High-Performance Computing – Exercise 1

Comparative Analysis of OpenMPI Algorithms for Collective Operations using OSU Benchmark

Tanja Derin [SM3800013]
Data Science and Artificial Intelligence [346SM]
Supervised professor: Stefano Cozzini
July 2025

Abstract

This report presents the results of Exercise 1 of the High Performance Computing course, aimed at benchmarking and modeling the performance of MPI collective operations using the OSU Micro-Benchmarks on the ORFEO cluster. The mandatory `MPI_Bcast` operation and the selected `MPI_Reduce` operation were evaluated under several algorithmic variants available in the OpenMPI tuned component. The benchmarks were conducted on two full EPYC nodes, with latency measurements collected across a range of message sizes and process counts. Additional point-to-point latency tests were used to model inter-core communication based on architectural distances. Results show algorithmic differences in scaling behavior, with measured latencies compared to simple analytical models that reflect the topology-aware communication cost.

1 Introduction

Efficient communication between processes is essential for high-performance parallel applications. The Message Passing Interface (MPI) standard defines a wide range of collective operations, such as broadcast, reduce, scatter, and gather, to coordinate data movement among multiple processes. These collectives are widely used in scientific computing and large-scale distributed systems. The OpenMPI implementation provides multiple algorithms for each collective operation, including linear, binomial tree, pipeline, chain, and more. The choice of algorithm affects performance depending on the number of processes, message size, and system topology. OpenMPI allows users to select specific algorithms at runtime using environment variables and MCA parameters.

This project evaluates the performance of different OpenMPI algorithms for two collective operations: `MPI_Bcast` (mandatory) and `MPI_Reduce` (selected).

To complement the measurements, simple analytical models were developed using point-to-point latency data from the `osu_latency` benchmark. These models supported the interpretation of performance trends in relation to architectural boundaries such as CCX, CCD, NUMA domains, and inter-node communication.

The following sections outline the methodology, describe the algorithms evaluated, compare empirical results with model predictions, and conclude with observations on algorithm selection.

System Architecture

All benchmarks were performed on the EPYC partition of the ORFEO high performance computing cluster. This partition consists of 8 compute nodes, each equipped with:

- 2 × AMD EPYC 7H12 CPUs (Rome architecture);
- Total: 128 cores and 512 GiB DDR4 memory per node;
- 4 NUMA nodes per CPU (8 NUMA domains total);
- 100 Gb/s interconnect between nodes.

Each AMD EPYC 7H12 processor contains 8 Core Complex Dies (CCDs), each comprising 2 Core Complexes (CCXs) with 4 cores and 16 MB of L3 cache. These CCDs communicate through an internal I/O die, while sockets are interconnected via AMD's Infinity Fabric (3 links at 16 GT/s, with a theoretical bandwidth of 96 GB/s). Figure 1 illustrates the structure of the Rome processor.

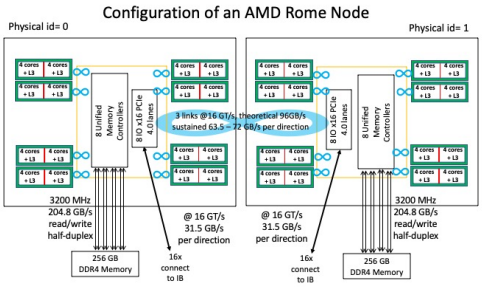

Figure 1: Block diagram of the AMD EPYC Rome architecture.

2 Benchmarking Methodology

All benchmarks were submitted using SLURM, explicitly requesting 2 full nodes using the `--exclusive` flag to ensure no resource contention. The environment was configured as follows:

- MPI library: OpenMPI 4.1.6
- Benchmark suite: OSU Micro-Benchmarks 7.4
- SLURM flags: `--nodes=2 --ntasks-per-node=128 --partition=EPYC`
- Placement: `--map-by core` for consistent process affinity

