# High Performance Computing - Part 1

Comparison between OpenMPI algorithms for collective operation: Broadcast and Reduce

Munini Matteo

October 2025

**Abstract**

This report presents the benchmarking and performance modeling of MPI collective operations on the ORFEO cluster using the OSU Micro-Benchmarks. The study evaluates the `MPI_Bcast` and `MPI_Reduce` operations under multiple algorithmic variants of the OpenMPI tuned component. Experiments were conducted on two EPYC nodes, measuring latencies across message sizes and process counts. Complementary point-to-point tests provided insights into inter-core communication costs. The results highlight algorithmic differences in scalability, with observed latencies compared against topology-aware analytical models.

## 1 Introduction

Efficient communication between processes is critical for high-performance parallel applications. The Message Passing Interface (MPI) defines a range of collective operations to coordinate data movement, with performance strongly influenced by the choice of algorithm. OpenMPI provides multiple algorithmic variants for each collective, and their efficiency depends on message size, process count, and system topology.

In this project, we benchmarked and modeled the performance of two collectives on the ORFEO cluster using the OSU Micro-Benchmarks:

- Broadcast: Basic Linear, Pipeline, and Binary Tree.
- Reduce: Binary, Binomial, and Rabenseifner.

Measurements were conducted on two EPYC nodes, with latency data collected across different message sizes and process counts. Point-to-point tests further enabled modeling of inter-core communication, accounting for CCX, CCD, NUMA, and inter-node effects. Analytical models were compared against empirical results, highlighting algorithmic differences in scalability and guiding informed algorithm selection.

# 2 Architecture and Software

All experiments were conducted on the EPYC partition of the ORFEO high-performance computing cluster. This partition comprises 8 compute nodes, each equipped with two AMD EPYC 7H12 CPUs (Rome architecture), providing a total of 128 cores and 512 GiB DDR4 memory per node. Each CPU is organized into four NUMA nodes, yielding eight NUMA domains per node, and the nodes are interconnected by a 100 Gb/s high-speed network.

*The EPYC Rome microarchitecture design*: each processor integrates 8 Core Complex Dies (CCDs), and each CCD contains two Core Complexes (CCXs), with 4 cores and 16 MB of shared L3 cache per CCX. Communication between CCDs is managed through a central I/O die, while inter-socket communication relies on AMD's Infinity Fabric.

*The software used*: the benchmarking experiments were executed using SLURM for job submission. The software stack included OpenMPI v4.1.6 and the OSU Micro-Benchmarks v7.4.

# 3 Experiment Design

In this section there will be presented the general configuration of each test ran, the two kind of tests and where to find and execute these tests in the GitHub repository.

## 3.1 General structure

Each experiment use OSU Micro-Benchmarks to measure the latency of a MPI collective operation by specifying an algorithm for doing that. On each script that run an experiment is specified a variable that refers to a specific collective operation, like

- `OSU_BCAST="$OSU_BENCHMARK_DIR/osu_bcast"`
- `OSU_REDUCE="$OSU_BENCHMARK_DIR/osu_reduce"`

For each test it has been required 256 cores, in 2 nodes with 128 tasks per node. There have been run thousands of iterations for statistical reliability purpose. It has been asked for exclusive node allocation. MPI processes were mapped by core to minimize NUMA effects and inter-node variability (process affinity). The job submission used the following configuration: `SBATCH --nodes=2`, `SBATCH --ntasks-per-node=128`, `SBATCH --exclusive`, and `--map-by core`.

## 3.2 The two tests proposed

The first experiment is defined as *fixed*, meaning that it has been defined to measure how the latency changes with the number of processes involved while keeping the message constant in its size. The message size is of 4 bytes and the processes varies from 2 to 256. Here, the goal is to understand how the algorithm

changes its performance by adding more processes. The second experiment is identified as *variable*, meaning that here the goal is to analyze the change in latency by increasing both the content size and the number of processes. A progression of processes is 2, 4, 8, ... to 256, while progression in size goes from 2 to 1024 bytes by following same increment rule of processes (power of 2). This experiment provides information on performance landscape across both dimensions, ideal for identifying where the algorithm transitions between different performance regimes appears.

