High Performance Computing final project
Exercise 1

Giulio Fantuzzi [SM3800012]

21th February 2024

1 Problem statement

This project aims to assess the performance of various `openMPI` [1] algorithms for specific collective operations, specifically broadcast and barrier. My objective is to estimate the latency of the default implementation and compare it with values obtained through the selection of different algorithms.

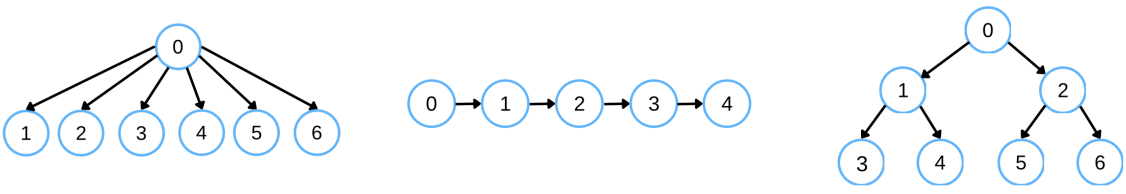
2 Methodology

To estimate the latency of the operations, I referred to the `OSU` benchmark[2], running its scripts on two *thin* [1] nodes of the `ORFEO` [3] cluster. To enhance the efficiency of the analysis, the entire data gathering process was automated by using some bash scripts, which were then submitted to the cluster using the `SLURM` workload manager, utilizing the `sbatch` command for streamlined execution. The analyzed problem presented several degrees of freedom, so I got different measures of the latency by varying the number of processes, their allocation and the size of the messages exchanged. I concluded my analysis by attempting to construct a performance model and to extrapolate results basing on the specific architecture on which the operations were executed. For more information about the scripts, check the repository https://github.com/giuliofantuzzi/HPC_FinalProject.

3 Broadcast

3.1 Broadcast algorithms

In addition to evaluating the default `openMPI` algorithm for broadcast operations, this study incorporates a comprehensive analysis of three alternative algorithms: linear/flat tree (Figure 1(a)), chain (Figure 1(b)), and binary tree(Figure 1(c)).



(a) Linear algorithm (b) Chain algorithm (c) Binary Tree algorithm

Figure 1: Broadcast algorithms scheme

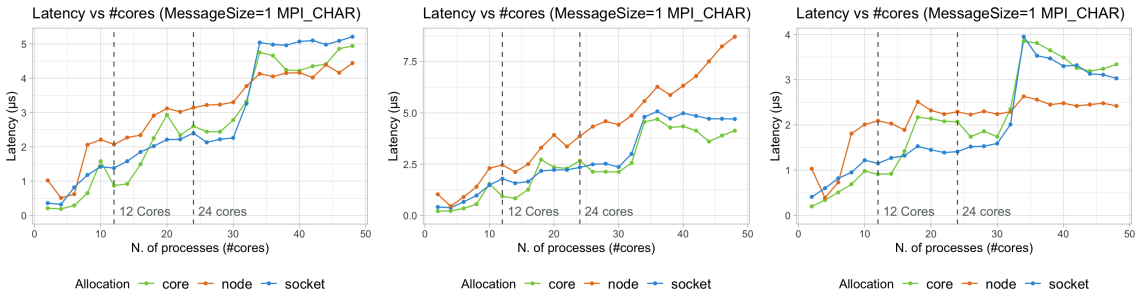
[1] Opting for *thin* nodes in the measurements was driven by the intense activity and prolonged resource allocation times observed on *Epyc* nodes during the data collection phase. Nevertheless, performing experiments on *Epyc* nodes would probably result in similar interpretations, though some adjustments may be needed due to differences in the nodes' structure.

3.2 Broadcast latency analyses

The data collection phase generated a sizable dataset, providing latency details for different combinations of algorithm, number of processes, processes allocation and message size. The extensive nature of the dataset introduced a multitude of degrees of freedom, offering flexibility for analysis. To handle this complexity, I initially performed analyses by constraining specific degrees of freedom and emphasizing marginal effects. Subsequently, a thorough examination was carried out to explore the interactions between variables in various ways.