For each collective operation, a loop over powers of two message sizes from 2 bytes to 1 MiB and process counts from 2 to 256 was performed. Each test was repeated with different algorithm IDs using OpenMPI's `coll_tuned` module.
To ensure stable and accurate latency measurements, each test was executed with 100 warm-up iterations (-x 100) followed by 1000 measured iterations (-i 1000). The warm-up phase allows the system to stabilize by mitigating initialization overhead, while the measured phase provides reliable average latency values.
Results were parsed directly from stdout using `awk`, and written to a CSV file for plotting and model validation.
All tests were conducted using the `--map-by core` strategy to reduce variability introduced by NUMA boundaries or inter-node communication. Although alternative mappings across sockets or nodes could reveal additional insights, the goal was to isolate and evaluate the inherent performance of each algorithm under controlled, low-latency conditions.

3 Collective Benchmarks Results

For each operation `MPI_Bcast` (broadcast) and `MPI_Reduce`, the default OpenMPI implementation and three additional algorithms from the `coll_tuned` module were tested. Benchmarks were executed under identical conditions, systematically varying both the number of processes and the message sizes.

3.1 Broadcast Algorithms

The broadcast algorithms tested were: Pipeline, Binomial Tree, and Chain.

Each algorithm demonstrates distinct communication characteristics that affect performance depending on message size and process count. The following two plots compare these algorithms from two perspectives: first, how latency varies with the number of processes for fixed message sizes, and second, how latency scales with message size across different process counts.

Broadcast Latency vs Number of Processes

Figure 2 presents the latency scaling of five broadcast algorithms across increasing process counts and three different message sizes. Each subplot uses a log-log scale to capture exponential growth trends and reveal distinctions among algorithms.

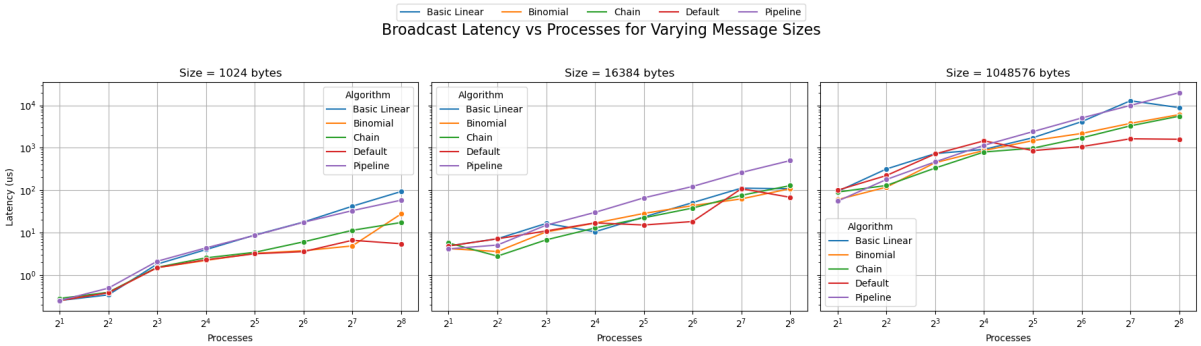

Figure 2: Broadcast latency vs number of processes for three message sizes: 1024 bytes, 16384 bytes and 1 MB. Log-log scale is used to highlight scaling behavior.

For larger message sizes, latency increases for all algorithms, but the Default implementation consistently exhibits the lowest latency across all process counts. The Chain and Binomial algorithms show similar performance, with moderate scalability. In contrast, the Basic linear and Pipeline approaches suffer from higher latency, particularly as the number of processes grows, due to their limited ability to overlap communication and computation.

Broadcast Latency vs Message Size

Figure 3 illustrates the broadcast latency as a function of message size (in bytes) across various process counts, ranging from 4 to 256. Each subplot corresponds to a fixed number of processes.

