High Performance Computing - Exercise 1

Alessandro Minutolo

1 Introduction

This report explores the efficacy of various algorithms implemented within OpenMPI for optimizing collective operations, a cornerstone functionality in parallel programming environments. Specifically, the exercise focuses on evaluating the performance of different algorithms for the broadcast operation, which is pivotal for data distribution among processes in a parallel system, and for the gather operation.

The analysis is structured around the utilization of the OSU micro-benchmarks, which serve as a standardized metric to analyze the performance of MPI operations across varying conditions. This approach is instrumental in identifying optimal configurations that reduce execution time and enhance the efficiency of data exchanges in multi-node environments. Conducted on the ORFEO cluster, this exercise leverages its diverse computational nodes to give a detailed comparative analysis of the default OpenMPI implementation against alternative algorithms under different system loads and message sizes.

By integrating theoretical knowledge with practical benchmarks, some performance models were built in order to compare the real data to the ideal scenario.

2 Methods

This report encompasses a thorough analysis of the implementation and performance characteristics of selected MPI broadcast and gather algorithms within the OpenMPI library, vital for efficient data communication in parallel computing environments.

2.1 Data Collection

For the MPI broadcast operation, where a root process sends identical data to all other processes in a communicator, three specific strategies were analyzed:

- Basic Linear: Sends data sequentially from the root to each process through a non-blocking send promoting parallelization.
- Chain: Stages communication through a series of chains of processes.

- Pipeline: Simpler than the chain algorithm, it stages communication through a series of processes, lightening the workload for root process.

Conversely, the gather operation, which collects data from all processes to assemble at the root, was analyzed using three distinct algorithms:

- Basic Linear: Collects data sequentially.
- Binomial: Utilizes a binomial tree structure to improve efficiency by reducing the depth of communication.
- Linear with Synchronization: Adds synchronization steps to ensure data integrity during the collection process, sending firstly a segment of the message and then the remaining part.

2.2 Performance Models

This practical application allowed for assessing how theoretical models translate into real-world performance. To develop a simple performance model, it is necessary to estimate the latency of point-to-point communication routines, since collectives are built on top of them. The OSU benchmark was used again, which provides a tool that allows to determine the communication time between two cores that are on different locations; more precisely, there are six different communication times depending on whether two cores are on the same CCX, same CCD, same NUMA region, same socket, same node, different nodes. Through this detailed examination and practical testing, significant insights into the mechanics of data communication in distributed systems were gained. The exercise significantly contributed to understand how specific algorithmic choices can impact the performance of parallel computations, guiding future optimizations and implementations.

3 Implementation

In this project, a detailed methodology was employed to implement and evaluate the performance of MPI broadcast and gather algorithms using the OpenMPI library on the ORFEO cluster. The goal was to rigorously test these algorithms under varied computational conditions to derive insights into their practical behaviors in a distributed computing setting.

3.1 Execution Environment

Bash scripts were crafted to automate the submission of benchmark jobs via SLURM. These scripts were specifically designed to run the OSU micro-benchmarks suite for latency measurements of the chosen MPI operations. Each script was configured to vary the message size and the number of cores utilized, providing a comprehensive evaluation across different scales of operation. The experiments were conducted on two EPYC nodes with AMD processors, ensuring a

controlled environment for accurate and consistent benchmarking results. A critical feature employed in the bash scripts was the `--map-by` option used with the OSU benchmarks. This MPI runtime option allows for fine control over how processes are mapped to the hardware:

- `--map-by core`: Distributes processes tightly across CPU cores, potentially maximizing intra-node communication.
- `--map-by socket`: Allocates processes across sockets, potentially leveraging shared memory within sockets while introducing inter-socket communication.
- `--map-by node`: Spreads processes across nodes, resulting in increased inter-node communication overhead.

