MSc in Data Science and Artificial Intelligence

High Performance Computing A.A. 2023/2024

Exercise 1: Comparison OpenMPI algorithms for Collective Operations

Broadcast and Reduce Operations

Prepared by:

Gabriel Masella
University of Studies of Trieste, Italy
Escuela Politécnica Superior, Universidad de Alicante, Spain

February 17, 2025

Contents

1 Introduction

In the field of High-Performance Computing (HPC), efficient communication between processes is important to achieving scalable performance. The Message Passing Interface (MPI) standard offers a suite of collective operations that facilitate coordinated data exchange across distributed processes. *Broadcast* and *Reduce* operations are foundational for many parallel algorithms, and understanding their performance characteristics and the impact of algorithm selection is crucial for optimizing applications on modern clusters.
This report evaluates the latency of different OpenMPI algorithms for these collective operations, using the OSU benchmarks on the ORFEO cluster, to identify optimal strategies under varying conditions. The study focuses on the following operations:

- *Broadcast*, comparing algorithms like linear, chain, pipeline and binary tree
- *Reduce*, evaluating the equivalent algorithms.

Using the OSU benchmarks, we measured latency across varying message sizes and process counts on ORFEO's different nodes, like *AMD EPYC 7H12* based on "Rome" architecture and *Intel Xeon Gold*, that are of two different types, *THIN* or *FAT*. By analyzing these results, we aim to derive performance models that explain algorithmic efficiency and guide optimal configuration choices for specific HPC workloads. The methodology, results, and insights presented here underscore the importance of algorithm-aware tuning in MPI-based applications.

2 Broadcast and Reduce

The *broadcast* operation is one of the most fundamental collective communications in MPI, used to disseminate a message from a single root process to all other processes. In our experiments, we evaluated four *broadcast* algorithms. The **basic linear** algorithm sends the message sequentially from the root to each process, which is simple but often inefficient for large process counts. The **chain** algorithm improves on this by having each process forward the message to the next, thus distributing the communication load. The **pipeline** algorithm takes this further by splitting the message into segments so that different parts can be transmitted concurrently, enhancing throughput for larger messages. Finally, the **binary tree** algorithm organizes processes in a tree-like structure, significantly reducing the number of communication steps and often yielding better scalability.
Similarly, the *reduce* operation combines data from all processes using an associative operation and returns the result to the root process. Here, the **linear** *reduce* aggregates data sequentially, which can be a bottleneck in high-process environments. The **chain** approach propagates partial reductions along a sequence of processes, aiming to distribute the workload more evenly. The **pipeline** method overlaps computation with communication by segmenting the data, which can be advantageous when handling large message sizes. The **binary** reduction, akin to its *broadcast* counterpart, organizes the reduction in a tree structure, reducing the number of communication rounds and typically offering superior performance as the number of processes increases.
Together, these four algorithms for each operation provide a spectrum ranging from simple, sequential approaches to more sophisticated, scalable methods. By benchmarking these algorithms across different hardware architectures and configurations, we can analyze the trade-offs in latency, scalability, and communication efficiency, and better understand how each algorithm's design interacts with the underlying system architecture.

3 Experimental Results

3.1 Setup

A set of scripts has been developed to evaluate the performance of MPI collective operations, specifically Broadcast and *reduce*, across various architectures, including FAT, THIN, and EPYC nodes.
The OSU Micro-Benchmarks are used to assess these operations under diverse configurations. For the FAT nodes, consisting of two nodes with 18 cores each, the mapping is done by NUMA domains or by core. In the case of the THIN nodes, which consist of two nodes with 12 cores each, the mapping is

done at the core level. The EPYC nodes, with two nodes and 128 cores each, are mapped by socket or core.
Tests systematically vary the number of cores per node, core-level mapping, and socket or core mapping for each node type. The number of processes (NP) ranges from 2 to 36 for FAT, 2 to 24 for THIN, and up to 256 for EPYC in extended runs, iterating through various algorithm implementations such as linear, chain, and binary tree for Broadcast, and linear, pipeline, and binary for *reduce*.
Each run saves message size, latency metrics (average, minimum, maximum), and iterations into CSV files, allowing for direct comparison of algorithm efficiency and scalability.
The approach emphasizes hardware-specific optimizations, such as NUMA awareness for FAT nodes and core parallelism for THIN and EPYC nodes, analyzing how latency scales with increasing processes, particularly for high-core-count EPYC architectures.
The insights gathered from this study aim to identify optimal algorithms, compare the moderate-scale performance of FAT and THIN nodes with the extreme scalability of EPYC, and inform HPC application tuning based on empirical latency trends across different architectures.