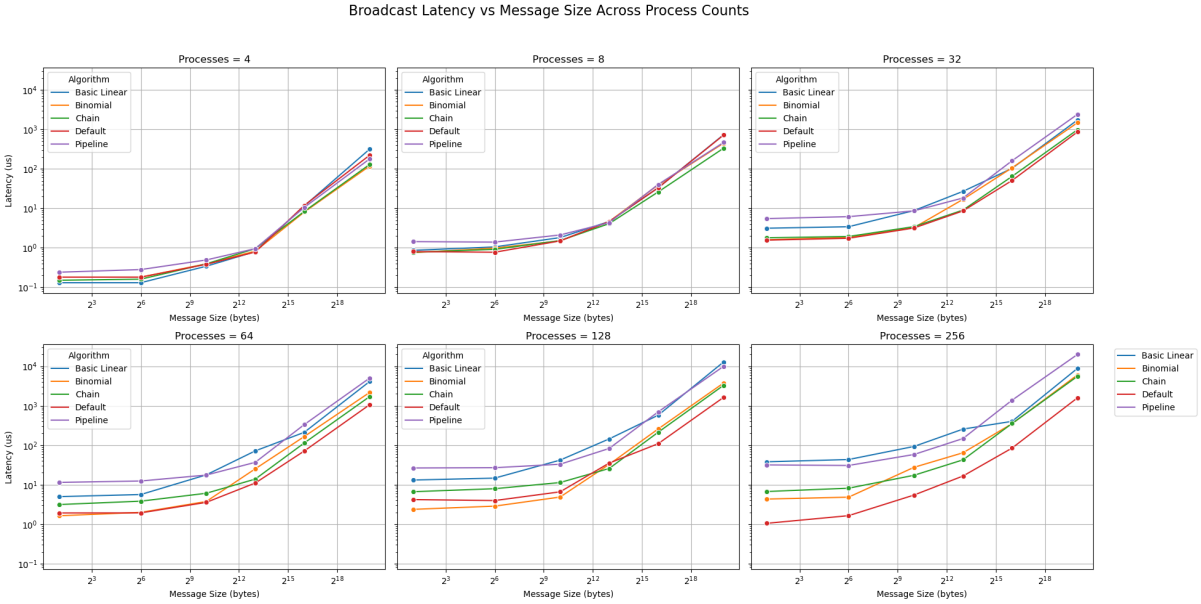

Figure 3: Broadcast latency versus message size for various algorithms across increasing process counts.

As the message size increases, the performance differences among broadcast algorithms become more pronounced. Starting from 32 processes, the Pipeline algorithm consistently

exhibits the highest latency, making it the least efficient option, particularly for larger messages. The Basic Linear algorithm also performs poorly. The Binomial and Chain algorithms perform very similarly and consistently maintain low latency across all process counts and message sizes. Interestingly, the Default algorithm often achieves the best performance, especially as the process count and message size grow. This suggests that the Default implementation is optimized to balance latency and communication overhead in practical scenarios.

3.2 Reduce Algorithms

The following algorithms were evaluated for the `MPI_Reduce` operation: Binomial Tree, Rabenseifner, and Binary.

Each of these algorithms implements the reduction operation with different communication patterns and trade offs. While the Binomial Tree and Binary algorithms emphasize structured pairwise reductions, the Rabenseifner method is optimized for large messages by combining reduce-scatter and gather phases. The Default selection often reflects Open MPI's internal tuning to balance performance across various scenarios. The two plots that follow analyze the performance of these algorithms.

Reduce Latency vs Number of Processes

Figure 4 presents the latency of the `MPI_Reduce` operation across increasing process counts for three representative message sizes.