3.2 Data Collection and Analysis

Latency measurements were taken for a range of message sizes to determine scalability and efficiency across different algorithms. Additional tests measured the impact of varying the number of cores while keeping the message size constant. Data was meticulously analyzed to evaluate the performance characteristics associated with different `--map-by` configurations, providing deeper insight into how process placement affects algorithm efficiency. All the data were collected using two EPYC nodes (a total of 256 cores) on the ORFEO cluster

4 Detailed Analysis of MPI Broadcast Operations

This section discusses the results from the broadcast operation with a varying message size across different numbers of processes, focusing on the latency implications of different MPI algorithms and `--map-by` configurations.

4.1 Impact of Mapping Strategy

The testing was done collecting data with an increasing number of cores and message size as it is shown in Figures 1, 2 and 3

- **Core Mapping:** Provides the best performance for all three algorithms due to minimized communication overhead. It is ideal when processes can be tightly packed within a node
- **Socket Mapping:** Introduces moderate overhead due to inter-socket communication, which becomes more apparent in communication-heavy algorithms like Chain and Pipeline.
- **Node Mapping:** Performs the worst, as the reliance on inter-node communication significantly increases latency, especially for communication-intensive algorithms.

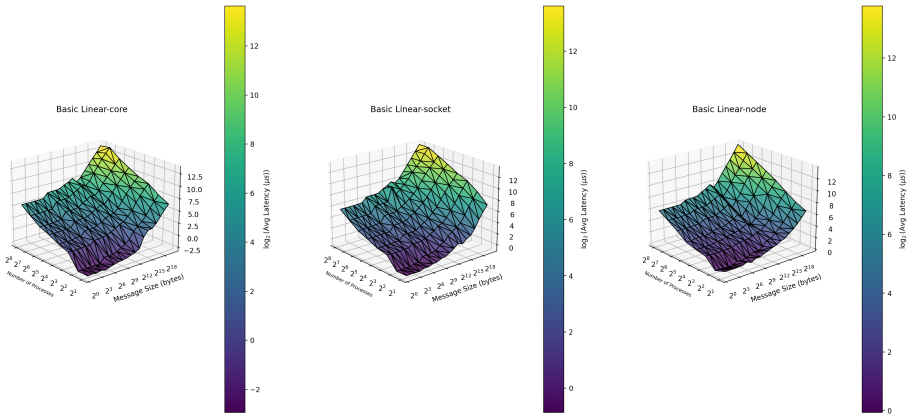

Figure 1: Different mapping for broadcast-basic linear

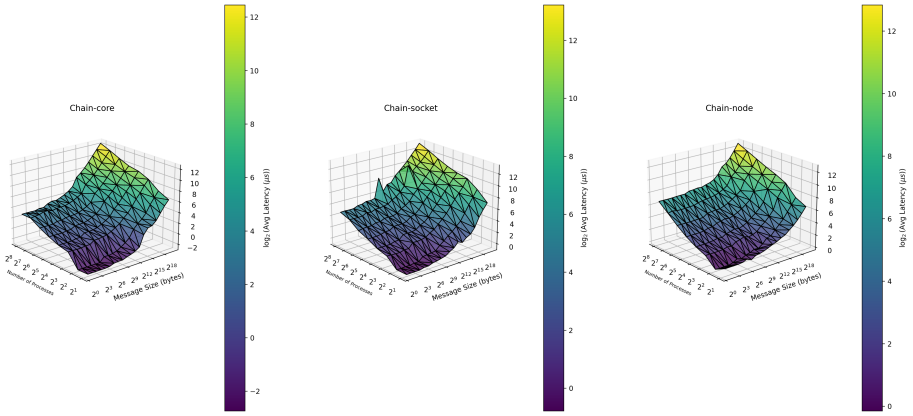

Figure 2: Different mapping for broadcast-chain

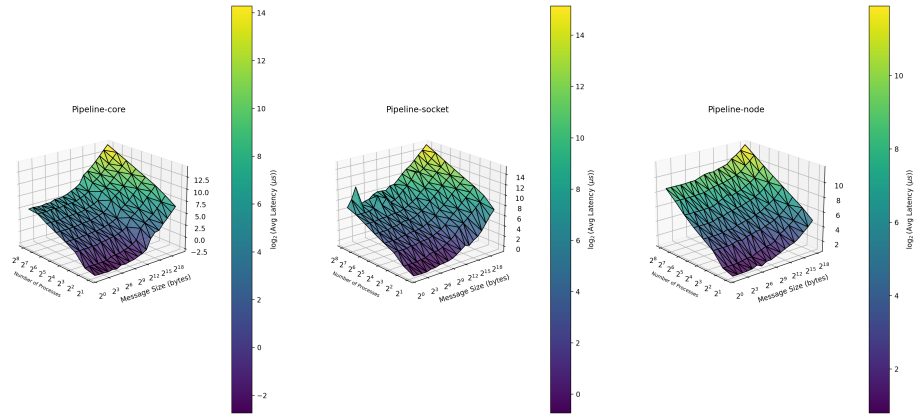

Figure 3: Different mapping for broadcast-pipeline

