High Performance Computing
Exercise 1

Luis Fernando Palacios Flores [*]

University of Trieste

1 Introduction

Collective operations are very important in parallel applications and require efficient and effective algorithms to ensure optimal performance, for instance in MPI applications. However, no single algorithm has proven to be superior in all possible scenarios [1, 2].

The implementation of openMPI algorithms for collective operations involves various parameters that impact performance. Different algorithms may be optimal for different collective operations based on factors like the physical network topology, number of processes, and message sizes [1, 2]. The openMPI library's default mechanism considers these factors and selects an algorithm to execute the task at hand [1].

There are various modeling approaches for estimating the execution time of collective operations. Because of its variety of parameters, it is a complex system in itself. In particular, Hockney models for a collective operation with P processes [1–4] can take the form:

$$T(P, m, \alpha, \beta) = f(P, m) \times g(m, \alpha, \beta) \tag{1}$$

The first term of the RHS might not necessarily involve a message of size m and can depend on the number of processes in different forms, for instance linearly or with the algorithm. The second term includes the latency α and the inverse of the bandwidth β of point-to-point operations which are assumed to be constant for all message sizes and number of processes [1–4]. This is a strong assumption due to the many possible interactions between the factors involved in collective operations.

The aim of this project is to compare different openMPI algorithms designed for the blocking collective operations broadcast and barrier, with the objective of gaining insights that could aid in modeling them. This comparison could be beneficial for selecting algorithms in MPI applications.

2 Methodology

The OSU Micro-Benchmarks library [5] allows for the performance of collective latency tests for various MPI blocking collective operations. Therefore, in this report, latency is the metric used to compare the performance of algorithms that implement the collective operations.

The OSU programs for conducting the latency tests were run on the ORFEO cluster utilizing all cores in two EPYC nodes (epyc002 and epyc004) and, for comparison, two THIN nodes

[*]luisfernando.palaciosflores@studenti.units.it

(thin001 and thin002). The `openMPI/4.1.5/gnu/12.2.1` library was chosen to execute the programs. Different numbers of processes were spawned for each operation to gain insights into the overall performance of the various algorithms. Additionally, the broadcast operation was tested with different message sizes ($2^0 - 2^{20}$ bytes) of the `MPI_CHAR` data type. To minimize noise and capture the variability in measurements, each procedure was repeated 10 times for broadcast and 20 times for barrier. Furthermore, to bind each MPI process to a single physical CPU core and ensure efficient execution, the mapping policy `--map-by core` was enabled to prevent process migration between cores.

The algorithms selected for each collective operation were (algorithm number for each operation in the OSU library displayed in parenthesis):

- broadcast: basic linear (1), chain (2), binary tree (2)
- barrier: linear (1), double ring (2), Bruck (4)

For benchmarking the operations, default algorithms (0) were used. These algorithms are designed to dynamically choose the most suitable algorithm based on the message size and the number of processes [1].

3 Results and discussion

Figure[1] 1 shows the results of the broadcast operation for the THIN partition. The latency tends to increase as the size of the message increases, not necessarily in a linear fashion as commented in the introduction. Similar results were observed for the EPYC nodes, and this is more evident in Figure 2. In the first row of Figure 1, it can be seen that the execution time of the operation is constrained by the message size, as expected according to the Hockney models. The latency can be higher for 2 bytes of an `MPI_CHAR` than for 64 bytes.