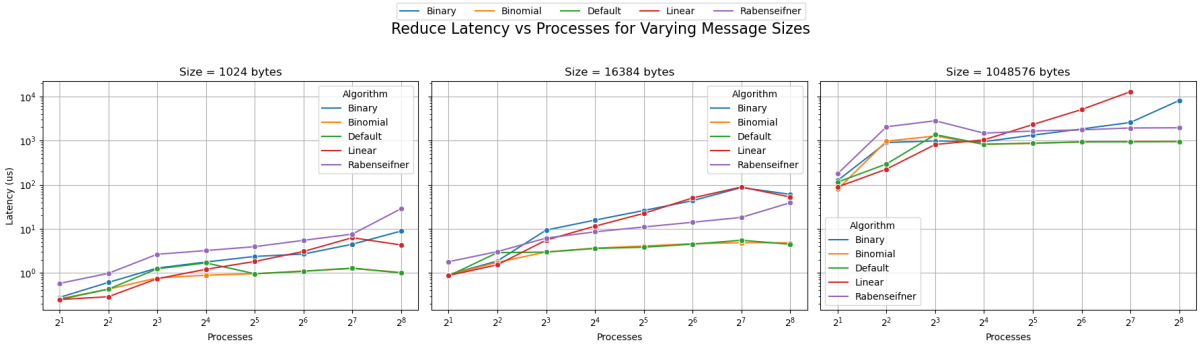

Figure 4: Latency of `MPI_Reduce` across different algorithms as the number of processes increases, for three message sizes: 1KB, 16KB, and 1MB.

For small messages, all algorithms exhibit relatively low latency, with Binomial and Default showing the best scalability. As the message size increases the Binomial and Default algorithms still perform well, while Linear and Binary show higher and less stable latency, especially with more than 16 processes. Rabenseifner improves and becomes one of the better-performing algorithms, validating its design for large data.

Reduce Latency vs Message Size

Figure 5 shows how the latency of `MPI_Reduce` varies with message size across different process counts.

The Default and Binomial algorithms consistently maintain low latency and scale well across both dimensions. The Linear algorithm exhibits poor scalability, especially at higher process counts and larger messages. Rabenseifner, while more efficient than Linear, tends to show the highest latency for small messages, but improves relatively for large message sizes. Overall, the Default and tree-based methods Binomial offer the best balance between scalability and performance.

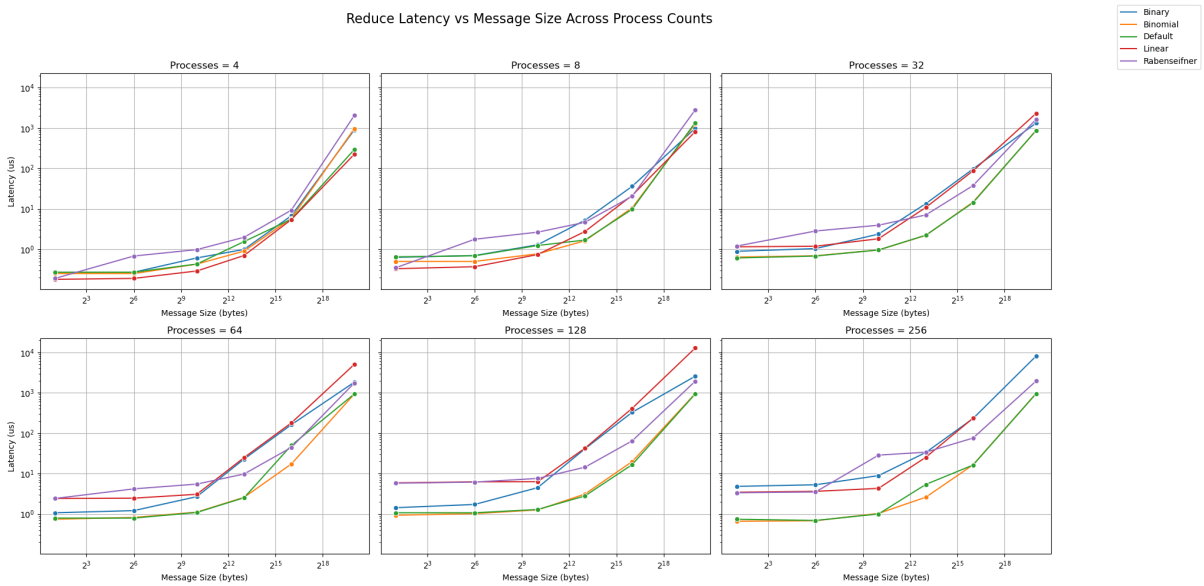

Figure 5: Latency of `MPI_Reduce` for varying message sizes across increasing process counts, comparing five different algorithms.

3.3 Point-to-Point Latency

To estimate the base communication costs used in the performance models for collective operations, we measured point-to-point latency between **core 0** and a selection of distant cores that represent meaningful architectural boundaries on the EPYC processors.