4.2 Broadcast Performance Models

For the performance models, the point to point communication time were taken considering a message size of 4 bytes. Also the real data were taken considering a fixed size of the message of 4 bytes with number of cores increasing from 2 to 256 (two full EPYC nodes).

4.2.1 Basic Linear Algorithm

Through the basic linear algorithm the root process is responsible to send the data to all the other processes using a non-blocking isend. In particular, before sending the message a request array (**reqs**) is created of size $N - 1$ where N is the number of cores involved. Considering that the isend is non-blocking, the total execution time will not be the sum of the individual point-to-point communication times, because when the root sends the messages, even if a process hasn't received it yet, it will keep sending messages to the other processes. Hence, the performance model will have this form:

$$T = \max(T_{reqs}) + overhead$$

where T_{reqs} is the point-to-point communication times between root and the other processes and **overhead** is an additive factor due to overloading of cores. In Figure 4, the blue points, labeled "Basic linear real," depict the actual latency observed during experiments. While the general trend of the real data follows the model, there is noticeable deviation, particularly as the number of cores increases. These discrepancies suggest that real-world factors, such as communication overhead, hardware limitations, or network contention, become more significant as the system scales. Overall, the graph demonstrates that while the theoretical model provides a useful baseline for understanding performance trends, real-world conditions introduce complexities that result in higher latency compared to the idealized predictions, especially at larger scales.