3.2.1 Fixing algorithm and compare processes allocation effects

For each algorithm, I assessed the influence of process allocation on the operation latency. To maintain the independence of my analysis from message size, I fixed it to 1 `MPI_CHAR`, effectively isolating and examining a form of 'pure' latency associated with the algorithms.



(a) Linear algorithm (b) Chain algorithm (c) Binary Tree algorithm

Figure 2: Analyzed broadcast algorithms

Referring to Figure 2(a), the linear algorithm's behavior aligns with expectations. Both socket and core allocations show latency "jumps" when changing node, while node allocation progresses seamlessly without noticeable jumps. Similar observations apply to socket changes: core allocation exhibits a jump, whereas socket allocation remains unaffected. An intriguing aspect arises when examining latency jumps. In the *thin* architecture, with sockets of 12 cores and nodes of 24 cores, one might expect jumps at 12 and 24 processes. Surprisingly, jumps occur at 16 and 32 processes, hinting at additional factors influencing communication beyond straightforward node/socket changes.

Transitioning to the chain algorithm (Figure 2(b)), there are parallels to the previous considerations. A notable distinction arises when changing node: for the linear case, when the number of processes surpasses 24, the three distinct allocations appear to converge. This convergence aligns with the nature of the linear algorithm, where messages are sent from process 0 to all the others. In particular, when processes span both nodes, the execution steps remain consistent across allocations. For the chain algorithm, on the other hand, the communication among processes follows a precise order, leading to a distinct behavior. When processes are assigned by node, the greater distance traveled by messages becomes evident if compared to allocations by core or by socket.

Ultimately, the y-axis in Figure 2(c) underscores slower latency values associated with the Binary Tree broadcast, showcasing its (expected) superior performance compared with the two algorithms analyzed above. Allocating processes by node reveals a stable latency profile after 24 cores (node change), indicative of optimized intra-node communication upon reaching a specific depth within the tree structure. When allocating by socket or by core, instead, an apparent jump followed by a decrease in latency is observed. This might be attributed to the hierarchical structure of the algorithm: the jump may correspond to the transition between different levels of the tree, introducing a slight delay in communication. Once passed this initial transition, the latency decreases as the algorithm progresses through subsequent levels of the tree.

3.2.2 Fixing processes allocation and compare algorithms

I fixed the processes allocation by configuring it to `--map-by core` and I assessed the algorithmic impact on the operation latency. As detailed in Section 3.2.1, I ensured the consistency of my analysis by fixing the message size to 1 `MPI_CHAR`. My initial focus was on comparing linear and chain algorithms, as they provided more insightful points of comparison due to their inherent characteristics.

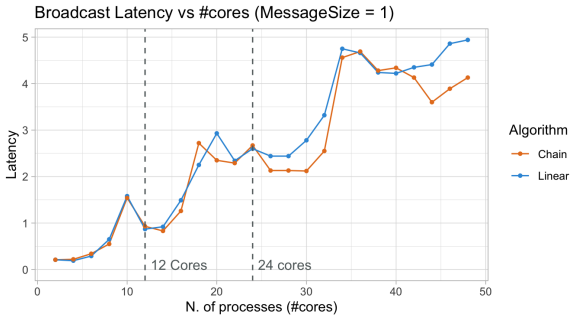


Figure 3: Linear vs Chain comparison

Figure 3 offers some insights into how the algorithm choice influences latency across 3 regions:

- **Region 1 (within socket)**
  Both the linear and chain algorithms exhibit nearly identical performance. This suggests that intra-socket communication, occurring between cores within the same socket, is so fast that the algorithmic implementation has a minimal (if not irrelevant) impact.

- **Region 2 (within node)**
  As we move to communication within the entire node, a subtle difference emerges, indicating that the choice between chain and linear algorithms might have a more nuanced impact. Despite the slight advantage of the chain algorithm becomes a bit more apparent, the two algorithms still perform quite similarly.

- **Region 3 (outside node)**
  In the external node communication region, where latency is inherently higher, the contrast between chain and linear algorithms becomes more pronounced. This aligns with expectations, as the chain algorithm's utilization of contiguous cores contrasts with the linear algorithm's approach of sending from rank 0 to all the other ranks.

