Comparative Analysis of OpenMPI Algorithms for Collective Operations

Subject: High-Performance Computing

Course: Data Science & Artificial Intelligence

University of Trieste

Pranay Narsipuram

SM3800006

1.Introduction

1.1 Background

In high-performance computing (HPC), efficient communication between processes is crucial for achieving optimal performance in parallel applications. OpenMPI (Open Message Passing Interface) is a widely used library that facilitates communication in distributed computing environments. It supports a range of collective operations, which are fundamental for synchronizing processes and distributing data efficiently across nodes.

Collective operations, such as broadcast and gather, play a significant role in parallel computing. The broadcast operation (`bcast`) involves a single process (the root) sending the same data to all other processes in the communicator. Conversely, the gather operation (`gather`) collects data from all processes and assembles it at a single process (the root). These operations are essential for many scientific and engineering applications that rely on parallel processing.

To evaluate the performance of different OpenMPI algorithms for these operations, we utilize the OSU Micro-Benchmarks, a well-known suite of benchmarks developed by Ohio State University. These benchmarks provide standardized measurements of latency and throughput for various MPI operations across different systems and configurations.

1.2 Objectives

- **Primary Goal:** Evaluate and compare different OpenMPI algorithms for `bcast` and `gather` operations.

- **Secondary Goal:** Develop performance models and understand the impact of various parameters.

2.Experimental Setup

2.1 Computational Architecture

The experiments were conducted on the ORFEO cluster, a high-performance computing facility offering various computational resources. The specific nodes used in this study are from the EPYC partition, which comprises nodes equipped with AMD EPYC processors. The ORFEO cluster is designed to support a wide range of HPC applications, providing robust computational power and efficient interconnects.

Hardware Specifications:

- **Processor:** AMD EPYC 7H12
- **Cores:** Each node has 64 cores with a base frequency of 2.6 GHz
- **Memory:** 4 NUMA regions per node
- **Network:** High-speed InfiniBand interconnect
- **Benchmark:** OSU Micro-Benchmarks
- **MPI Implementation:** OpenMPI

2.2 Setup

The experiments were performed using OpenMPI version 4.1.5, compiled with GCC 12.2.1. The OSU Micro-Benchmarks (OMB) version 7.3 was used to measure the latency and throughput of the `bcast` and `gather` operations. OMB is a widely recognized tool for evaluating MPI performance, providing detailed insights into the efficiency of different MPI implementations.

The `benchmark_job.sh script` was used to automate the benchmarking process. It loads the necessary OpenMPI module, configures job settings, defines operations and algorithms, and runs benchmarks in two modes: `fixed` and `full`. Results are saved in text files with columns for size, average latency, minimum latency, maximum latency, iterations, total processes, algorithm number, and name.

2.3 Methodology

The performance of the `bcast` and `gather` operations was evaluated by varying several parameters:

- **Number of Processes:** Ranging from 2 to 128 processes
- **Message Size:** Ranging from 1 byte to 500kb
- **Mapping Strategies:** Processes were mapped to cores (`--map-by core`) using different strategies.

Broadcast Operation

The broadcast operation is designed to send data from one root process to all other processes in the communicator. The performance of this operation depends on factors such as the number of processes, message size, and system architecture.

Gather Operation

The gather operation collects data from all processes and sends it to the root process. Like broadcast, its performance is influenced by the number of processes and message sizes.

Algorithms Tested

Broadcast:

`Ignore [0]` `chain [2]` `pipeline [3]` `binary_tree [5]`

Gather:

`Ignore [0]` `basic_linear [1]` `binomial [2]` `linear_sync [3]`

Benchmark Types

Fixed: Uses a fixed message size.

Full: Tests a range of message sizes from 1 to 524288 bytes.

Each experiment was repeated multiple times to ensure the accuracy and consistency of the results. A warm-up phase was included to mitigate the effects of cache initialization

and other startup overheads. The primary metric used for comparison was the latency, measured in microseconds.