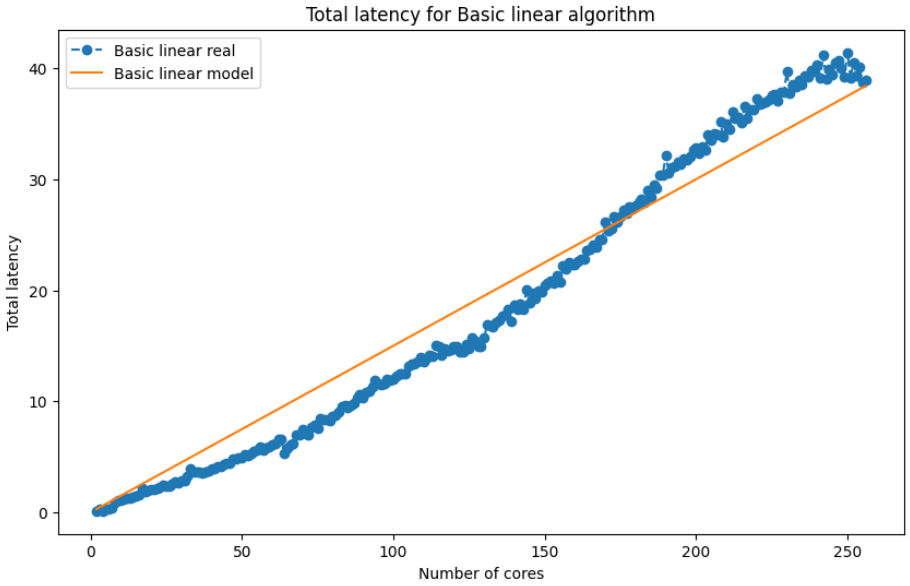

Figure 4: Performance model vs real model for broadcast-basic linear

4.2.2 Chain Algorithm

The chain-based broadcast algorithm organizes processes into multiple chains or paths emanating from the root process. Each chain serves as a conduit through which data is propagated, enabling parallel data transmission across the system. This structure reduces the number of communication steps compared to a linear broadcast, thereby decreasing overall latency. The number of chains is by default 4 from the OpenMPI settings. The performance model has been built has follows:

$$T = \max(T_{chain_i}), i = 1, ..., 4$$

where T_{chain_i} is the time to complete a message passing for the i-th chain and it is computed as follows:

$$T_{chain_i} = T_{0,i_1} + \sum_{k=1}^{m_i - 1} T_{i_k, i_k+1}$$

where T_{0,i_1} is the time to send the message from the root to the first process to the i-th chain (i_1) and m_i is the number of processes in the i-th chain.

As it is shown in Figure 5, the plot illustrates the performance deviation of the chain algorithm from its theoretical expectations as the system scales. It shows that the model is accurate for smaller numbers of cores but fails to account for real-world inefficiencies at larger scales, emphasizing the need to address scalability challenges in practical implementations.