Conducting similar plots while systematically varying the message size would yield more insightful results, but we will delve into details in section 3.2.3. As for now, let's keep the message size to 1 `MPI_CHAR` and include the default algorithm and the binary tree in the analysis.

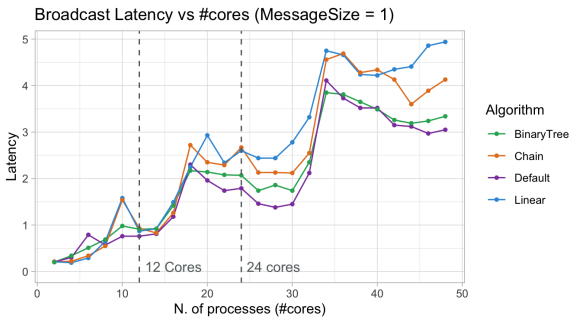


Figure 4: Broadcast algorithms comparison

As observed in earlier sections, the binary tree algorithm consistently exhibits lower latency values. From an algorithmic standpoint, this aligns with expectations, as a tree structure efficiently distributes

the communication among its branches, enhancing the overall performance. Ultimately, Figure 4 shows how the default configuration outperforms the other algorithms, emerging as the one with the lowest latency. This was something I also expected: default configurations are often tuned to strike a balance between different use cases, making them robust and efficient in various environments. Additionally, the default settings may leverage platform-specific optimizations, taking advantage of system characteristics to enhance communication efficiency.

3.2.3 Fixing processes allocation and compare algorithms for different message sizes

As suggested in Section 3.2.2 and showed in Figure 5, increasing the message size may provide a clearer distinction between the algorithms, thus enhancing our understanding of their effectiveness.

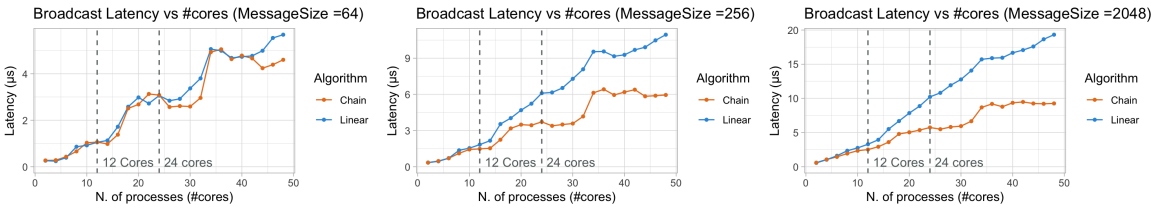


Figure 5: Linear vs Chain comparison varying message size (`--map-by core` allocation)

Chain algorithm exhibits superior performance, primarily attributable to its utilization of contiguous cores. Initially, when the message size was fixed at 1 `MPI_CHAR`, the performances of the linear and chain algorithms were nearly indistinguishable. However, as we introduce variations in the message size, the advantages of the chain algorithm become more evident. A critical aspect contributing to its effectiveness is the algorithm's capacity to partition messages into chunks during transmission, a feature absent in the linear broadcast approach (see [4]). As a consequence, with an increasing message size, the latency of the linear algorithm experiences sharp growth, accentuating the widening performance disparity with respect to the chain algorithm. The direct correlation between message size and communication latency remains consistent for both the binary tree and default algorithms. As shown in Figure 6, both the binary tree and default configurations continue to outperform linear and chain algorithms, underscoring their robustness across the increasing message size.

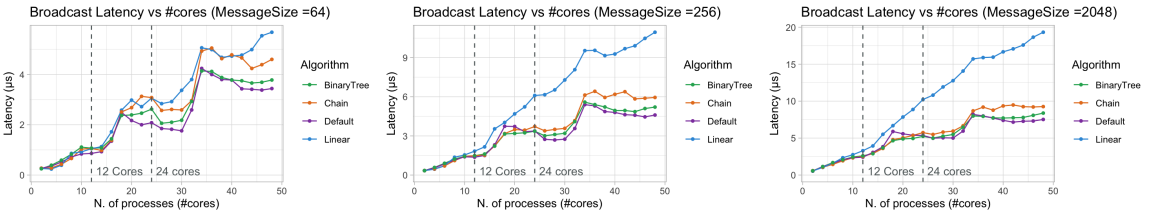


Figure 6: Broadcast algorithms comparison varying message size (`--map-by core` allocation)