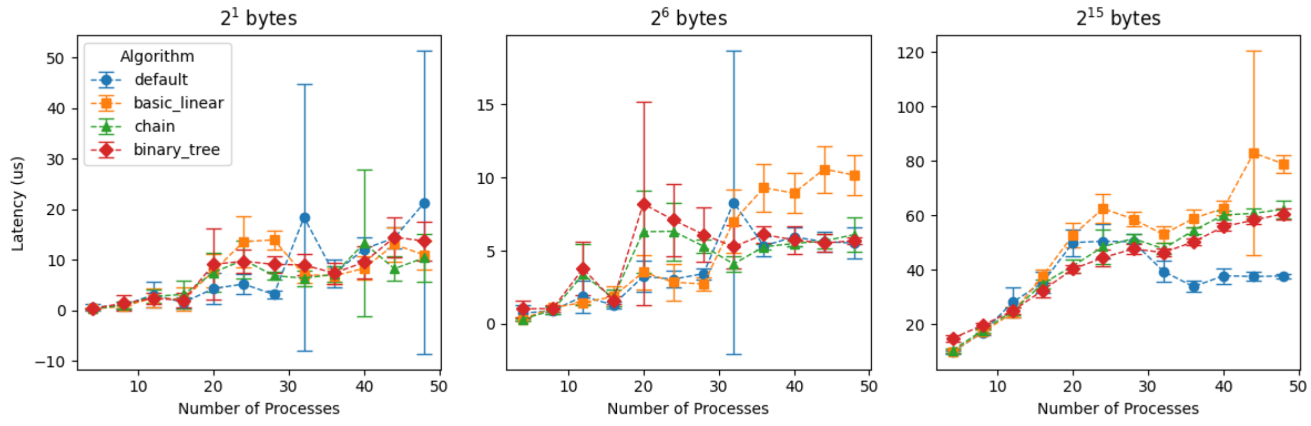

Figure 1: Latency of the `MPI_Bcast` operation vs the number of processes, for two THIN nodes, for the different algorithms utilized. From left to right: message size of 2, 64, and 32768 bytes of the `MPI_CHAR` data type.

[1]In all the plots the dashed lines were included only for visualization and to help understand the relationship between the variables.

For a specific number of processes, the variability in the latency increases dramatically regardless of the algorithm used. For instance, in Figure 1 the default algorithm has a large variability for small message sizes at 32 processes. In this case, the latency at the end of the interval indicated by the standard deviation makes no sense because of negative values. This was observed for all algorithms in both partitions and was more frequent for processes that exceeded the number of cores in one node. It suggests that the performance of the algorithms can be influenced by factors related to the computer architecture as commented in the introduction.

