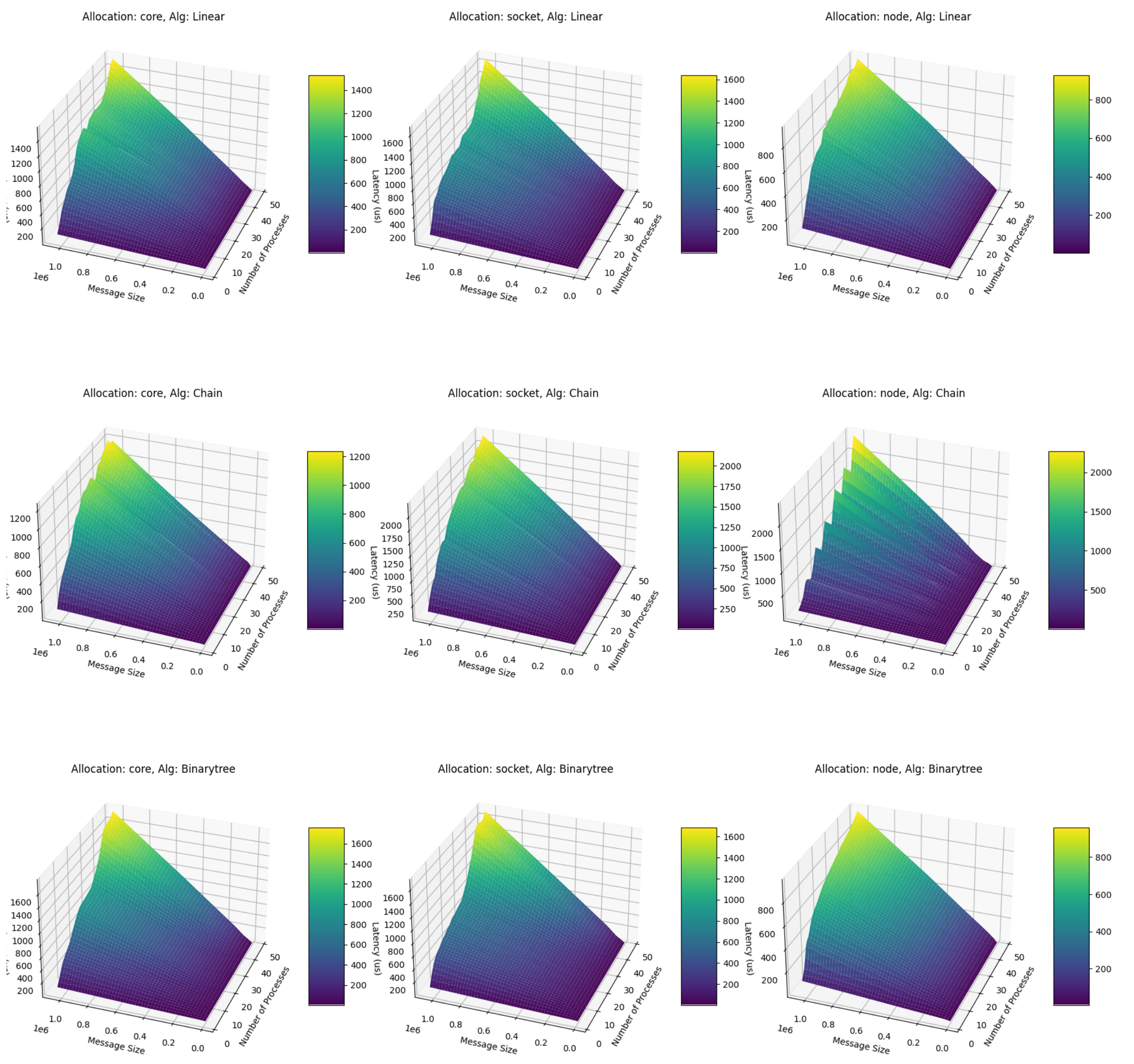

Figure 13: 3D plot: on the x-axis the message size, on the y-axis the number of processes and on the z-axis the latency

References

[1] Area Science Park. Orfeo cluster documentation. https://orfeo-doc.areasciencepark.it/HPC/computational-resources/, 2023. Accessed: 2024-09-23.

[2] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Edgar Gabriel, and Jack Dongarra. Performance analysis of mpi collective operations. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 10–16. IEEE, 2005.

[3] R. Rabenseifner. Automatic profiling of mpi applications with hardware performance counters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 35–42. Springer, Berlin, Heidelberg, 1999.