3.3 Broadcast performance models

As a conclusive step of my latency analysis for broadcast communication, I attempted to develop a simple performance model to better understand the intricate dynamics at play. My initial approach involved estimating both latency and bandwidth within point-to-point communication routines, employing the `OSU` benchmark again (utilizing the `osu_latency` tool at the path `/osu-micro-benchmarks-7.3/c/mpi/pt2pt/standard.`). The intention was to leverage these estimated values to construct a model resembling the Hockney model I found in literature (see [5], [6]). Unfortunately, the results I obtained did not align well with the collected data, prompting me to explore alternative methodologies. Guided by my statistical background, I moved towards the implementation of a linear model.

In selecting predictors for the model, I considered variables such as message size and the number of processes, while I chose to maintain a fixed process allocation using `--map-by core`. Additionally, I opted to exclude the intercept from the model: although the latency of a 0-sized message shared among 0 processes is undefined, I assumed it to be 0 for the sake of convention. Furthermore, I constructed separate models for each algorithm, aligning with established practices in the literature where performance models are frequently algorithm-specific [2]. Although I derived individual models for each algorithm, it is important to note that they share a common structure [3] , with formula:

$$log_2(\text{Latency}) = \beta_1 \cdot \text{Number of Processes} + \beta_2 \cdot log_2(\text{Message Size}) \quad (1)$$

All the models were built using the software `R`. Table 1 reports, for each of the analyzed algorithms, the estimates for β_1, β_2 and the R^2_{adj} as a metric to assess the goodness of fit of the associated model. Despite all the coefficients being statistically significant and the R^2_{adj} being fairly large for all the models, Figure 7 shows how the association between the (log2 of) message size and the latency is not linear. eq. (1) was hence updated by introducing a quadratic term for log_2(Message Size), obtaining:

$$log_2(\text{Latency}) = \beta_1 \cdot \text{Number of Processes} + \beta_2 \cdot log_2(\text{Message Size}) + \beta_3 \cdot log_2(\text{Message Size})^2 \quad (2)$$

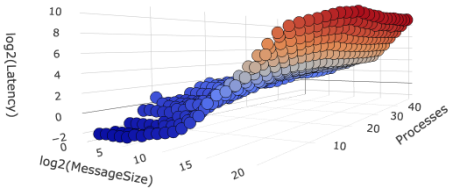


Figure 7: log_2(Latency) w.r.t log_2(Message Size) and Processes

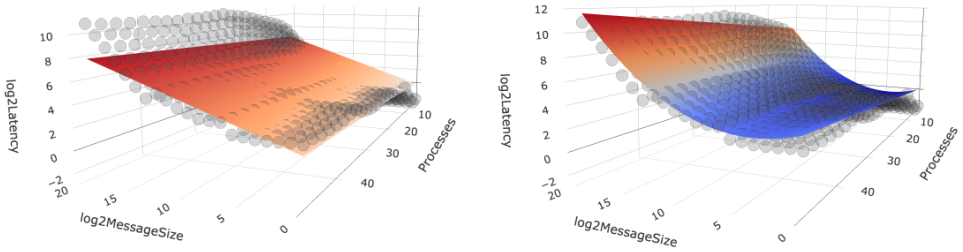
The improvement was notable, and it is illustrated in Table 2 and in Figures 8(a) and 8(b).

