HPC

PERFORMANCE EVALUATION OF COLLECTIVE OPERATIONS

NIGAM VISHAL [SM3800014]

University of Trieste

1. INTRODUCTION

The OpenMPI library is a comprehensive implementation of the Message Passing Interface, supporting a wide array of algorithms for collective operations. These operations are critical in parallel computing, enabling efficient data exchange and synchronization among numerous processes. To evaluate the performance of these operations, we utilize the OSU Micro-Benchmark (OMB) suite, which is a well-regarded tool for benchmarking MPI implementations.

Computational Architecture

Our performance assessment employs the high-performance capabilities of the ORFEO cluster, specifically utilizing its EPYC partition. This partition consists of nodes powered by AMD EPYC 7H12 CPUs, each containing 64 cores. These processors are based on the "Rome" architecture, designed for robust parallel processing. The interconnect of the system is an infinity fabric, providing a theoretical bandwidth of 96 GB/s. Each EPYC processor is structured with eight Core Complex Dies (CCDs), and each CCD contains two Core Complexes (CCXs). Within each CCX, there are four cores and 16 MB of L3 cache, ensuring efficient processing and high-speed data access. This architecture is meticulously designed to support high-performance computing tasks, making it an ideal environment for our MPI benchmark evaluations.

OSU Micro-Benchmark

The OSU Micro-Benchmark suite includes tests for various MPI blocking collective operations such as Allgather, Alltoall, Allreduce, Barrier, Bcast, Gather, Reduce, Reduce-Scatter, and Scatter. These benchmarks measure the latency of MPI operations across multiple processes and message sizes over numerous iterations. Our experiments focus on the average latency for each message size to provide a detailed performance analysis.For this assignment two collective MPI operations have been taken into consideration –"**broadcast**" and "**scatter**".

This report investigates the performance of the broadcast and scatter operations, which are essential collective operations in MPI.

Broadcast Operation (MPI_Bcast)

The broadcast operation involves a root process sending a message to all other processes within a communicator. We evaluate the performance of four broadcast algorithms:

- **Default Algorithm**: This algorithm is typically optimized for the specific architecture of the system and involves the root process sending the message to all other processes using the most efficient available method, which may combine various strategies depending on the context.
- **Chain Algorithm**: The message is divided into segments and relayed from one process to the next in a pipeline until all processes have received the complete message.
- **Pipeline Algorithm**: Similar to the chain algorithm, but specifically optimized for dividing the message into segments that are passed along a pipeline, with each process forwarding segments as soon as they are received, thus optimizing the flow of data.

- **Binary Tree Algorithm**: Processes are structured as nodes in a binary tree, with the root process at the top. Each parent node splits the message and sends it to its child nodes, which merge the segments upon receipt.

Scatter Operation (MPI_Scatter): The scatter operation involves distributing different portions of a message from the root process to each participating process. We examine the performance of four scatter algorithms:

- **Default Algorithm:** The standard implementation provided by OpenMPI.
- **Basic Linear Algorithm:** The root process sequentially sends unique message segments to each process.
- **Binomial Algorithm:** Processes are organized in a binomial tree structure, where each node receives a segment and forwards the remaining segments to other nodes.
- **Non-Blocking Linear Algorithm:** This algorithm is similar to the basic linear algorithm but uses non-blocking communication to enhance performance.

Objectives

The primary aim of this study is to compare the latency performance of these algorithms for broadcast and scatter operations across fixed and different message sizes and numbers of processes. By analyzing the results from the OSU Micro-Benchmark suite, we intend to develop a predictive model for latency that incorporates the effects of both the number of processes and the message size.

In the following sections, we present a detailed analysis of the benchmark results, discussing the efficiency of each algorithm and the factors that influence their performance.

2. Experimental Framework

To evaluate the performance of broadcast and scatter operations on the ORFEO cluster, we utilized two EPYC nodes, totaling 256 cores. The OSU benchmark was employed for performance evaluations, with an increased number of iterations, particularly warm-up iterations, to achieve more stable results.