The benchmark used was `osu_latency` from the OSU pt2pt suite. Each test was conducted with:

- `-x 100`: 100 warm-up iterations
- `-i 1000`: 1000 measured iterations
- `-m 2:2`: fixed message size of 2 bytes

MPI ranks were explicitly pinned to selected cores using `--cpu-list` to ensure deterministic placement and avoid NUMA-induced noise. The latency was tested from core 0 to the following targets:

- Core 1 (Same CCX): 0.15 μs
- Core 4 (Same CCD): 0.32 μs
- Core 8 (Same NUMA domain): 0.35 μs
- Core 32 (Cross NUMA): 0.43 μs
- Core 64 (Cross socket): 0.69 μs
- Core 128 (Inter-node): 1.82 μs

These empirical values were used to construct a latency matrix $L(i, j)$ between any two cores i and j, defined based on the processor topology (CCX, NUMA, socket, node). This formed the foundation of the performance models for collective operations, allowing realistic predictions based on topology-aware communication costs.

4 Theoretical Model Comparison

To better understand the performance behavior of the different collective algorithms, simple analytical models for each algorithm were implemented. These models aim to estimate the total communication latency of the collective operations, using point-to-point communication costs obtained from the `osu_latency` benchmark. All models assume:

- Point-to-point latency dominates the cost (bandwidth effects are neglected)
- No overlap of computation and communication
- Perfect process placement

The latency model uses measured values for communication between core 0 and other selected cores, depending on the topology.

Broadcast Algorithms

Pipeline Broadcast

In the pipeline algorithm, each rank forwards the message to the next in sequence. The total latency is the sum of all pairwise latencies along the chain:

$$T_{\text{pipeline}}(P) = \sum_{i=0}^{P-2} \text{latency}(i \to i+1)$$

This assumes that each link in the pipeline must be traversed sequentially, and the initial segment must complete before the next begins.

Binomial Tree Broadcast

The Binomial Tree algorithm performs $\log_2 P$ communication steps from the root to increasingly distant ranks. At each step $i = 0, \ldots, \log_2 P - 1$, a message is sent from rank 0 to rank 2^i:

$$T_{\text{binomial}}(P) = \sum_{i=0}^{\log_2 P - 1} \text{latency}(0 \to 2^i)$$

This captures the geometric nature of the dissemination pattern in binomial trees.

Chain Broadcast (Parallel Chains)

The Chain algorithm can be parallelized by dividing the processes into f separate chains, each initiated by the root. Within each chain, the message is forwarded linearly. The total latency is determined by the longest chain, assuming chains operate in parallel without overlap:

$$T_{\text{chain-parallel}}(P, f) = \max_{j=1,\ldots,f} \left\{ \text{latency}(0 \to p_{j,1}) + \sum_{k=2}^{n_j} \text{latency}(p_{j,k-1} \to p_{j,k}) \right\}$$

where each chain j has n_j processes labeled $p_{j,1}, \ldots, p_{j,n_j}$.

Comparison with Measured Latency

Figure 6 compares the analytical models and empirical measurements for three broadcast algorithms on small messages (2 bytes). As expected, the Binomial Tree algorithm achieves the lowest latency and scales efficiently with process count.

The Pipeline algorithm exhibits linear growth in latency, both in measurement and model, with good agreement between the two. The sequential forwarding of messages across processes becomes increasingly costly at larger scales.

The Chain algorithm, when implemented with fixed parallel sub-chains, performs better than Pipeline in practice, particularly at larger scales. The parallel chain model captures the latency plateau effectively, although occasional spikes in the measured data likely stem from scheduling overheads or network imbalance.