Algorithm	β_1	β_2	R^2_{adj}
Linear	0.029815	0.321259	88,19 %
Chain	0.021957	0.306713	86,77 %
Binary Tree	0.016315	0.317449	86,58 %
Default	0.009586	0.321418	85,86 %

Table 1: Broadcast models w/o quadratic term

Algorithm	β_1	β_2	β_3	R^2_{adj}
Linear	0.0718655	-0.3022613	0.0355722	97,57 %
Chain	0.0646477	-0.3262914	0.0361133	98,00 %
Binary Tree	0.0600791	-0.3314744	0.0370215	98,38 %
Default	0.0525293	-0.3153369	0.0363273	97,74 %

Table 2: Broadcast models w/ quadratic term



(a) Model w/o quadratic term (linear broadcast) (b) Model w/ quadratic term (linear broadcast)

Figure 8: Improvement of the model's fit when introducing quadratic term

[2]Alternatively, I could treat *Algorithm* as a categorical regressor, with the foresight of incorporating it into an interaction term rather than as a standalone factor. However, it would introduce unnecessary complexity to the model in terms of interpretability and explainability, and I prioritized maintaining clarity and simplicity in the model design.

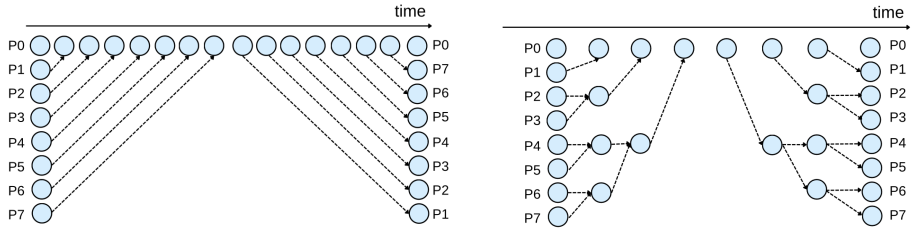
[3]the log_2 trasformation was applied to both Latency and Message Size in order to mitigate their skewness.

4 Barrier

The analytical approach employed for Barrier closely follows the methodology applied in section 3 for Broadcast, ensuring consistency and allowing us to focus more directly on the results. As the Barrier primarily aims for synchronization without the involvement of varying message sizes, the current analysis inherently introduces fewer degrees of freedom than the previous one.

4.1 Barrier algorithms

My analysis includes the default MPI configuration, along with three additional algorithms: the linear algorithm, the Bruck algorithm, and the tree algorithm. In a linear barrier (Figure 9(a)) all the processes report to a pre-selected root; once every process has reported to the root, the root sends a releasing message to all participants. Bruck algorithm, instead, requires $log_2(P)$ communication steps: at step k, process r receives a zero-byte message and sends it to process $(r - 2^k)$ and $(r + 2^k)$, with *wrap around*. [4] Tree algorithm (Figure 9(b)), as the name suggests, utilizes a hierarchical structure where processes synchronize in a tree-like fashion.



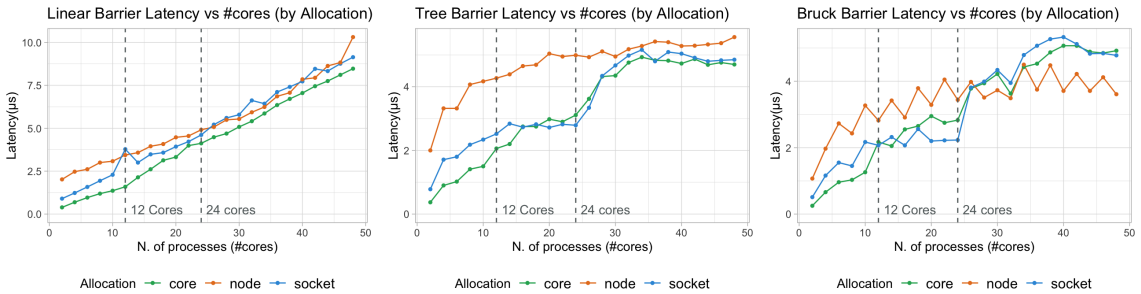
(a) Linear algorithm (b) Tree algorithm

Figure 9: Scheme for linear and tree algorithm

4.2 Barrier latency analyses

4.2.1 Fixing algorithm and compare processes allocation effects

For each of the algorithms, we assessed the influence of process allocation on the operation latency:



(a) Linear algorithm (b) Tree algorithm (c) Bruck algorithm

Figure 10: Analyzed barrier algorithms

The linear barrier algorithm exhibits a distinct behavior compared to its broadcast counterpart. Unlike broadcast, the absence of jumps in latency can be attributed to the nature of the barrier operation itself, which involves messages of size 0. As highlighted in Section 3.2, when processes span both nodes, the execution steps of the linear algorithm remain consistent across the different

[4] Behavior where a value, when reaching its max or min limit, wraps back around to the opposite end of the range.