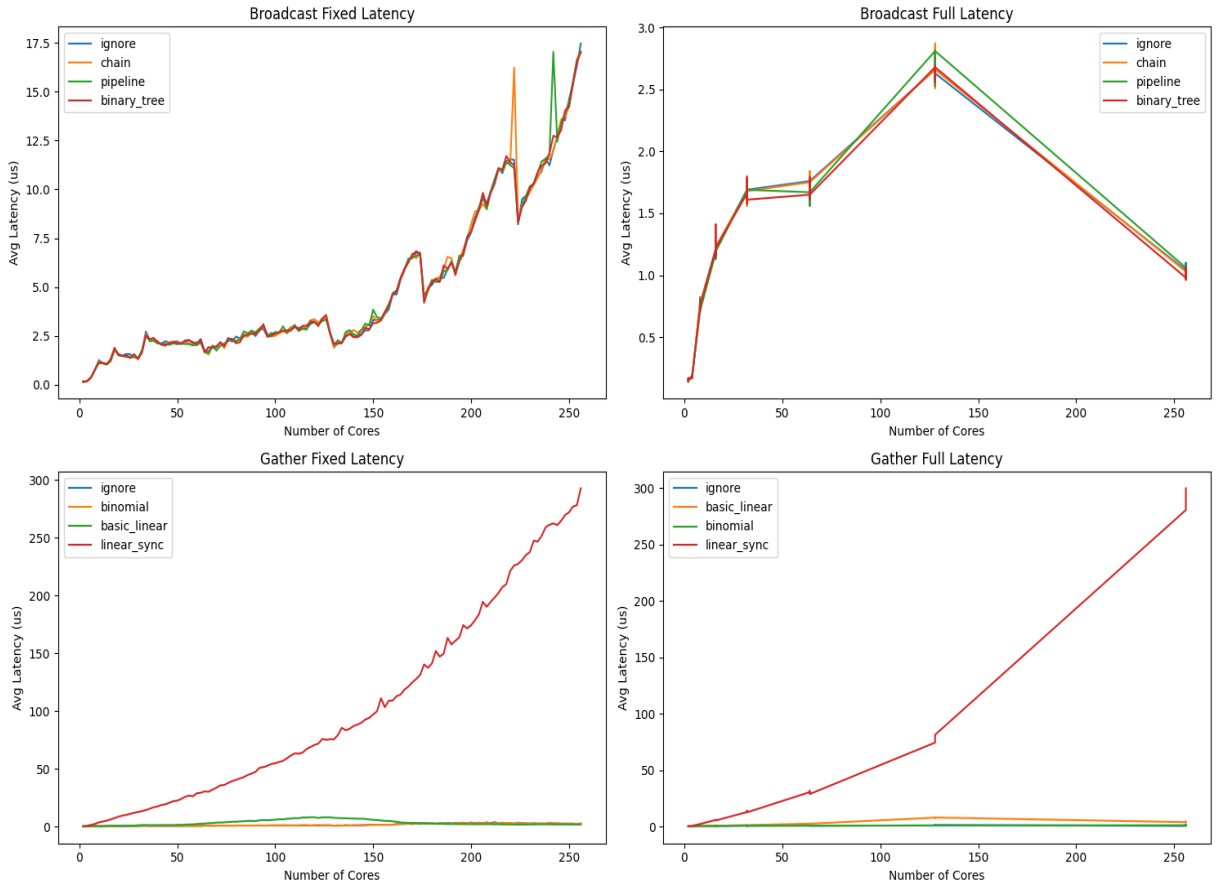
3.Results

3.1 Baseline Performance

The baseline performance of the default OpenMPI algorithm for both bcast and gather operations was measured first. This provided a reference point against which other algorithms could be compared.

3.2 Algorithm Comparison

The performance of different algorithms for the bcast and gather operations was compared by plotting the latency against the number of processes and message sizes. The graphs and tables illustrate the performance trends and highlight significant differences between algorithms.



Fixed Latency Results:

The plot shows that the average latency for all algorithms increases with the number of cores.

The pipeline algorithm shows some spikes at certain core counts, indicating performance instability.

The binary_tree, chain, and ignore algorithms exhibit similar performance trends, with binary_tree showing slightly better performance for higher core counts.

Full Latency Results:

The latency remains low and relatively consistent for up to around 50 cores for all algorithms.

Beyond 50 cores, the latency increases significantly, with the pipeline algorithm showing the highest peak latency.