### 3.3 Scripts in the repository

In the folder `source_code` in directory `Project 1` the user will find six scripts for each collective operation tested, a bash file for creating the structure of the directory and to download and install the OSU Micro-Benchmarks library `get_osu.sh` and a bash script to get point-to-point latency measurements `latency_test.sh`. Additional information on how to run the jobs and about each single script can be found on the ReadMe of the GitHub directory.

## 4 Description of algorithms used

It will be presented in this section all the algorithms tested, both for Broadcast and Reduce.

### 4.1 Latency Estimation

MPI point-to-point latencies were measured between representative core pairs spanning same-CCX/CCD/NUMA, cross-socket, and internode distances. Each test binds rank 0 to core 0 and rank 1 to a target core $i$, running multiple iterations of a 2-byte message. Results, from the `latency.txt` have been manually ordered into a symmetric latency matrix $T(i, j)$ representing the communication time between cores $i$ and $j$. This matrix serves as a baseline for modeling communication performance.

### 4.2 Broadcast - Linear

The Basic Linear broadcast algorithm operates as follows: the root process sends the full message directly to all other ranks. In Open MPI's tuned component, this is implemented as a flat, single-level tree with $(p-1)$ children. Communication uses nonblocking sends, and total time depends on the slowest link and small per-process overheads.

**Total communication time:**

$$T(n) = \max_{p=1}^{n-1} T_{r\to p} + O \cdot n$$

where $T_{r\to p}$ is the time from root to process $p$, and $O$ is the per-process overhead.

### 4.3 Broadcast - Pipeline

In the Pipeline algorithm, processes form a logical chain: the root sends to rank 1, which forwards to rank 2, and so on. For large or segmented messages, partial overlap of communication occurs; for small ones, latency accumulates along the chain.

**Total communication time:**

$$T(n) = \sum_{p=0}^{n-2} T_{p \to p+1}$$

where $T_{p \to p+1}$ is the time from process $p$ to $p$+1.

### 4.4 Broadcast - Binary Tree

The Binomial Tree algorithm disseminates messages in $\lceil \log_2 n \rceil$ rounds. At round $l$, each process forwards to a partner at distance $2^{l-1}$. Open MPI implements this as a balanced binomial tree, often segmented for larger payloads. For small messages, round-wise parent–child transfers dominate the time.

**Total communication time:**

$$T(n) = \sum_{lv=1}^{height_{tree}} \max_{node \in S(lv)} T_{P(node) \to node} + O \cdot n$$

where $height_{tree} = \lceil \log_2(n) \rceil$, $S(lv)$ is the set of nodes at level $lv$, $P(node)$ the parent node, $T_{P(node) \to node}$ the parent-to-child time, and $O$ the per-process overhead.

### 4.5 Reduce - Binary

The Binary Reduce algorithm forms a full binary tree with $\log_2 n$ levels. At each level $lv$, processes exchange data with partners $2^{lv}$ apart. Multiple pairs communicate concurrently, and total time is the sum of per-level maxima.

**Total communication time:**

$$T(n) = \sum_{lv=0}^{\log_2 n - 1} \max_{R} \{T(R + 2^{lv} \to R)\}$$

where $lv$ is the tree level (0 to $\log_2 n - 1$), $R$ the receiver rank, and $R + 2^{lv}$ the sender rank.

### 4.6 Reduce - Binomial

In the Binomial Reduce algorithm, at each step $i = 0, \ldots, \log_2 n - 1$, rank $2^i$ sends its data to the root (rank 0). Steps occur sequentially, so total latency sums across all transmissions.

**Total communication time:**

$$T(n) = \sum_{i=0}^{\log_2 n-1} T(2^i \to 0)$$

where $2^i$ is the sender rank at step $i$ and the root is rank 0.

### 4.7 Reduce - Rabenseifner

The Rabenseifner algorithm combines recursive halving (reduce-scatter) and a binomial tree allgather. At each step $i = 0, \ldots, \log_2 n - 1$, processes exchange data with partners $2^i$ apart in phase 1 and propagate results in phase 2. Level completion depends on the slowest transfer; total latency is the sum across both phases.