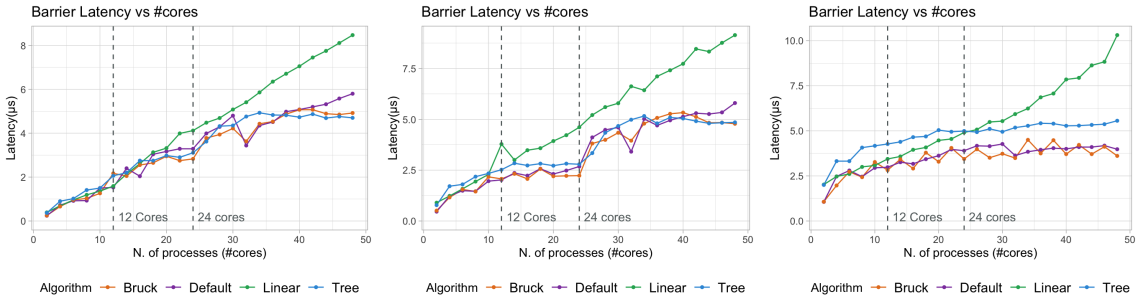
allocations. Figure 10(a) further exemplifies this behavior: the lines in the plot converge as processes fill available cores, showcasing the independence of execution steps from the allocation type.

Similar to the broadcast, the tree barrier algorithm demonstrates lower latency compared to the linear algorithm, aligning with expectations from an algorithmic perspective. Consistency between allocations by core and by socket was expected: the communication between sockets on the same node has a minimal impact, especially when dealing with zero-sized messages. However, allocation by node introduces higher average latency, suggesting potential overhead associated with the inter-node coordination. Figure 10(b) effectively illustrates this behavior, particularly when focusing on the plot region corresponding to processes lower than 24 (when the number of processes is lower than the available cores in a node, allocation by node is obviously the least efficient among the alternatives).

The Bruck barrier algorithm, like the tree, exhibits lower latency compared to the linear algorithm. Similarity in latency between allocations by core and by socket is consistent, emphasizing -once again- the reduced impact of inter-socket communication for zero-sized messages. An intriguing observation is the 'zig-zag' shape in the latency curve when allocating by node (Figure 10(c)). Determining definitive motivations is challenging, even though this peculiar pattern might derive from the wrap-around behavior inherent in the Bruck algorithm. Nevertheless, this remains a personal speculative consideration, and further investigation would be necessary to gain a deeper understanding.

4.2.2 Fixing processes allocation and compare algorithms

In this section, we focus on fixing the processes allocation and evaluate the algorithmic impact on operation latency. Unlike the approach in Section 3.2.2, here I delve into the algorithmic effects for each configuration of the `--map-by` option. For brevity, I will present the plots, as interpretations would be repetitive and largely align with my previous observations. In particular:



(a) Configuration by core (b) Configuration by socket (c) Configuration by node

Figure 11: Analyzed barrier algorithms varying allocation

A clear distinction becomes apparent when comparing the linear algorithm to the others. Although variations in performance among the non-linear algorithms are less pronounced, the default configuration exhibits remarkable similarity to the Bruck algorithm. To summarize, both default and Bruck stand out as the most performant barrier algorithms in my assessment.