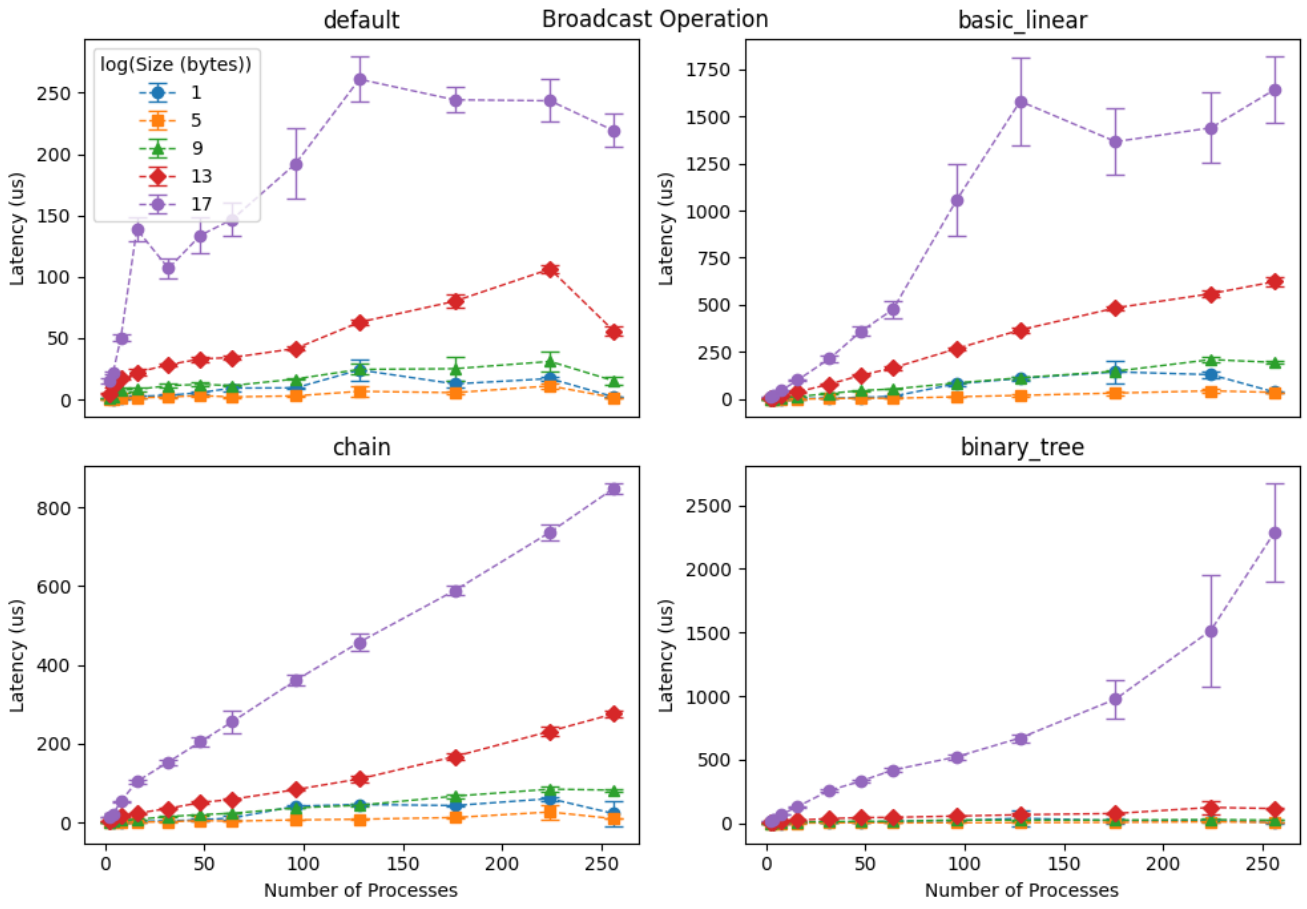

Figure 2: Latency of the `MPI_Bcast` operation vs the number of processes, for two EPYC nodes, for the different algorithms utilized. From top to bottom and from left to right: default, basic linear, chain, and binary tree algorithms.

Figure 1 suggests that, for the THIN nodes, all algorithms exhibit similar performance across the number of processes for each message size. This observation is further supported by Figure 3, which confirms consistent latency averages across different message sizes. The plot indicates that for tasks requiring only one THIN node, the binary tree algorithm might be preferable, while the default algorithm could be more suitable for applications involving a large number of processes. Averaging the latency across the number of processes, instead of message size, yielded a similar trend in the algorithms' performance. Overall, the default algorithm outperformed the basic linear, binary tree, and chain algorithms in terms of average latency, aligning with the findings anticipated from Figure 3.

In contrast, for the EPYC nodes, the performance differences are more pronounced, as depicted in Figures 2 and 3. The default algorithm consistently performed the best across all scenarios, supporting the notion that this method dynamically selects the optimal algorithm based on collective operation conditions. The binary tree algorithm showed superiority over the chain algorithm for small to medium-sized messages but performed worse for larger messages ($> 2^{14}$ bytes of `MPI_CHAR`). The ranking of the algorithms based on average latency was as follows: default, chain, binary tree, and basic linear, with the latter consistently performing the worst.

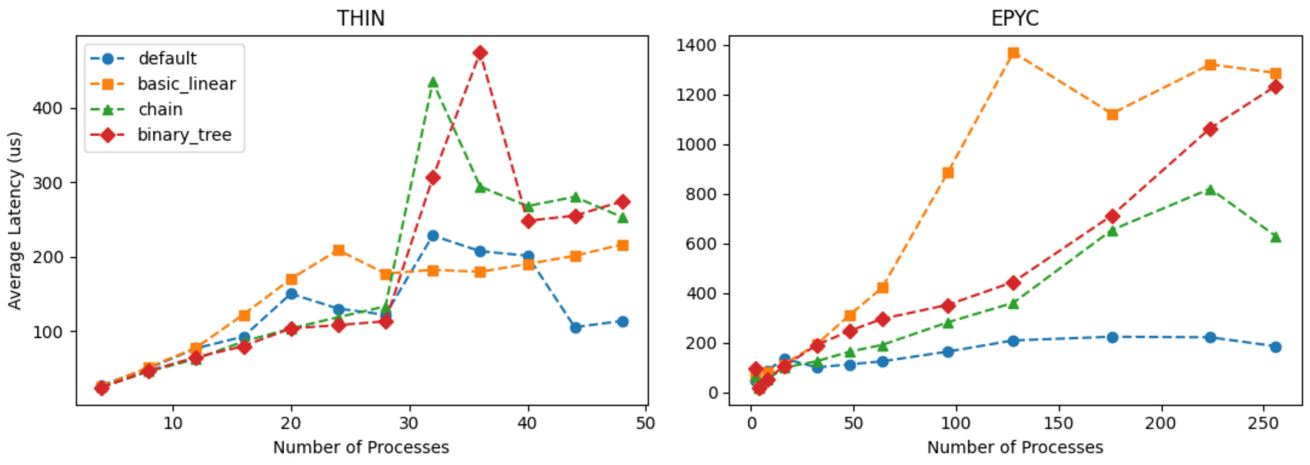

Figure 3: Average Latency of the `MPI_Bcast` operation vs the number of processes over different message sizes of the `MPI_CHAR` data type, for the different algorithms utilized. From left to right: THIN nodes and EPYC nodes.