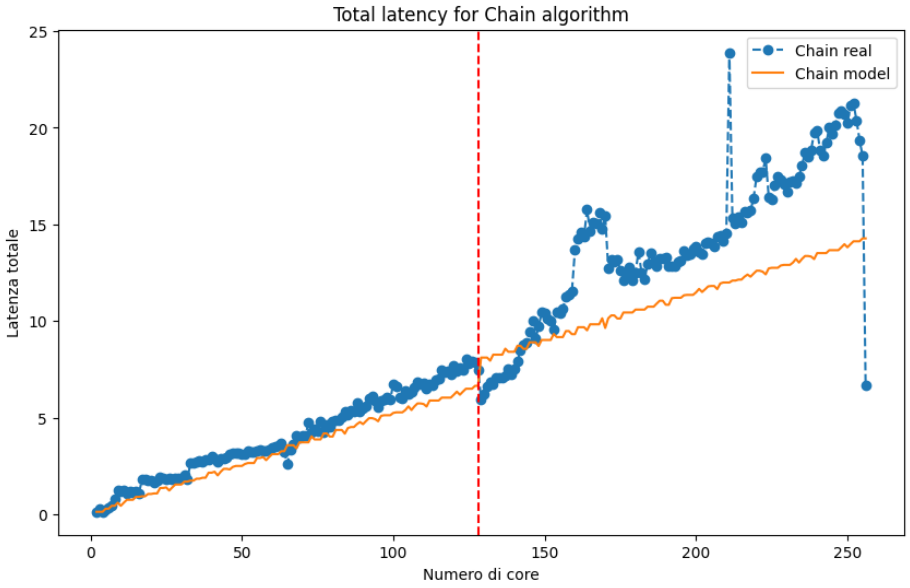

Figure 5: Performance model vs real model for broadcast-chain

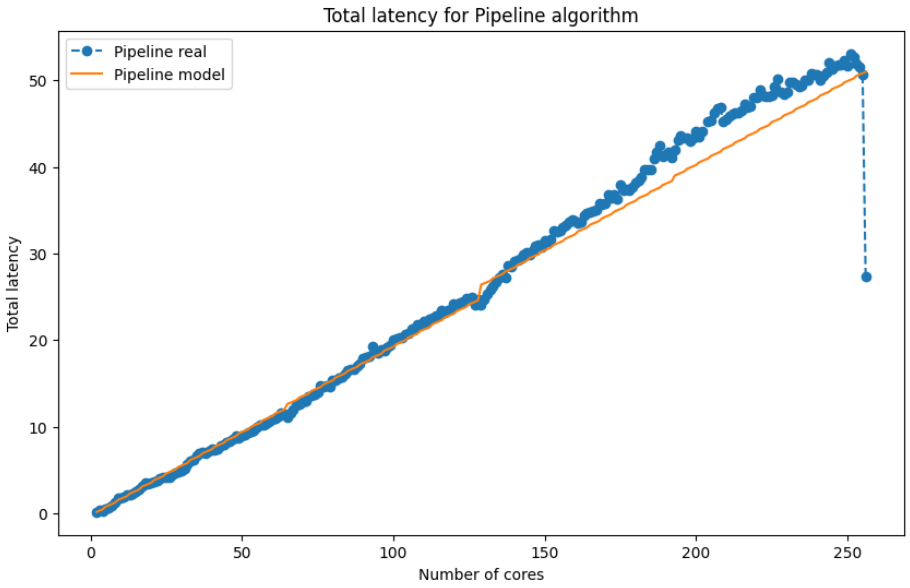

Figure 6: Performance model vs real model for broadcast-pipeline

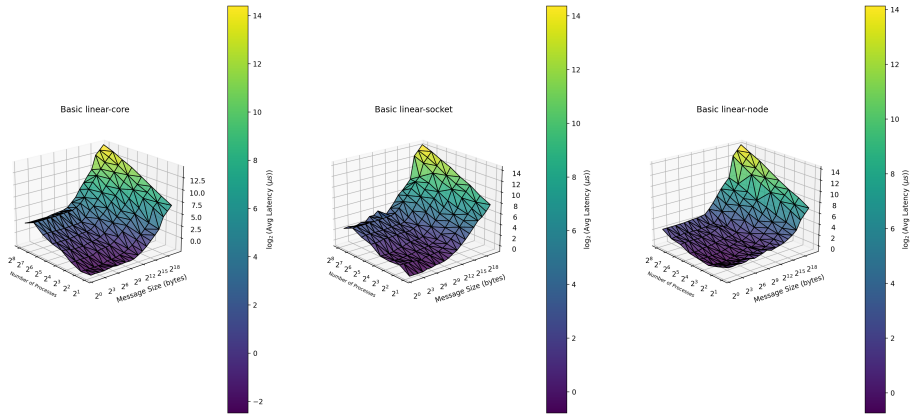

Figure 7: Different mapping for gather-basic linear

4.2.3 Pipeline Algorithm

Pipeline algorithm is very easy to implement and it is one of the most naive strategies for implementing broadcast. It consists of simply sending the data sub-sequentially core by core starting from the root. Hence, the performance model (Figure 6) has built as follows:

$$T = \sum_{i=0}^{N-1} T_{i,i+1}$$

where $T_{i,i+1}$ is the point to point communication time between two consequent cores and N is the number of cores.

5 Detailed Analysis of MPI Gather Operations

This section provides an in-depth analysis of the gather operation with a fixed message size across different numbers of processes, focusing on the impact of various MPI algorithms and `--map-by` configurations.