3.2 Algorithms Comparison

To effectively analyze the performance characteristics of MPI collective operations, we present our results through four comprehensive sets of visualizations. These visualizations examine both broadcast and reduce operations across varying message sizes and process counts on the EPYC architecture with core-level mapping.

3.2.1 Broadcast Performance Analysis

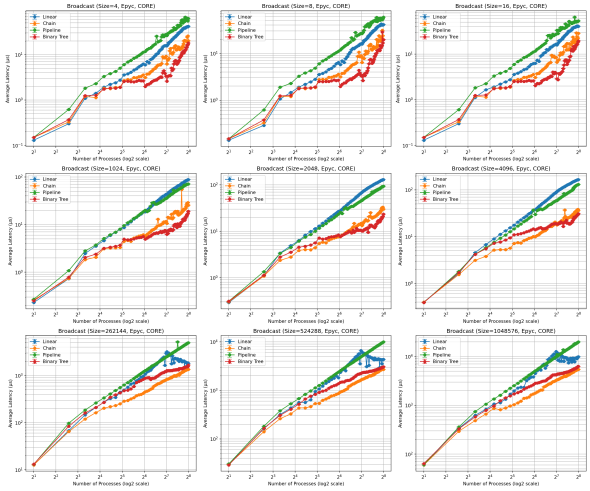

Figure 1: Broadcast latency vs number of processes for different message sizes

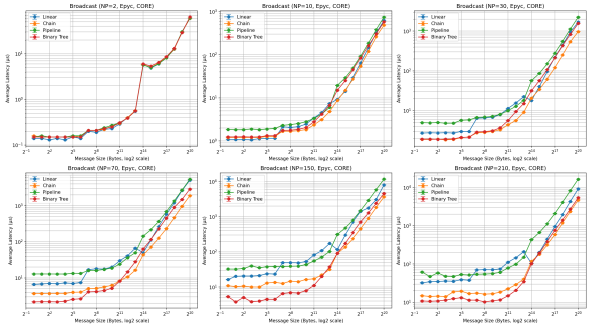

Figure 2: Broadcast latency vs message size for different process counts

The broadcast operation exhibits complex performance characteristics that vary significantly with both message size and process count. For small messages ($4 - 16$ bytes), the binary tree algorithm demonstrates superior scalability, maintaining consistently lower latency even as the number of processes increases. This advantage is particularly pronounced between 2^4 and 2^6 processes, where binary tree shows up to 40% lower latency compared to linear and chain approaches. The pipeline algorithm, while initially showing higher latency for small process counts, becomes increasingly competitive as message sizes grow beyond 1024 bytes. This trend becomes particularly evident in the range of 262KB to 1MB, where pipeline achieves the best performance among all algorithms, albeit with higher variability in latency measurements. An interesting crossover point occurs around 2048 bytes, where the relative performance of different algorithms shifts significantly. The chain algorithm, which performs moderately well for small messages, begins to show better scaling properties than the linear approach but cannot match the efficiency of binary tree or pipeline implementations for larger message sizes. The linear algorithm's performance degrades most notably with increasing process counts, showing a nearly linear increase in latency beyond 64 processes, making it unsuitable for large-scale deployments.