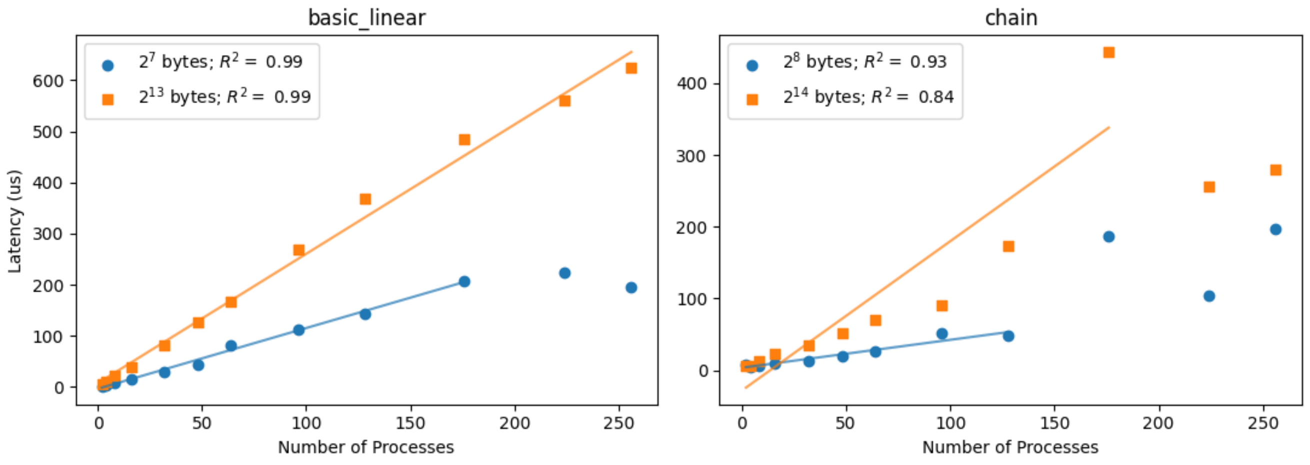

Figure 4: Linear regression of the latency of the `MPI_Bcast` operation (on the EPYC nodes) as a function of the number of processes for different message sizes of the `MPI_CHAR` data type. From left to right: basic linear and chain algorithms.

The comparison between the two partitions suggests that the execution of collective operations may necessitate a thoughtful selection of algorithms for applications running on a specific number of nodes, depending on the partition. Additionally, it seems that latency is influenced when processes are spread across different nodes or even different sockets, potentially indicating a sequential allocation of cores, filling one socket at a time and then one node at a time. The observed high variability for a certain number of processes could potentially be an artifact of the methodology used, warranting additional analysis.

Figure 2 indicates a possible linear relationship between the latency and the number of processes[2] for all message sizes for the basic linear and chain algorithms consistent with descriptions in the literature [1, 3, 4]. After thorough analysis, it was found that the latency is well described by a linear model using almost all processes for small messages ($\leq 2^{14}$ bytes) for the basic linear algorithm and large messages ($\geq 2^{15}$ bytes) for the chain algorithm (see Figure 2). For other message sizes, a satisfactory description is achievable only up to a certain number of processes, as depicted in Figure 4.