4.3 Barrier performance models

The final step of the barrier analysis involved developing a performance model for the latency. As detailed in Section 3.3, the chosen approach was to construct algorithm-specific models, in accordance with established practices in the literature. For the linear algorithm, I successfully integrated the estimated point-to-point latency (from the `OSU`) into a Hockney model. More precisely, with $\hat{\alpha}$ being the estimate for the point-to-point latency and P the number of processes involved, the model is:

$$\text{Latency} = \hat{\alpha}(P-1) = \hat{\alpha}P - \hat{\alpha} \tag{3}$$

The resulting analysis, presented in Figure 12, was notably convincing:

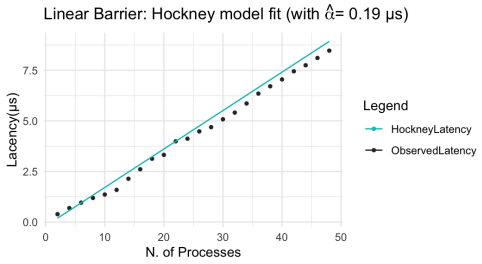


Figure 12: Hockney model for linear barrier

However, attempts to extend the Hockney model to other barrier algorithms were unsuccessful, so I reverted to a statistical approach, implementing a simple linear model. In the current regression analysis the sole predictor is the number of processes (intercept not included), and the log2 transformation to the response variable is not required anymore: the latency distribution, previously skewed by message size differences in broadcast, is now normalized (barriers are built on 0-sized messages). For non-linear algorithms, an additional enhancement was achieved by introducing a quadratic term.

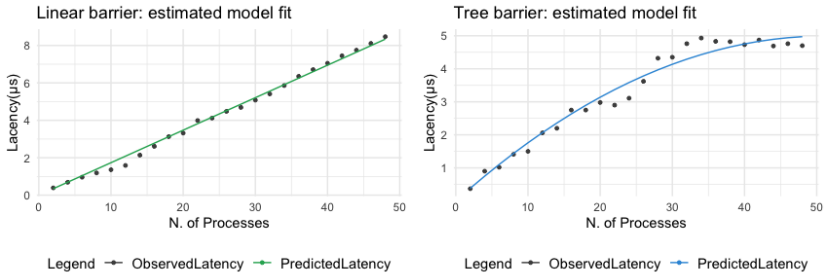
$$\textbf{Linear algorithm:} \quad \text{Latency} = \beta_1 \cdot \text{Number of Processes} \tag{4}$$

$$\textbf{Non-linear algorithms:} \quad \text{Latency} = \beta_1 \cdot \text{Number of Processes} + \beta_2 \cdot (\text{Number of Processes})^2 \tag{5}$$

Table 3 reports, for each of the analyzed algorithms, the estimates for β_1, β_2 (for the non-linear algorithms only) and the R^2_{adj} as a metric to assess the goodness of fit of the associated model.

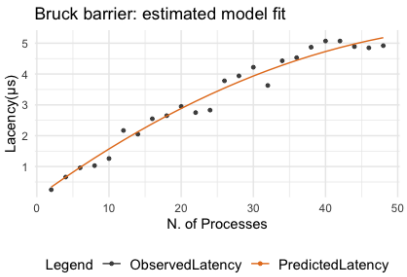
Algorithm	$\boldsymbol{\beta_1}$	$\boldsymbol{\beta_2}$	$\boldsymbol{R^2_{adj}}$
Linear	0.1736878	/	99,86 %
Tree	0.1950677	-0.0019060	99,46 %
Bruck	0.1700697	-0.0012954	99,45 %
Default	0.1690773	-0.0010560	99,26 %

Table 3: Barrier latency models summary

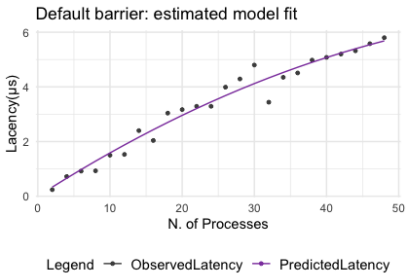


(a) Linear algorithm

(b) Tree algorithm



(c) Bruck algorithm



(d) Default algorithm

References

[1] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[2] K. S. Hemmert, D. S. Milroy, R. M. Senger, and M. C. Miller. OSU micro-benchmarks 7.3. Technical report, The Ohio State University, 2022.

[3] areasciencepark. Orfeo-doc. https://orfeo-doc.areasciencepark.it.

[4] 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.

[5] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994.

[6] Jelena Pjesivac-Grbovic, George Bosilca, Graham Fagg, Edgar Gabriel, and Jack Dongarra. Performance analysis of mpi collective operations. volume 2005, 01 2005.