3.2.2 Reduce Performance Analysis

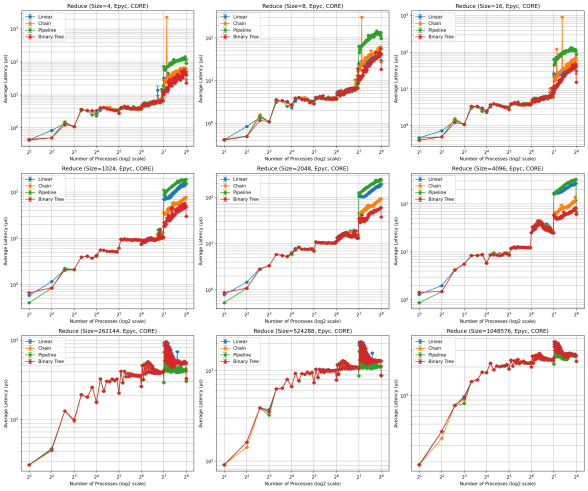

Figure 3: Broadcast latency vs number of processes for different message sizes

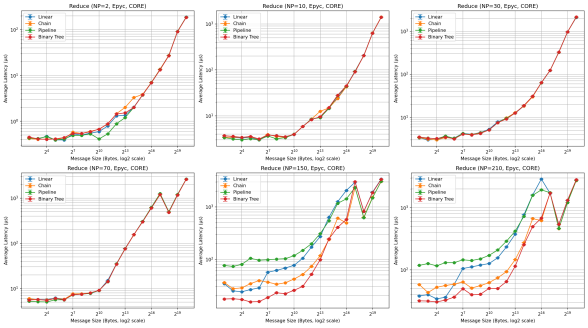

Figure 4: Broadcast latency vs message size for different process counts

The reduce operation presents a markedly different performance profile compared to broadcast. For small process counts ($NP = 2$ to $NP = 30$), all algorithms show relatively similar performance across message sizes up to 2^10 bytes, with binary tree maintaining a slight advantage. However, as the process count increases beyond 70, the performance characteristics diverge significantly. The binary tree algorithm maintains its efficiency for small to medium message sizes but shows increased volatility for larger messages, particularly evident in the tests with $NP > 150$. A notable observation is the appearance of significant performance fluctuations in the large message size regime ($> 262KB$) for all

algorithms when operating with high process counts. These fluctuations are particularly pronounced for the pipeline and linear algorithms, which show latency spikes at certain process counts. The chain algorithm demonstrates more stable performance characteristics but generally higher latency compared to binary tree for most configurations. This suggests that the reduce operation is more sensitive to system architecture and process placement than broadcast, particularly when handling large data volumes. The performance data indicates that optimal algorithm selection for reduce operations should prioritize binary tree for configurations with process counts below 150, regardless of message size. For larger process counts, the choice becomes more nuanced, with pipeline potentially offering better stability for medium-sized messages ($4KB$-$256KB$), despite not achieving the peak performance of binary tree under optimal conditions. The linear and chain algorithms, while simple to implement, generally show inferior performance except for very small messages or process counts, making them suitable mainly for development or debugging purposes.

3.3 Nodes Comparison

The performance characteristics of broadcast and reduce operations were evaluated across three different node architectures: THIN (12 cores/node), FAT (18 cores/node), and EPYC (128 cores/node). The analysis reveals distinct behavioral patterns for each collective operation and node type under varying message sizes and process counts.

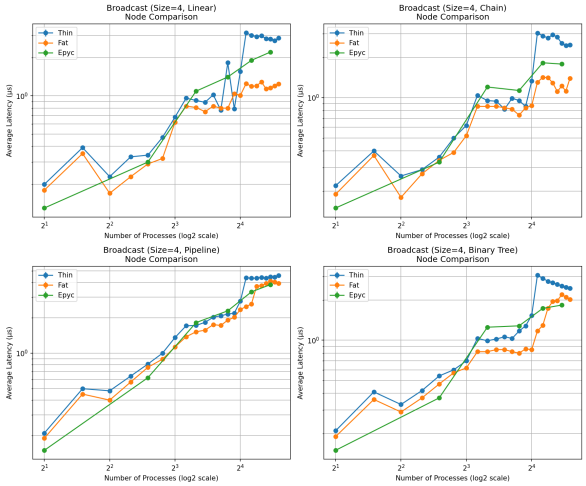

Figure 5: Broadcast performance comparison across node types with message size = 4 bytes

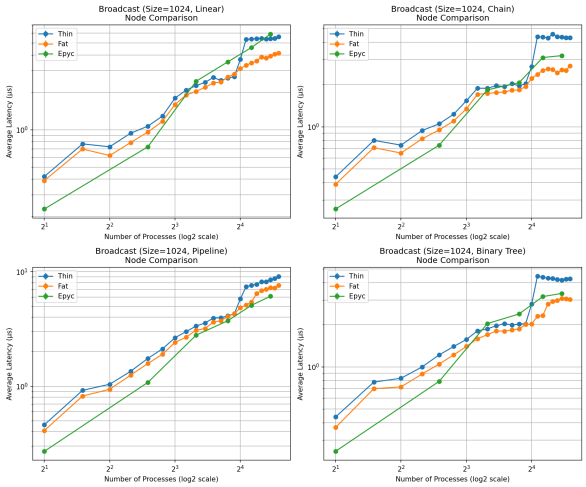

Figure 6: Broadcast performance comparison across node types with message size = 1024 bytes