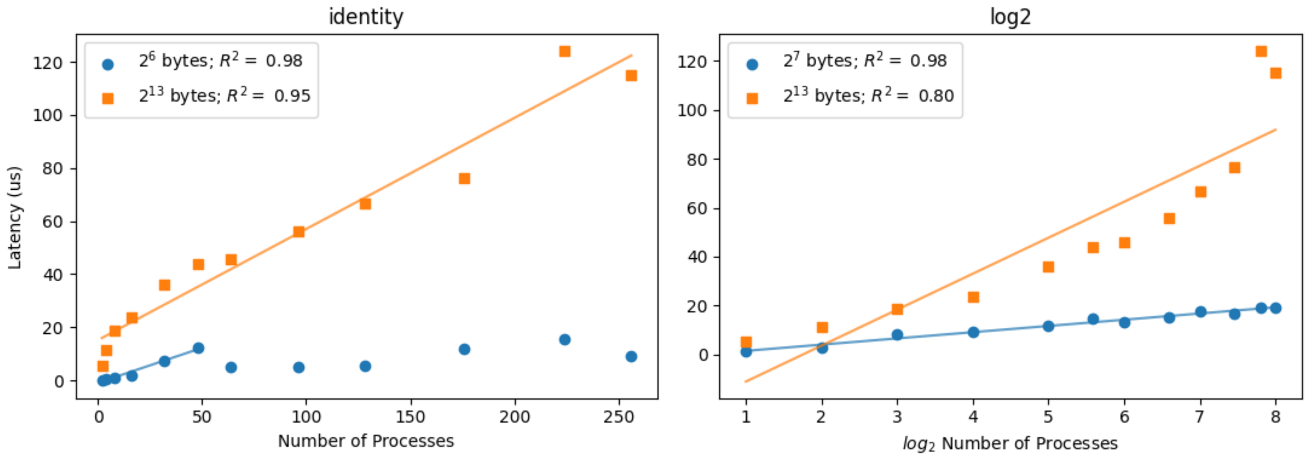

Figure 5: Linear regression of the latency of the `MPI_Bcast` operation (on the EPYC nodes) with the number of processes for different message sizes of the `MPI_CHAR` data type. From left to right: linear and logarithmic dependency in the number of processes.

For the binary tree algorithm, the literature [1, 3, 4] describes the relationship between the latency and number of processes as logarithmic. However, a brief analysis of Figure 2 casts doubt on this high-level model, which makes no trivial assumptions. Linear regressions of latency as a function of the number of processes and its logarithm were conducted. In most cases, the logarithmic model best fits the data. Nevertheless, for large message sizes ($> 2^{13}$ bytes) the linear relationship outperforms the logarithmic one. As before, the models struggle to accurately describe latency across all numbers of processes but are effective up to a certain point (refer to Figure 5 left) for all message sizes. For large messages, the logarithmic relationship provides a good description for nearly all processes, while the linear relationship performs well across all processes, as illustrated in Figure 5.

For the THIN nodes, similar observations were made regarding the binary tree algorithm. The only difference is that both the linear and logarithmic relationships describe well the data for all

[2]Here the "latency" provided by the OSU library programs is considered to be the execution time or at least to contribute to it, in accordance with Equation(1), as I interpreted it from the project instructions.

number of processes, with the latter performing better. Surprisingly, the basic linear algorithm is well characterized with a linear and also a logarithmic relation, in contrast to the observations for the EPYC nodes. The latency for large messages ($> 2^{13}$ bytes) was better represented by the logarithm of the processes rather than linearly. In the case of the chain algorithm, a logarithmic relationship between the latency and the number of processes provided a better explanation of the data than a linear one. Complex interactions between different factors involved in collective operations may account for this behavior. A more sophisticated approach, such as the one presented by Nuriyev et al.[1], may be necessary to develop a more accurate model. However, this level of analysis is beyond the scope of the project and the content covered in this HPC course.