5.1 Latency Trends Across Processes

As the number of processes increases, most configurations exhibit a rise in latency, which aligns with expectations due to the increased overhead involved in collecting data from a greater number of sources. The testing was done collecting data with an increasing number of cores and message size as it is shown in Figures 7, 8 and 9

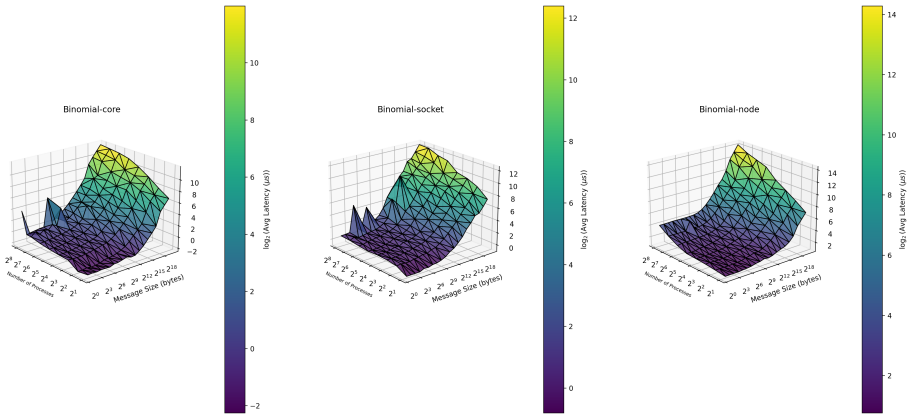

Figure 8: Different mapping for gather-binomial

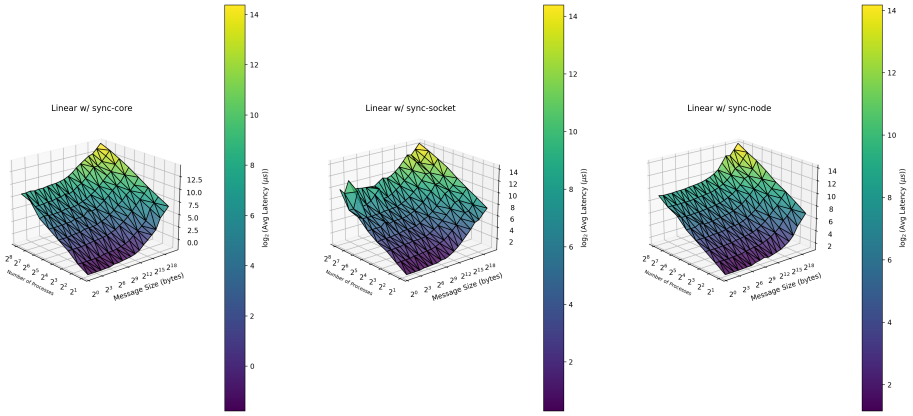

Figure 9: Different mapping for gather-linear w/ sync

5.1.1 Impact of Mapping Strategy

- **Core Mapping:** All algorithms perform best with this mapping due to minimized communication overhead. Binomial outperforms the others for larger scales due to its logarithmic communication structure.
- **Socket Mapping:** Performance degrades compared to core mapping as inter-socket communication introduces latency. Binomial remains more efficient due to its reduced communication steps.
- **Node Mapping:** All algorithms experience the highest latency with this mapping. Binomial is the most resilient, while Linear with Synchronization suffers the most due to compounded synchronization and inter-node overhead.

5.2 Gather Performance Models

As for Broadcast operation, the point to point communication time were taken considering a message size of 4 bytes. Also the real data were taken considering a fixed size of the message of 4 bytes with number of cores increasing from 2 to 256 (two full EPYC nodes).

5.2.1 Basic Linear Algorithm

The performance model for the basic linear algorithm has been the most challenging one. In theory, the root core has to receive all the messages from all the other cores sequentially, leading to a linear growth in execution time. The results are not really satisfying because it is clear that when the whole process starts to involve cores from the second node, the total time of the operation decreases with respect to the number of cores involved. The theoretical model (Figure 10) was built as follows:

$$T = \max(T_{reqs}) + overhead$$

with $T_{i,0}$ the point-to-point communication time between the cores in the array of senders and the root core.