**Phase 1 – Reduce-Scatter:**

$$T_{\mathrm{RS}}(n) = \sum_{lv=0}^{\log_2 n-1} \max_R \{T(R \leftrightarrow R + 2^{lv})\}$$

**Phase 2 – Allgather:**

$$T_{\mathrm{AG}}(n) = \sum_{lv=0}^{\log_2 n-1} \max_R \{T(R + 2^{lv} \to R)\}$$

**Total latency:**

$$T(n) = T_{\mathrm{RS}}(n) + T_{\mathrm{AG}}(n)$$

where $lv$ is the level index (0–$\log_2 n - 1$), $R$ the communicating rank, $2^{lv}$ the distance, and $\leftrightarrow$ / $\to$ denote bidirectional and unidirectional exchanges.

## 5 Performance analysis: Broadcast - Fixed experiment

The broadcast algorithms' performances will be analyzed here in the `FIXED` setting, where the purpose is to measure how the measured latency changes as the number of processes increases. As first analysis, for each algorithms tested it has been compared the experimental latency measured on Orfeo with the theoretical predictions. Interesting is the evaluation of the concordance between theoretical and experimental data in terms of errors made as well as general performance.

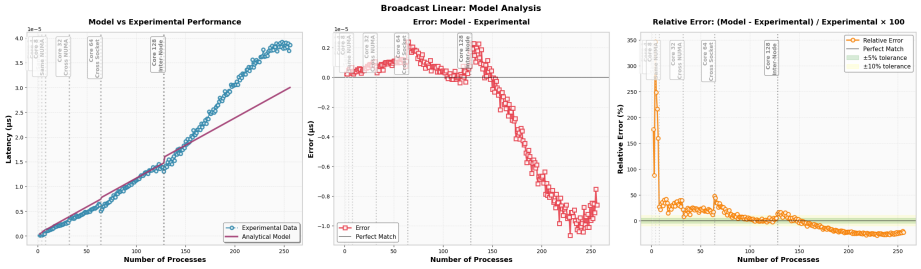



Figure 1: Linear Broadcast: Theoretical vs Experimental data

From Figure 1, the Linear Broadcast has an overall excellent match with the theoretical model until 150 processes involved, meaning that with introduction of two nodes a clear divergence appears. Even for the Pipeline Broadcast (Figure 2) there is a very good agreement with theory. The model captures the scaling behavior accurately, with only minor deviations and two outliers after introduction of second node.

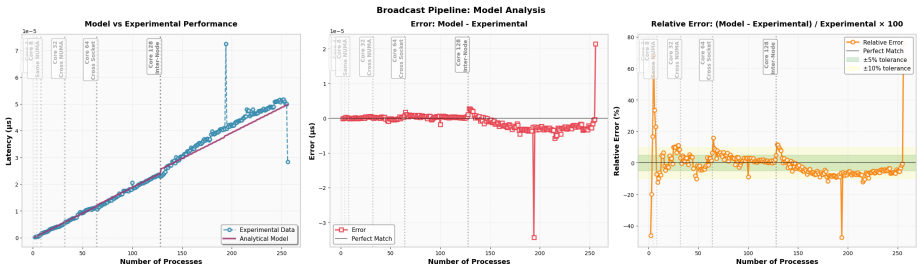



Figure 2: Pipeline Broadcast: Theoretical vs Experimental data

However, looking at Figure 3, the Binary Tree Broadcast has a good fit for low to moderate process counts, but the model struggles again when second node reached. Here, the experimental latency increases more steeply than predicted. However, the magnitude of the errors in both broadcast algorithms presented is low.

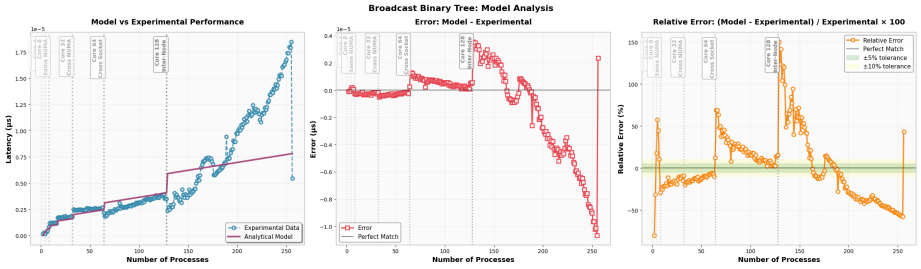
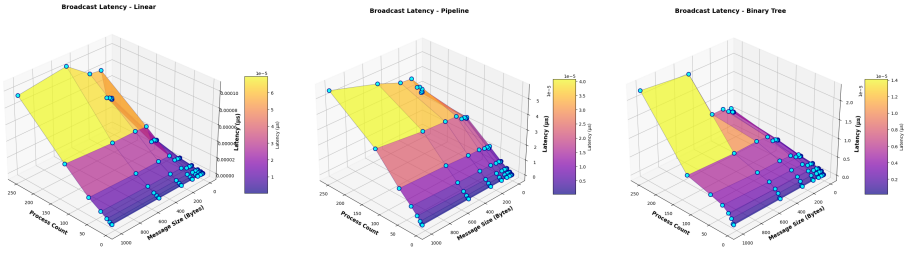