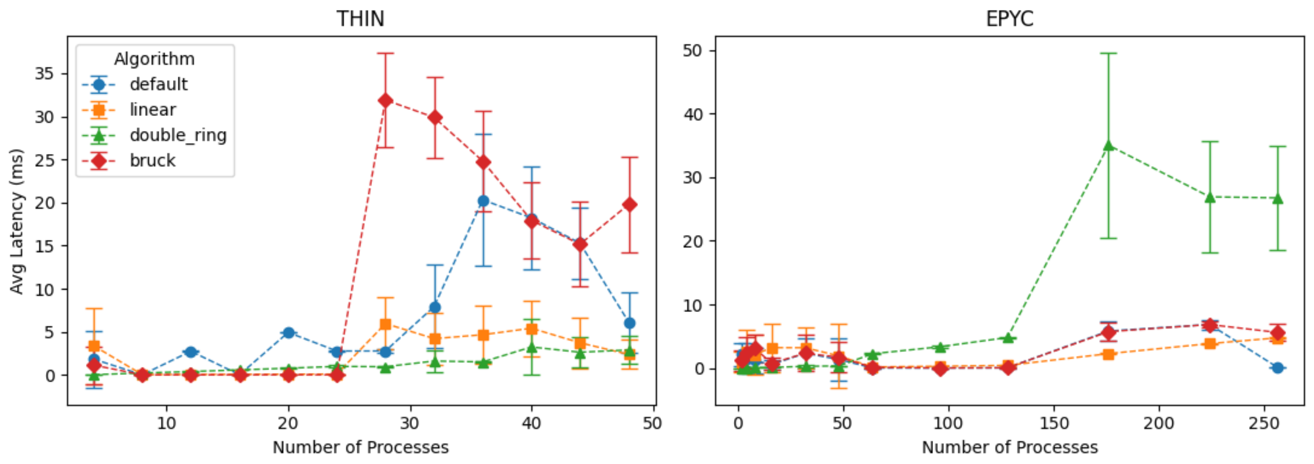

Figure 6: Latency of the `MPI_Barrier` operation vs the number of processes, for the different algorithms utilized. From left to right: THIN nodes and EPYC nodes.

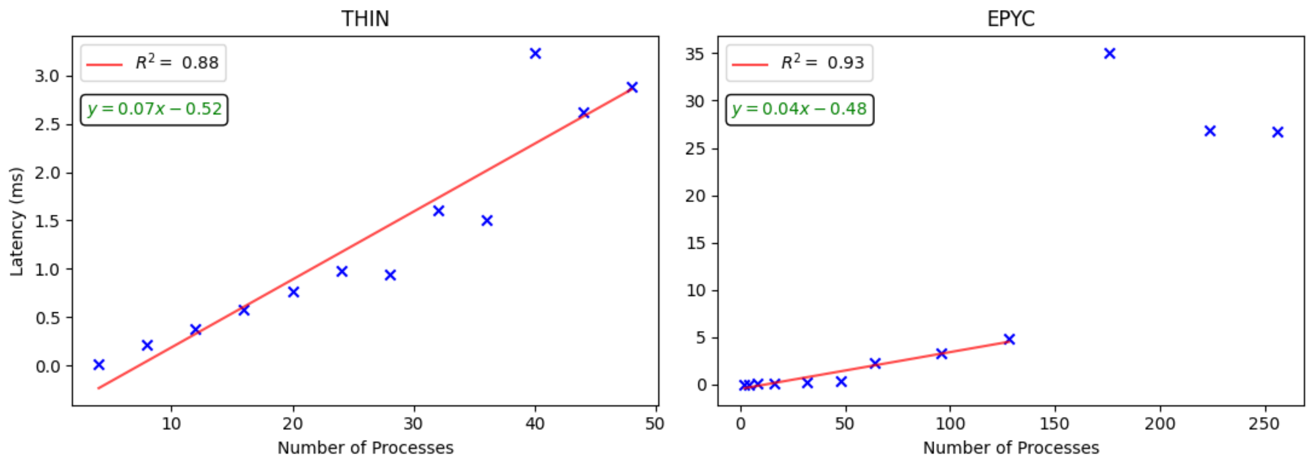

Figure 7: Linear regression of the latency of the `MPI_Barrier` operation as a function of the number of processes, for the double-ring algorithm. From left to right: THIN and EPYC nodes.

In Figure 6, the relationship between latency and the number of processes for the barrier operation in both THIN and EPYC nodes is not as clear as for the broadcast operation. It is noteworthy that in the THIN nodes, the Bruck algorithm performed well when the operation was executed in only one node but significantly increased its latency for two nodes. A similar pattern can be seen for the double-ring algorithm in EPYC nodes.

In this operation, only the double-ring algorithm is accurately described by the models found in the literature [3]. Figure 7 indicates that the latency for the THIN nodes allows a representation of all the processes involved in the collective operation, whereas, for the EPYC nodes, only the processes in one node were well represented by literature models, at least for the number of processes selected as part of the methodology. The latency of point-to-point communication [3] estimated from the linear regression is 35.12 ± 4.14 μs for the THIN nodes and 19.43 ± 2.03 μs for just one EPYC node, if the cores are indeed allocated sequentially. The intercept of these regressions is something not considered in the models as presented in Equation (1), and its interpretation is beyond the scope of this project. However, the fact that it is negative might well indicate that it doesn't correspond with reality.