5.2.2 Binomial Algorithm

The Binomial Gather algorithm uses a hierarchical, tree-based communication pattern, which theoretically minimizes the number of communication steps and reduces latency logarithmically as the number of processes grows. However, in practice, real-world factors such as network contention, synchronization delays, and non-uniform hardware performance introduce additional overhead that the model does not account for. The critical threshold at $x = 128$ represents the point where a second node starts to be involved and the communication time

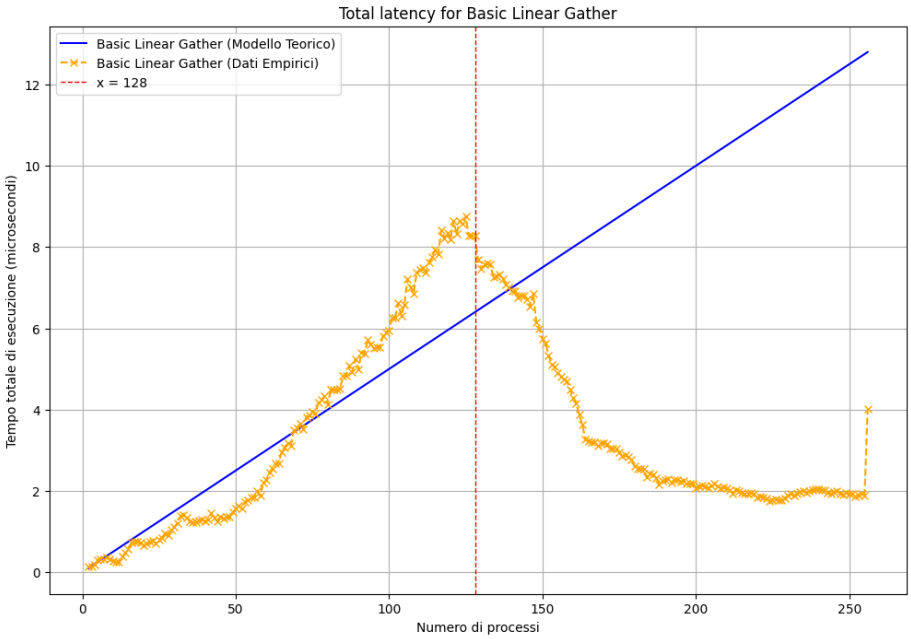

Figure 10: Performance model vs real model for gather-basic linear

significantly increases likely due to higher costs in transferring data across nodes. The performance model was built as follows:

$$T = \sum_{l=0}^{L-1} \max\{T_{i,i+2^l} \Big| i \quad \text{is a process at level l}\}$$

As it is shown in Figure 11, the empirical data aligns well with the performance model, showing small deviations, up to the first node completed. This suggests that the system is able to effectively utilize its resources and communication remains efficient, likely benefiting from intra-node or intra-socket communication. Then the empirical data diverges from the theoretical model, with latency increasing non-linearly and exhibiting significant fluctuations. This indicates that non-ideal factors, such as inter-node communication, contention for shared resources, or network overhead, start to dominate as the system scales. The theoretical model reflects the logarithmic communication pattern of the Binomial Gather algorithm. Latency increases in discrete steps as additional layers of the communication hierarchy are introduced. However, the real-world data shows that these steps are smoothed out or overwhelmed by other system-induced delays.