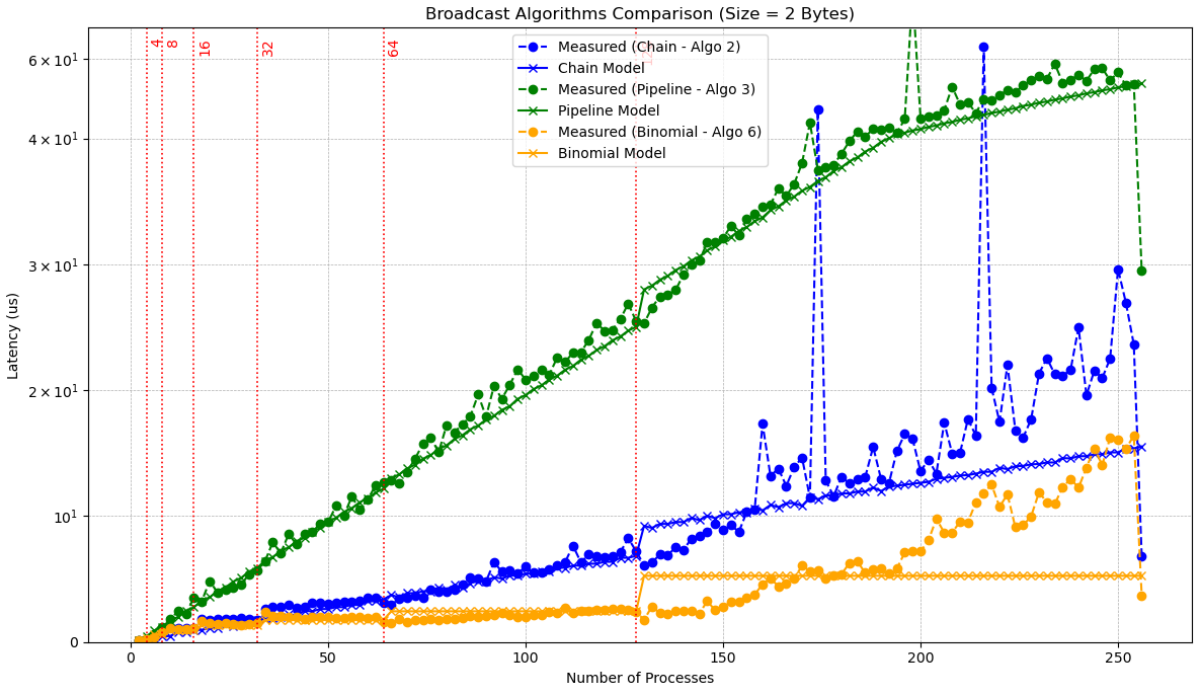

Figure 6: Comparison between measured and modeled latency for `MPI_Bcast` with message size 2 bytes.

Reduce Algorithms

Binary Tree Reduce

The communication pattern follows a full binary tree, with $\log_2 P$ levels. At each level, the slowest communication dominates:

$$T_{\text{binary}}(P) = \sum_{i=0}^{\log_2 P - 1} \max_r \left\{ \text{latency}(r + 2^i \to r) \right\}$$

This offers better scalability compared to the linear version.

Binomial Tree Reduce Model

At each step $i = 0, \ldots, \log_2 P - 1$, one message is sent from rank 2^i to the root. The total latency is:

$$T_{\text{binomial}}(P) = \sum_{i=0}^{\log_2 P-1} \text{latency}(2^i \to 0)$$

Rabenseifner Reduce

This algorithm consists of two phases: recursive halving (reduce-scatter) and binomial gather. At each phase $i = 0, \ldots, \log_2 P-1$, we consider the maximum latency between all communicating pairs:

$$T_{\text{rabenseifner}}(P) = \sum_{i=0}^{\log_2 P-1} \max_r \left\{\text{latency}(r \to r + 2^i)\right\} + \sum_{i=0}^{\log_2 P-1} \max_r \left\{\text{latency}(r + 2^i \to r)\right\}$$

In practice, the model selects appropriate latencies based on communication stride between processes.

Comparison with Measured Latency

The plot 7 shows the measured latency of three algorithms and the corresponding analytical model predictions. Vertical red lines mark process counts where architectural transitions occur.

The models effectively capture key trends in the measurements. The Binomial algorithm consistently shows the lowest latency among the three, both in the measured data and model prediction, highlighting its efficiency for small messages.