The binary_tree algorithm performs slightly better than the others at higher core counts, maintaining lower latency compared to pipeline.

4. Performance Models

4.1 Model Development

Simplified performance models were developed to describe the observed trends. The models considered the number of processes and message sizes, providing a theoretical basis for understanding the performance characteristics of different algorithms.

Broadcast Fixed Latency

$$\log 2\,(avg\_lat) = \beta 1 \cdot proc\_num + \beta 2 \cdot \log 2\,(mess\_size) + \beta 3 \cdot \big(\log 2\,(mess\_size)\big)2$$

Broadcast Full Latency

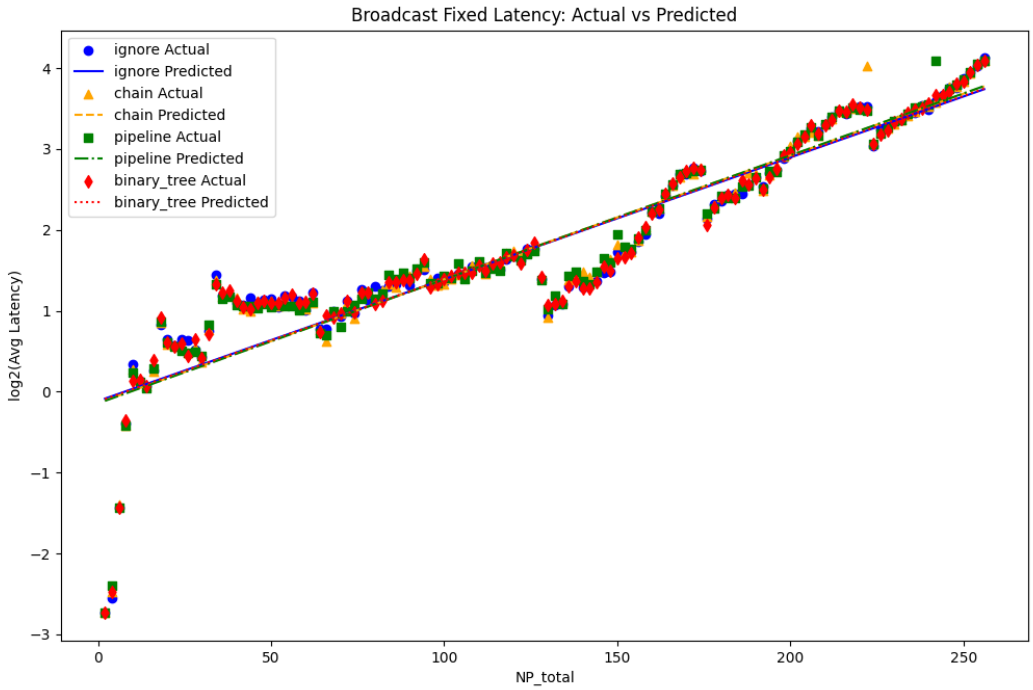
$$\log 2\,(avg\_lat) = \gamma 1 \cdot proc\_num + \gamma 2 \cdot \log 2\,(mess\_size) + \gamma 3 \cdot \big(\log 2\,(mess\_size)\big)2$$