A Bash shell script created to manage resource allocation via slurm commands and it manages job submission to the cluster, specifying resources like the number of nodes and tasks per node. The script iterates through different algorithms and message sizes, running the benchmarks and capturing performance metrics such as average latency. It ensures the results are consistent by using warm-up iterations and formats the output into a CSV file for further analysis.

Choice of mapping-“-map by”-

In Open MPI, the `map-by` option in job launch commands specifies how processes are distributed across hardware resources (e.g., cores, sockets, nodes). To find the best `map-by` setup for our experiments, we ran several latency tests.**Table 1**

Binding processes to specific cores significantly reduced latency, especially for small message sizes. However, as message size increased, the impact of pure latency decreased. Larger processor counts also showed increased latency, even with small message sizes.

Algorithm choice significantly affected the optimal `map-by` strategy. For instance, the Basic Linear algorithm's performance dropped when using `map-by socket` instead of `map-by core`, whereas the Binomial algorithm was less impacted. This was due to higher latency from distant communication paths in the `map-by socket` setup.

Table 1: Latency vs. `map-by` comparison for Scatter operation. 128 cores per Node (EPYC)

Test Type	Message Size	Avg Latency (us)
Map by socket - Basic Linear Algorithm	1	14.30
Map by core - Basic Linear Algorithm	1	5.70
Map by socket - Binomial Algorithm	1	5.20
Map by core - Binomial Algorithm	1	3.80

3. Results

After the slurm batch run in several iterations, we collected benchmark data of broadcast and scatter operations and examined various parameters to assess their performance. The tests consist of a range of core configurations, message sizes, and visualization techniques.

Test Parameters: Cores varied from 1 to 128 per node, in total 256 cores. Message sizes ranged from *2^0 to 2^19* bytes.

First (Figure 2, 3) a 3D heatmap for both of the collective operations and its algorithms is reported, showcasing latency vs message size and the number of processes. The 3D heatmaps reveal that the binary tree and non-blocking linear algorithms are the most efficient for broadcast and scatter operations, respectively. The binary tree algorithm demonstrates the lowest latency across varying message sizes and process counts, highlighting its effectiveness in hierarchical communication patterns. Similarly, the non-blocking linear algorithm excels in scatter operations, maintaining minimal latency even as the number of processes and message sizes increase.

In contrast, the chain algorithm and default algorithms exhibit higher latencies, particularly as message sizes grow, due to their less efficient data transfer methods. The pipeline algorithm performs better than the chain algorithm for broadcast operations by utilizing pipelining to reduce overall latency. Overall, these findings emphasize the importance of selecting algorithms tailored to specific communication patterns and workloads to achieve optimal performance in parallel computing environments.

4. Performance models

When considering both the number of processes and the message size as factors influencing latency, the generalized linear model include both variables to have better understanding of both operations. The The baseline model determined by the collected data is the following model which can be represented in the form:

$$\text{Latency}=\text{Intercept}+\beta1\times\text{Number of Processes}+\beta2\times\text{Message Size}$$

Where:

- Intercept is the model intercept.
- β1 is the coefficient for the number of processes.
- β2 is the coefficient for the message size.

To get to this model, we had to include both the number of processes and the message size in the regression analysis.

Equations for Each Model:

1. **Broadcast Fixed:**
   $\text{Latency}=-0.765883058562979+0.12053964\times\text{Number of Processes}+0\times\text{Message Size}$
   Simplified to $\text{Latency}=-0.765883058562979+0.12053964\times\text{Number of Processes}$

2. **Broadcast Full:**
   $\text{Latency}=-111.79086513468317+1.89627182\times\text{Number of Processes}+0.0023354\times\text{Message Size}$

3. **Scatter Fixed:**
   $\text{Latency}=0.6290812623031581+0.09523499\times\text{Number of Processes}+0\times\text{Message Size}$

   Simplified to: $\text{Latency}=0.6290812623031581+0.09523499\times\text{Number of Processes}$

4. **Scatter Full:**
   Latency=−443.3362849494679+6.39324967×Number of Processes+0.00711588×Message Size

Generalized Equation:

Considering the coefficients from all models, a generalized equation can be written as:

Latency=Intercept+β1×Number of Processes+β2×Message Size

Where:

- Intercept ranges from approximately −443.34 to 0.63.
- β1 ranges from approximately 0.095 to 6.39.
- β2 ranges from approximately 0.0023 to 0.0071.