Figure 3: Binary Tree Broadcast: Theoretical vs Experimental data

# 6 Performance analysis: Broadcast - Variable experiment

Here, the three broadcast algorithms' performances will be analyzed in the `VARIABLE` setting, meaning that it has been measured the latency for each combination of process count and message size. The purpose is to understand how latency in each algorithm scales in terms of two factors: size of message and number of processes. In Figure 4 the 3D plots for each algorithms have been reported:




(a) Linear (b) Pipeline (c) Binary Tree

Figure 4: Broadcast Algorithms: Evolution of latency in terms of size and processes

From Figure 4 it is clear that the most the Binary Tree algorithm is the one that shows a generally less steep behavior across all possible grid configurations, and when considering combinations with a high number of processes and maximum message size, the latency remains the lowest in terms of order of magnitude. The Linear algorithm shows the steepest behavior, meaning very affected by the inter-node links and overhead on root since $n-1$ processes send messages to it. Pipeline shows better results than Linear, and its plot shows latency increases with process count, due to sequential message forwarding.

# 7 Performance analysis: Reduce - Fixed experiment

Now, in this section, the Reduce algorithms' performances in the `FIXED` setting are presented. Again, the theoretical predictions are compared with the measured data in Orfeo.

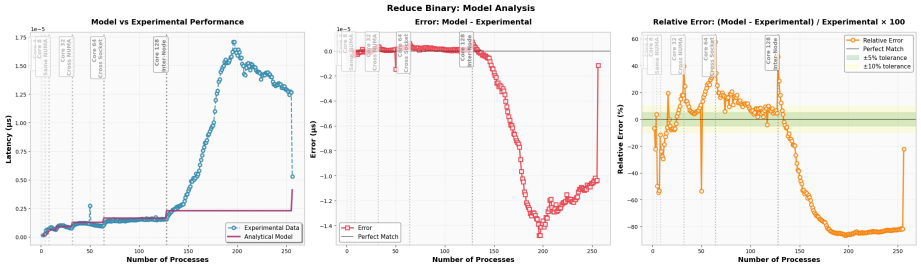



Figure 5: Binary Reduce: Theoretical vs Experimental data

On Figure 5, the Binary Reduce performances show a reasonable fit for lower process counts (<130), but shows systematic deviation afterward. Relative errors remain mainly within ±10–20% but increase significantly at higher process counts.

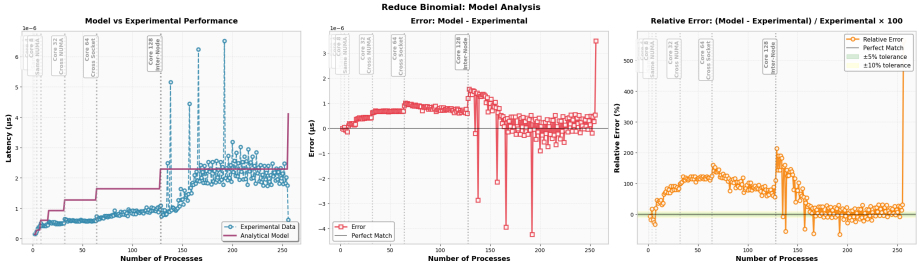



Figure 6: Binomial Reduce: Theoretical vs Experimental data

The Binomial Reduce (Figure 6) shows a concordance behavior between theoretical and experimental latency which is opposite to the others algorithms, since there is a good concordance once second node is introduced.