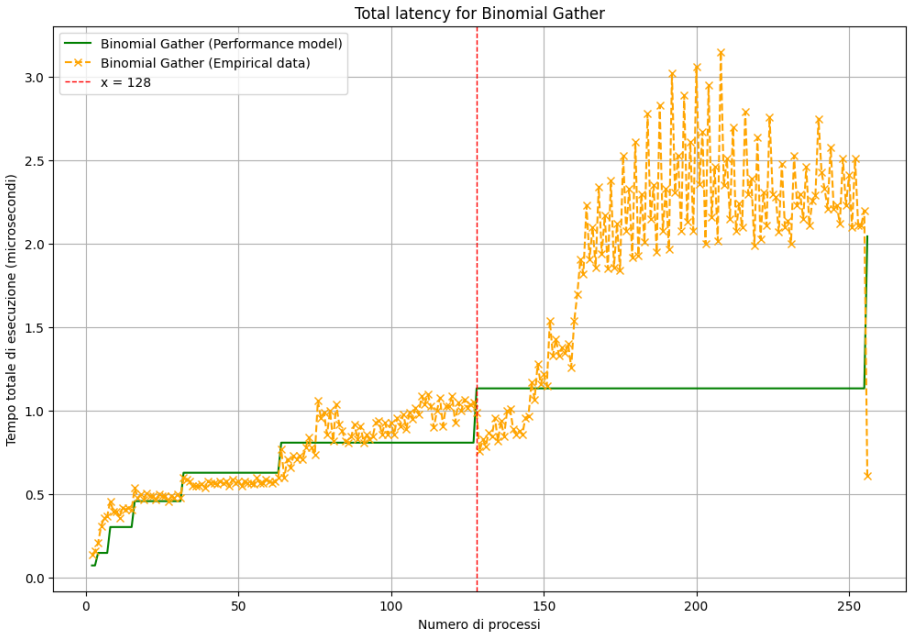

Figure 11: Performance model vs real model for gather-binomial

5.2.3 Linear with Synchronization

The linear with synchronization algorithm splits the message into two chunks. The first chunk is collected in sequence and the second one is collected asynchronously using non-blocking receives. The performance model was built as follows:

$$T = \max\{\frac{T_{i,0}}{2}\} + \sum_{i=1}^{N} \frac{T_{i,0}}{2}$$

where $T_{i,0}$ is the time of communication point to point between the i-th core and the root core. The two times are divided by 2 because of the chunk division of the message. As it is shown in Figure 12, the performance model can reflect quite well the real data; the only problem is that it is not capturing the increasing time when the number of cores is very large, probably because of some overhead in the communication with such many processes.

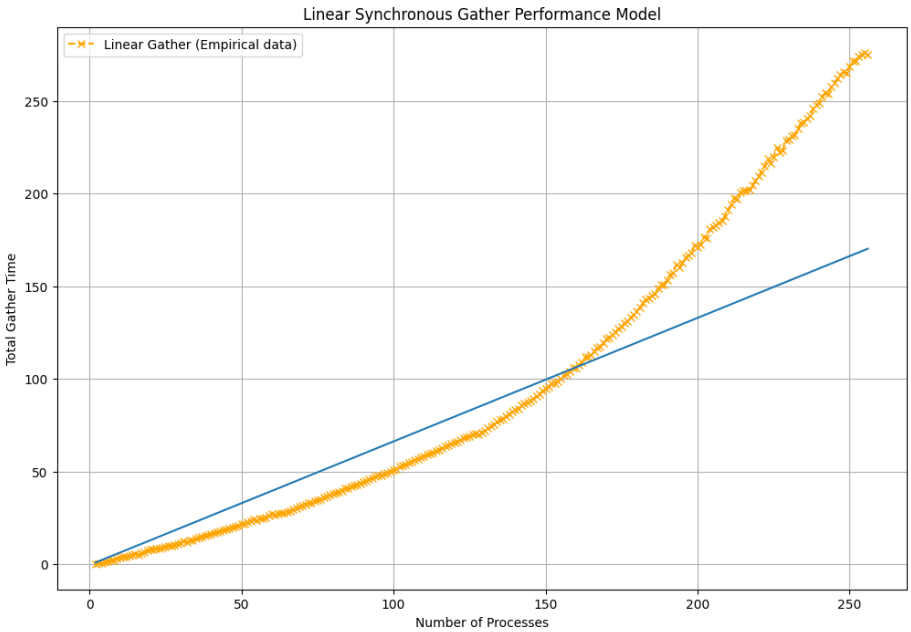

Figure 12: Performance model vs real model for gather-linear w/ sync