This model allows us to understand the behaviour of the broadcast operation with pipeline algorithm and mapping of the processes by core. **Figure 1.**The actual latency generally stays low initially and gradually increases as the number of processes increases. The naive model prediction follows a similar trend [**Figure 4**] but slightly underestimates the latency as the number of processes increases. The pipeline algorithm exhibits the highest latencies, suggesting it might not be as efficient for larger numbers of processes. The binary tree algorithm maintains lower latencies, indicating better performance and scalability[**Figure 5,6**]. Across all plots, the naive models provide a reasonable approximation of the latency trends but fail to capture specific performance anomalies, such as sudden spikes or dips in latency. These deviations highlight the importance of considering non-linear effects, such as network contention, process scheduling, and algorithmic overhead, when modeling the performance of collective operations. Additionally, the choice of algorithm has a significant impact on scalability and efficiency, underscoring the need for careful algorithm selection and optimization based on the specific characteristics of the communication pattern and hardware architecture.

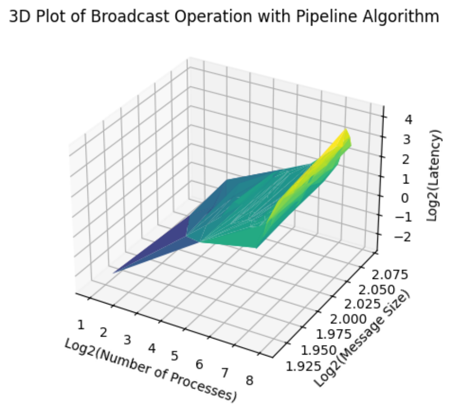

Figure 1: *3D plot of the broadcast operation with pipeline algorithm and mapping of the processes by core. The plot shows the relationship between the number of processes, the log of message size and of latency*

5. Conclusion

In conclusion, our study of broadcast and scatter operations on the ORFEO cluster shows the complex relationship between communication patterns, hardware setups, and algorithm choices in OpenMPI. The experiments and models revealed intricate latency behaviors, highlighting the shortcomings of simple or naive performance models. It is essential to consider factors like cluster layout, binding regions, communication latency differences, and scalability to create accurate performance model.