In contrast, the Binary algorithm displays significantly higher latency, especially beyond 128 processes, reflecting its poor scalability. The Rabenseifner model closely follows the measured performance, with minor overestimations due to conservative latency aggregation.

These results confirm that the binomial tree structure leads to superior scalability.

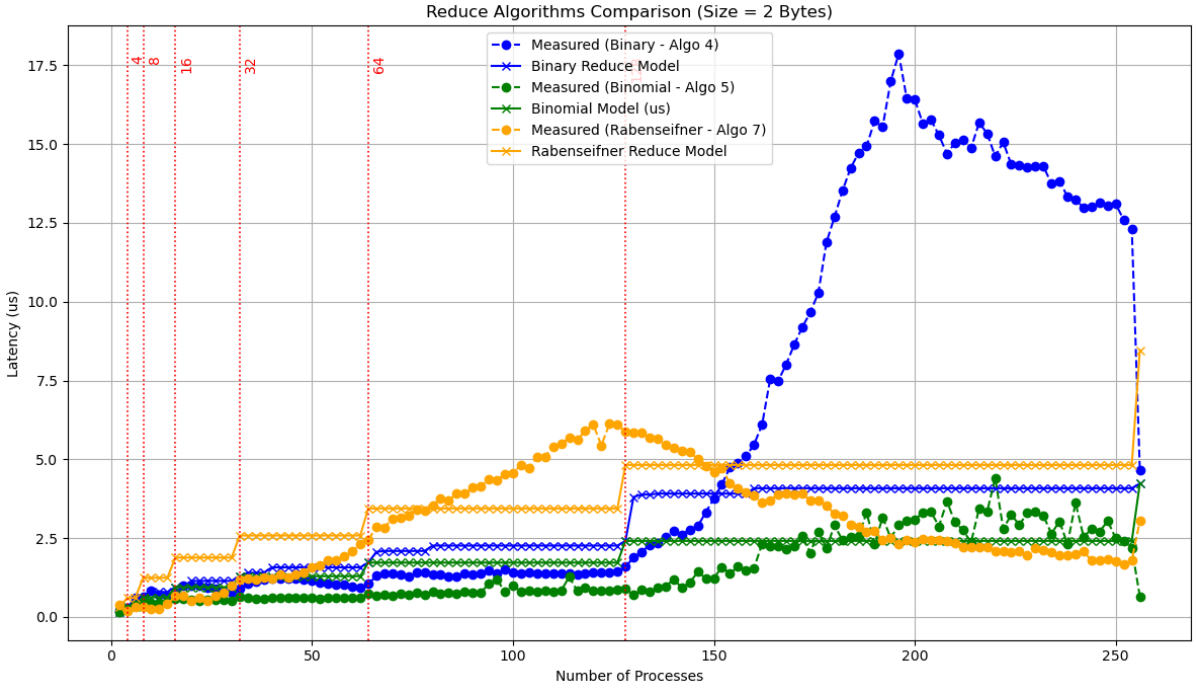

Figure 7: Comparison between measured and modeled latency for `MPI_Reduce` with message size 2 bytes.

Conclusion

This report evaluated OpenMPI collective communication algorithms for broadcast and reduce operations using OSU microbenchmarks on the ORFEO cluster. Default implementations were compared to manually strategies, analyzing how performance scales across varying message sizes and process counts.

The results confirm that topology-aware algorithms such as binary trees outperform simpler linear approaches, particularly at scale. Models based on latency measurements captured the overall trends well for small messages, while deviations at higher scales reflected unmodeled effects such as segmentation overhead and bandwidth saturation.

Importantly, topological shifts in latency visible in the core-to-core measurements aligned with performance discontinuities in collective operations, especially beyond 128 processes where inter-node communication becomes dominant. These architectural boundaries (e.g., CCX, CCD, NUMA, socket, and inter-node) significantly shape the behavior of collective operations and must be considered when designing models or tuning MPI implementations.

Overall, the findings highlight the importance of combining hardware-aware modeling with algorithm selection to achieve scalable and predictable performance in high-performance computing environments.