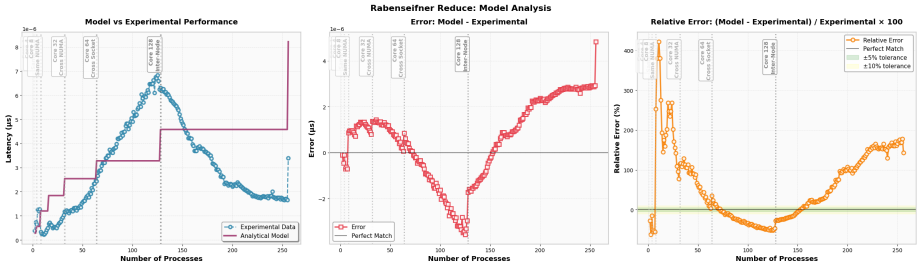
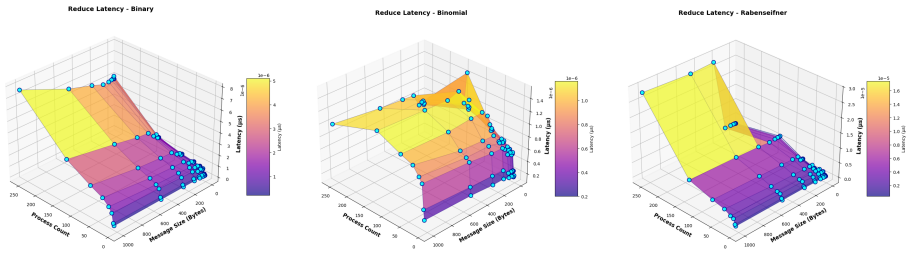



Figure 7: Rabenseifner Reduce: Theoretical vs Experimental data

Finally, on Figure 7, the Rabenseifner theoretical models underestimate latency between 64 to 128 processes range, and then it overestimate it when second node is introduced.

# 8 Performance analysis: Reduce - Variable experiment

Reduce algorithms' performances from the `VARIABLE` setting, are presented. From Figure 8 the Reduce Binomial has the steepest behavior, however, it is the model that has lower latency in general comparing all the grid combinations. While, in terms of latency performance, the worst performance is achieved by the Rabenseifner algorithm.




(a) Binary (b) Binomial (c) Rabenseifner

Figure 8: Reduce Algorithms: Evolution of latency in terms of size and processes

# 9 Final comparison

In the `FIXED` setting, for *Broadcast* algorithms the highest concordance between theoretical and experimental data is achieved in the *Pipeline* algorithm. The theoretical models for *Linear* and *Binary Tree* tends to underestimate latency generally after the introduction of second nodes. In this settings, overall the lowest latency is achieved by *Binary Tree* algorithms over number of processes. For *Reduce* algorithms both better fit between theoretical and experimental data, and performance in terms of latency is achieved by the *Binomial* algorithm. Overall, the *Binary* algorithms shows worst performance over the three algorithms analyzed. In the `VARIABLE` setting, for *Broadcast* algorithms the *Binary Tree* consistently outperforms the others, maintaining an almost constant latency even at high process counts. For *Reduce* algorithms, overall the *Binomial* algorithm performs better, especially when number of processes increase. While overall the *Rabenseifner* has the worst performances. The surface characteristics reveal distinct topology-related thresholds, corresponding to intra-CCD/NUMA, inter-socket, and inter-node communication domains, whose impact intensifies as the number of processes increases.

# 10 Conclusion

Using simple, topology-aware, latency-based models, we analyzed `MPI_Bcast` and `MPI_Reduce` on two EPYC nodes on ORFEO, with small messages, compar-

ing three algorithms per collective operation presented. Topological transitions observed in core-to-core latency measurements align closely with performance discontinuities in collective operations, particularly beyond 128 processes where inter-node communication dominates. As execution scales from intra- to inter-node domains, most latency-based models lose predictive accuracy due to unmodeled factors such as bandwidth contention and NUMA-induced variability. These architectural boundaries—across CCX, CCD, NUMA, socket, and inter-node levels—strongly couple communication efficiency with system topology. Despite the small-message and two-node scope, the findings emphasize the need to incorporate hardware topology into both algorithm selection and performance modeling to achieve reliable scalability in collective communication.