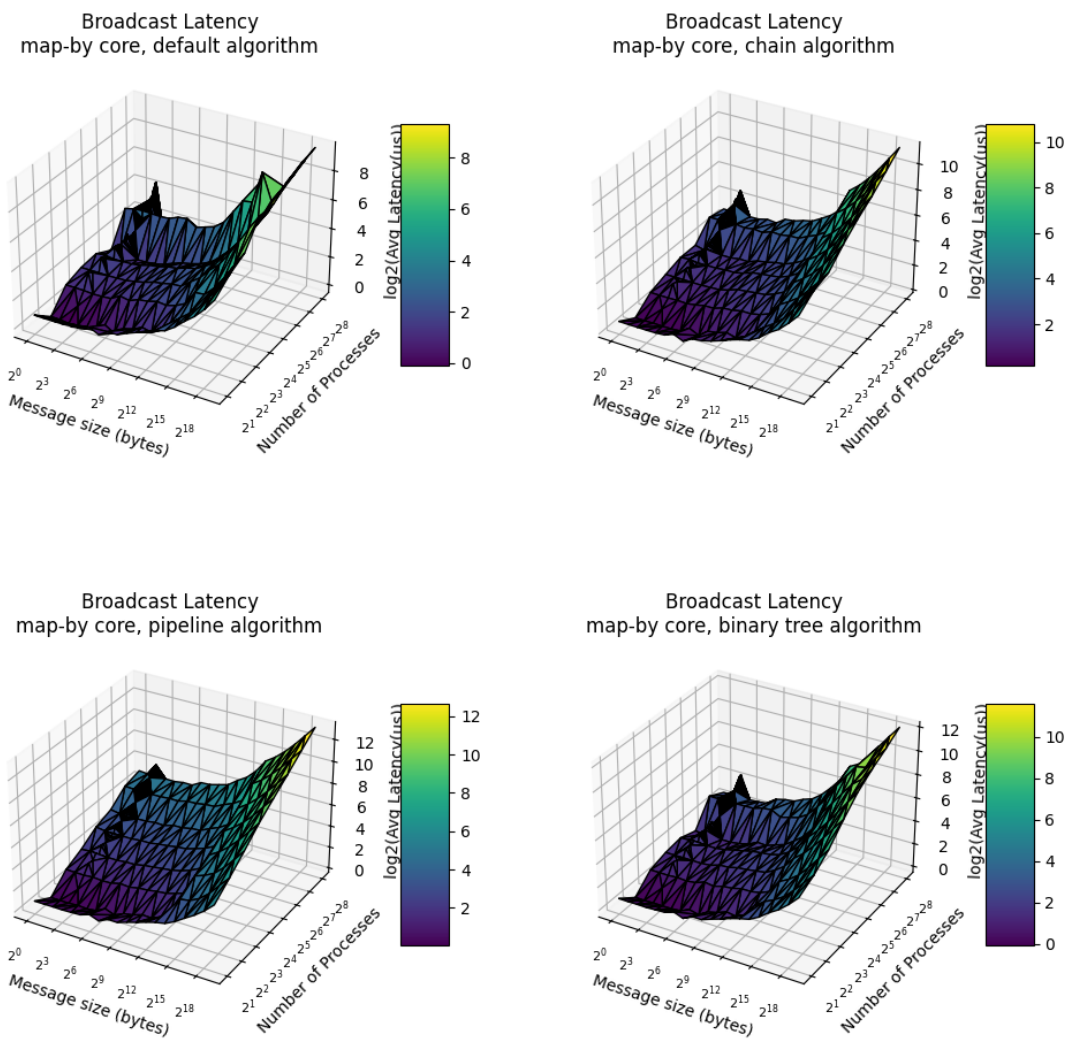

Figure 2: 3D Heatmap for all 4 algorithms of Broadcast operation

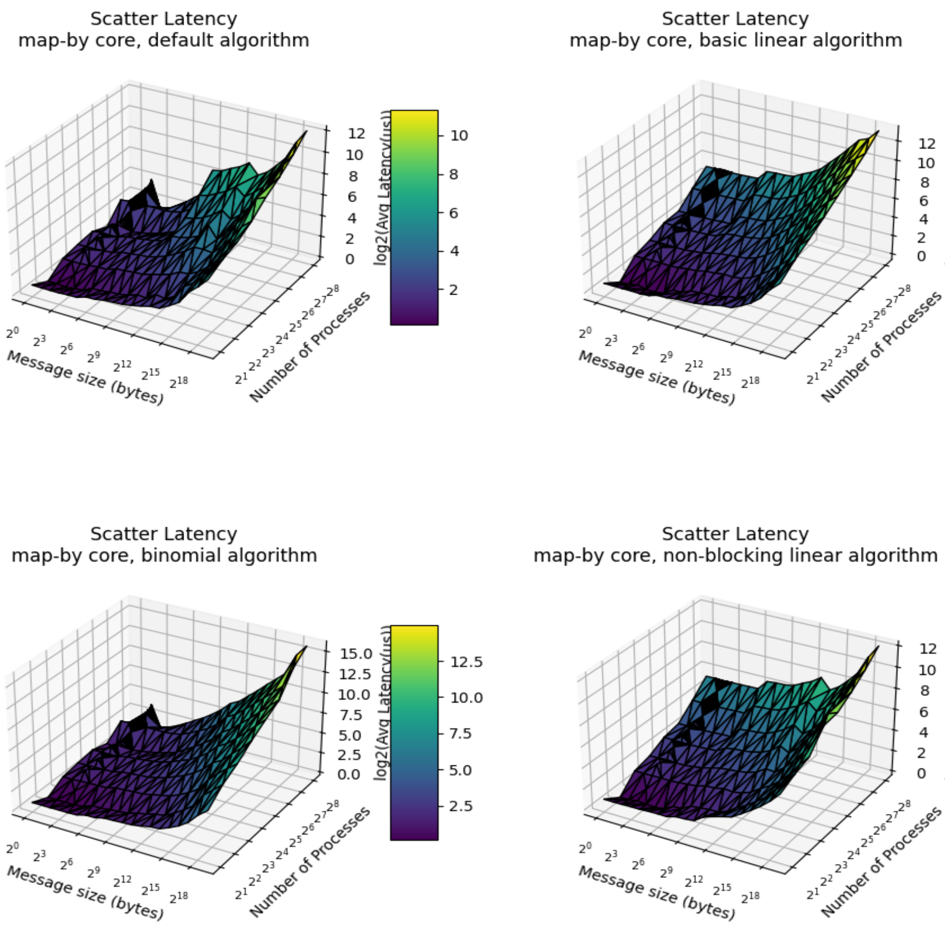