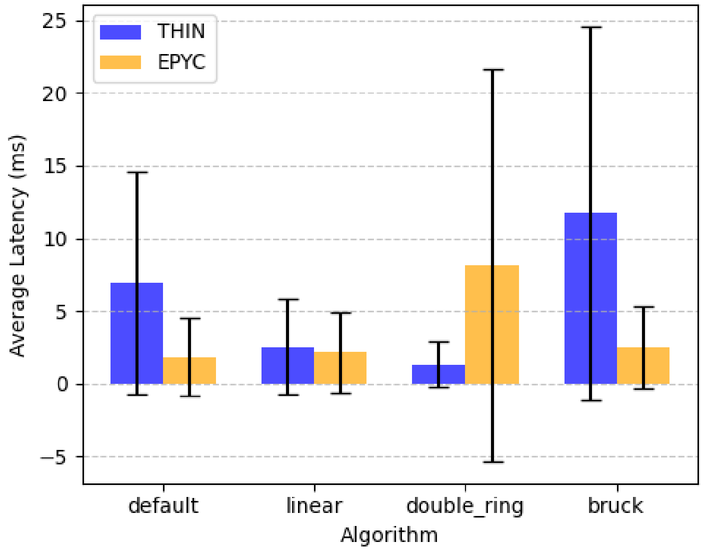

Figure 8: Average Latency of the `MPI_Barrier` operation vs the number of processes on THIN and EPYC nodes, for the different algorithms utilized.

Overall, for the barrier operation in the THIN nodes, the double-ring algorithm performs best across all the number of processes with an average latency of 1.31 ± 1.59 ms. It is followed by the linear, default, and bruck algorithms, as shown in Figure 8. Interestingly, the default algorithm does not perform the best on average in the barrier operation compared to the broadcast operation. Most algorithms perform better in the EPYC nodes than in the THIN nodes. In the EPYC nodes, the default algorithm performs slightly better than the linear one with an average latency of 1.83 ± 2.70 ms. In contrast to the THIN partition, the double-ring algorithm is the worst-performing one in these nodes.

4 Conclusions

The comparison of the algorithms revealed that there is no single algorithm that consistently performs best in all scenarios, as stated in the introduction. Overall, for the broadcast operation, the default algorithm delivered the best performance on both THIN and EPYC nodes. Notably, the EPYC nodes exhibited a significant performance difference among the algorithms for a large number of tasks, while the THIN nodes showed similar average latency across all algorithms. The basic linear and chain algorithms were well approximated by a linear relationship between latency and the number of processes in both node types. Interestingly, contrary to models found in the literature, a logarithmic relationship provided an equally or even more accurate model for these algorithms in the THIN nodes. For the binary tree algorithm, a logarithmic relationship best fit the data in both node types, aligning with findings in the literature. However, for large message sizes, a linear model proved to be more accurate.

In the barrier operation, the double-ring algorithm outperformed the default algorithm in the THIN nodes, while in the EPYC nodes, the default algorithm was slightly better than the linear one. Notably, only the double-ring algorithm was accurately described by the model from the literature, fully for the THIN nodes and partially for the EPYC nodes. Unlike the broadcast operation, a clear model for the barrier operation was not evident from the data.

In conclusion, the comparison between the two cluster partitions highlighted the need to carefully select algorithms based on the application, cluster partitions, and other parameters beyond this report's scope. For instance, a significant increase in latency variability was observed for a specific number of processes regardless of the algorithm used, potentially suggesting that factors related to computer architecture may influence algorithm performance.

References

[1] Emin Nuriyev, Juan-Antonio Rico-Gallego, and Alexey Lastovetsky. Model-based selection of optimal mpi broadcast algorithms for multi-core clusters. *Journal of Parallel and Distributed Computing*, 165:1–16, 2022.

[2] Udayanga Wickramasinghe and Andrew Lumsdaine. A survey of methods for collective communication optimization and tuning. *CoRR*, abs/1611.06334, 2016.

[3] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, June 2007.

[4] Emin Nuriyev and Alexey Lastovetsky. Efficient and accurate selection of optimal collective communication algorithms using analytical performance modeling. *IEEE Access*, 9:109355–109373, 2021.

[5] Mvapich :: Benchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/, 2024. Accessed: 2024-02-18.