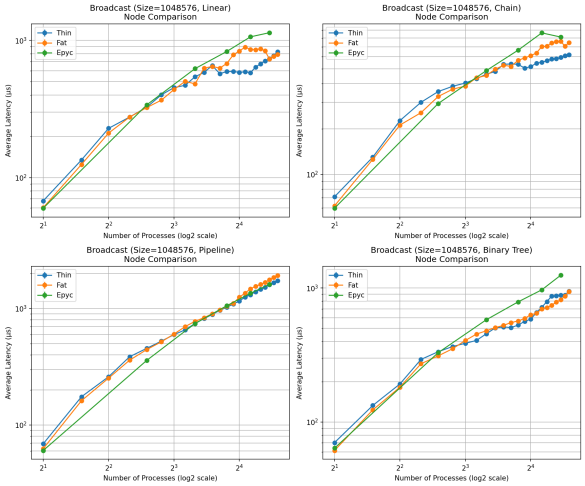

Figure 7: Broadcast performance comparison across node types with message size = 1048576 bytes

3.3.1 Broadcast Performance

The broadcast operation exhibits several interesting performance characteristics across the different node architectures. For small message sizes (4 bytes), as shown in Figure 5, all node types demonstrate relatively consistent latency patterns, with EPYC nodes generally showing lower initial latency compared to THIN and FAT nodes. However, as the number of processes increases beyond 2^3, the performance patterns begin to diverge significantly.

With medium-sized messages (1024 bytes), illustrated in Figure 6, the performance gap between architectures becomes more pronounced. The pipeline algorithm shows the most consistent scaling across all node types, with latency increasing linearly with process count. The binary tree algorithm demonstrates particularly efficient performance for EPYC nodes at higher process counts, likely due to its ability to better utilize the higher core count and memory bandwidth available on these nodes.

For large messages (1048576 bytes), as depicted in Figure 7, the impact of node architecture becomes most apparent. EPYC nodes maintain better latency characteristics at high process counts, particularly when using the pipeline algorithm. The linear and chain algorithms show similar performance patterns, with THIN nodes experiencing higher latency variability at process counts above 2^4.

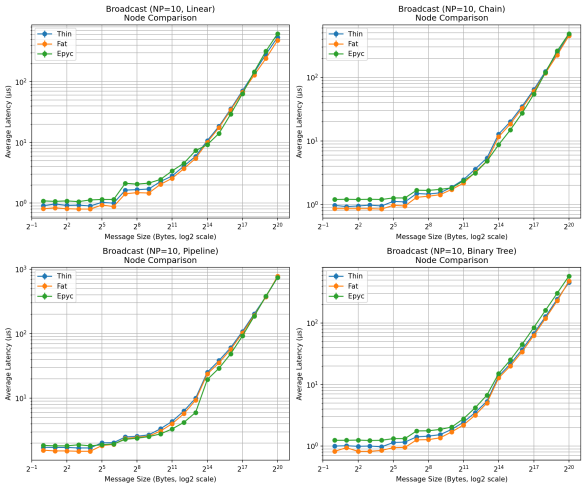

Figure 8: Broadcast performance with fixed number of processes (NP=10) across message sizes

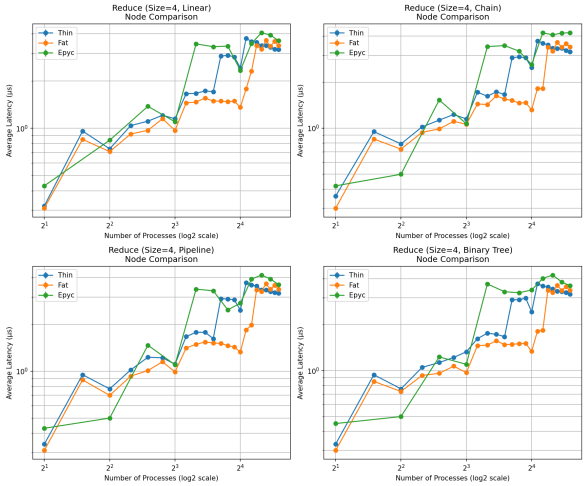

Figure 9: Reduce performance comparison across node types with message size = 4 bytes