Figure 3: 3D Heatmap for all 4 algorithms of Scatter operation

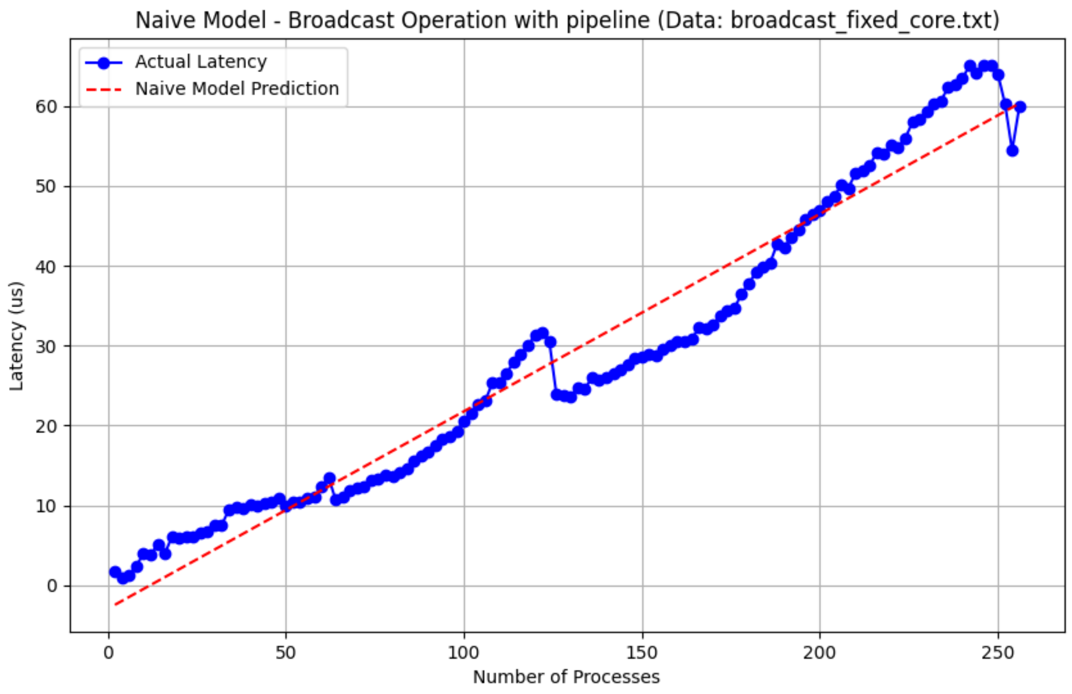

Figure 4: LineMap for Naïve Model Performance for Fixed Size (4)

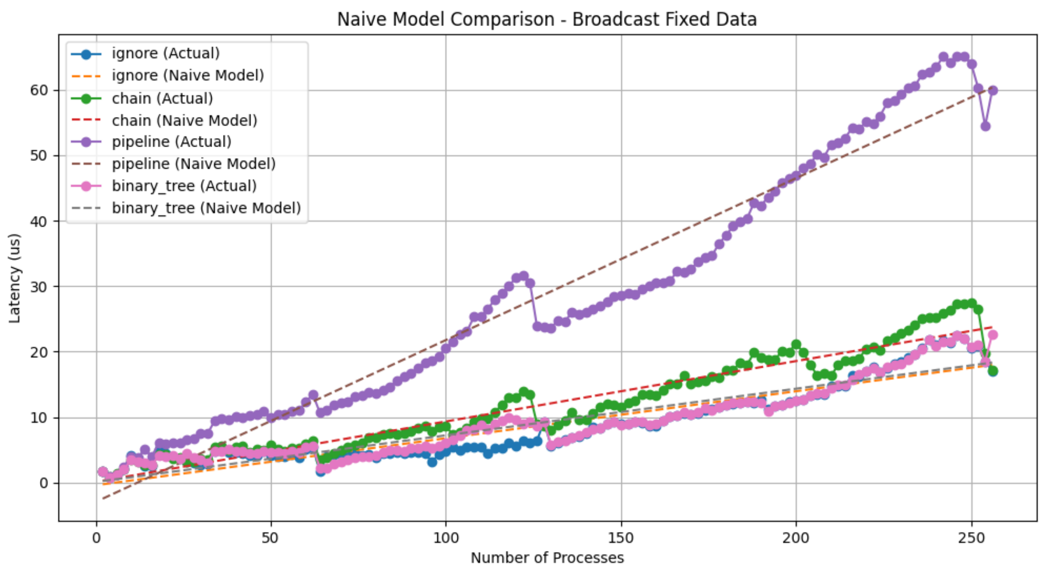

Figure 5: Plot of latency of different Broadcast algorithms for fixed message size and Performance against naïve model.

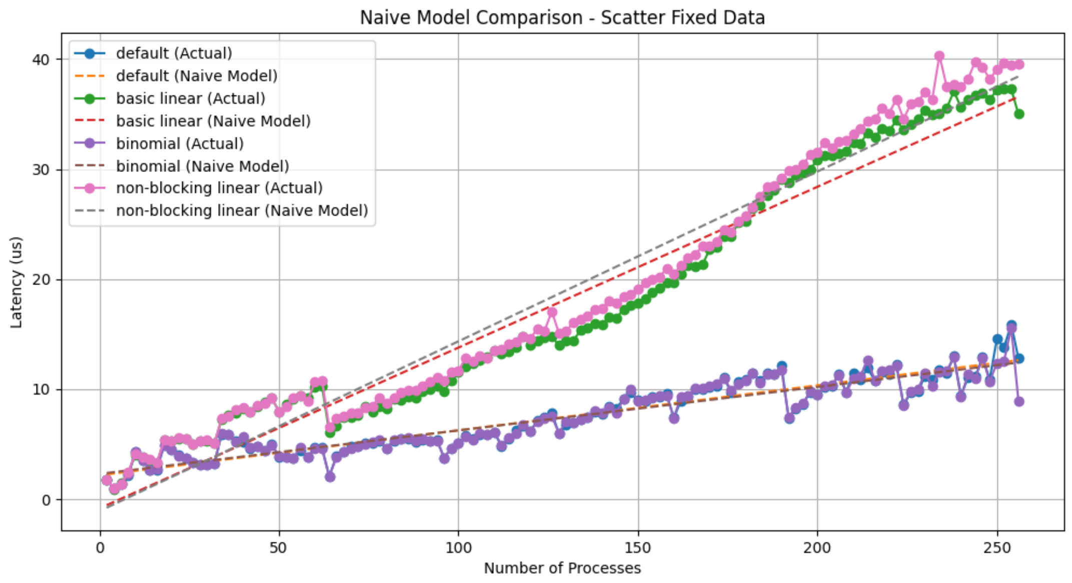

Figure 6: Plot of latency of different Scatter algorithms for fixed message size and Performance against naïve model.