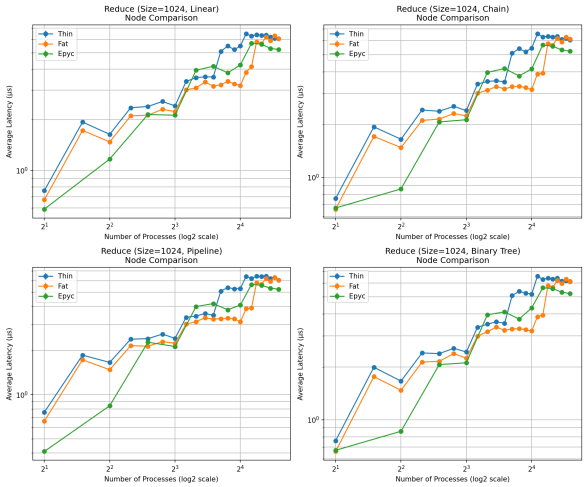

Figure 10: Reduce performance comparison across node types with message size = 1024 bytes

3.3.2 Reduce Performance

The reduce operation presents different scaling characteristics compared to broadcast, with more pronounced differences between node architectures. At small message sizes (4 bytes), as shown in Figure 9, all algorithms show similar performance patterns initially, but diverge significantly as process counts increase. EPYC nodes demonstrate notably different behavior patterns, particularly in the 2^3 to 2^4 process range, where they often show temporary performance degradation before stabilizing.

With medium-sized messages (1024 bytes), as illustrated in Figure 10, the performance variations between node types become more pronounced. FAT nodes generally maintain more stable latency patterns compared to THIN nodes, particularly when using the binary tree algorithm. EPYC nodes show superior scaling characteristics at higher process counts, though with more pronounced latency fluctuations compared to their broadcast performance.

For large messages (1048576 bytes), shown in Figure 11, the reduce operation shows the most significant architectural impact. All node types exhibit increased latency variability, but EPYC nodes maintain better overall scaling characteristics. The pipeline algorithm demonstrates the most consistent performance across all architectures, while the binary tree algorithm shows increased sensitivity to node architecture at high process counts. Notably, THIN nodes experience more pronounced latency spikes at certain process counts, particularly visible in the 2^3 to 2^4 range, suggesting potential resource contention issues.

As demonstrated in Figure 8 and Figure 12, the fixed-process ($NP = 10$) tests across varying

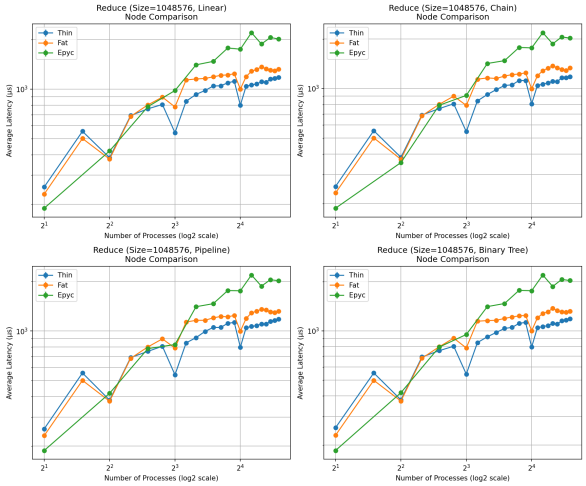

Figure 11: Reduce performance comparison across node types with message size = 1048576 bytes

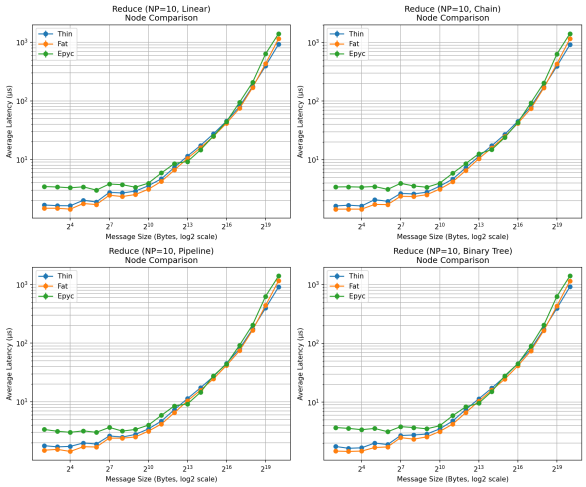

Figure 12: Reduce performance with fixed number of processes (NP=10) across message sizes

message sizes reveal that the performance difference between node architectures becomes negligible for small to medium message sizes, with divergence only becoming significant beyond 2^16 bytes. This indicates that node architecture choice becomes particularly critical for applications dealing with large message sizes or high process counts, while being less crucial for smaller-scale operations.