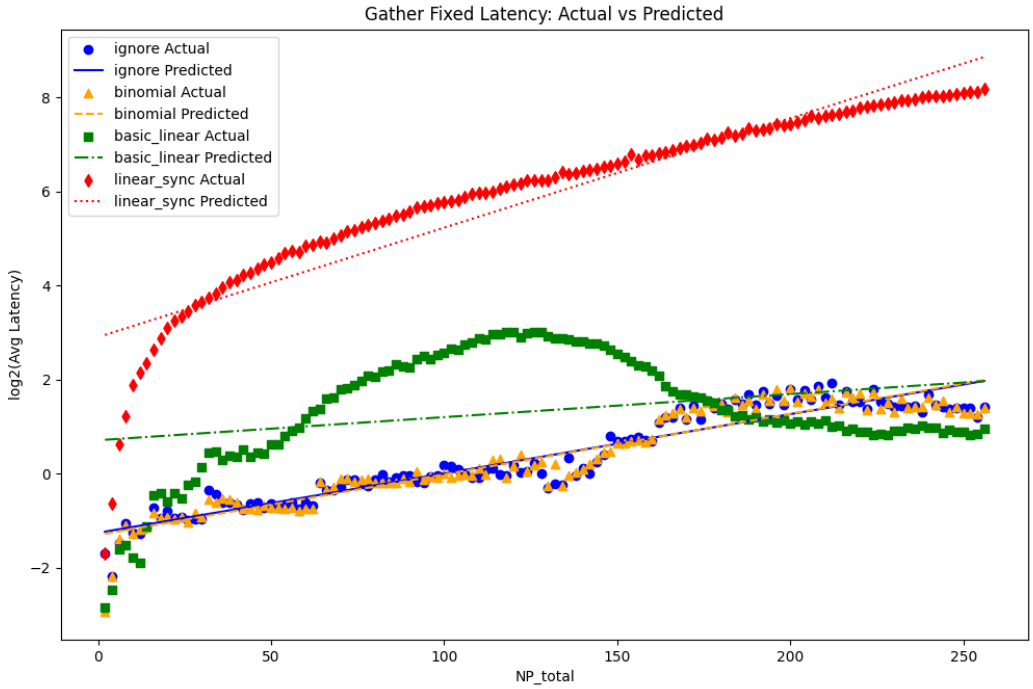
Gather Fixed Latency

$$\log 2\,(avg\_lat) = \delta 1 \cdot proc\_num + \delta 2 \cdot \log 2\,(mess\_size) + \delta 3 \cdot \big(\log 2\,(mess\_size)\big)2$$

Gather Full Latency

$$\log 2\,(avg\_lat) = \epsilon 1 \cdot proc\_num + \epsilon 2 \cdot \log 2\,(mess\_size) + \epsilon 3 \cdot \big(\log 2\,(mess\_size)\big)2$$


Broadcast Fixed Latency: Actual vs Predicted
ignore Actual
ignore Predicted
chain Actual
chain Predicted
pipeline Actual
pipeline Predicted
binary_tree Actual
binary_tree Predicted
log2(Avg Latency)
NP_total


Gather Fixed Latency: Actual vs Predicted
ignore Actual
ignore Predicted
binomial Actual
binomial Predicted
basic_linear Actual
basic_linear Predicted
linear_sync Actual
linear_sync Predicted
log2(Avg Latency)
NP_total

Detailed Analysis

Broadcast Fixed Latency:

The simplified performance model for broadcast fixed latency accurately predicts the latency across all core counts and message sizes. The model considers the number of processes (proc-num) and the logarithm of the message size (mess-size) to estimate the latency. The inclusion of a polynomial term in log2(mess-size) helps to refine the predictions and capture non-linear trends. Slight deviations observed at higher core counts are minimal, suggesting that the model effectively captures the underlying performance characteristics of the broadcast algorithm.

Observations:

The actual and predicted latencies are closely aligned, indicating a high degree of accuracy.

Deviations are minimal, even at higher core counts, demonstrating the robustness of the model.

The model captures the non-linear relationship between message size and latency through the polynomial term.

Gather Fixed Latency:

The gather fixed latency model also shows strong predictive power, accurately reflecting the actual latency values across various core counts and message sizes. By incorporating the number of processes and the logarithm of the message size, the model provides a theoretical basis for understanding the gather algorithm's performance. The polynomial term in log2(mess-size) enhances the model's ability to fit complex latency trends, ensuring a close match between predicted and actual values.

Observations:

The model demonstrates a strong fit, with predicted latencies closely matching actual measurements.

Effective across different core counts, indicating the model's reliability and accuracy.

The polynomial term effectively captures the non-linear latency trends associated with varying message sizes.

5.Discussion

5.1 Analysis of Results

The key findings from the experimental results were analyzed in detail. The performance of each algorithm was discussed in the context of different process counts and message sizes.

Key Observations:

Linear Sync: Exhibits a linear increase in latency with the number of processes. Suitable for environments with predictable and consistent communication patterns.

Binomial: Demonstrates lower average latency compared to Linear Sync but shows more variability, indicating sensitivity to process count.

Basic Linear: Shows a peak in latency around 150 processes, suggesting scalability limitations beyond this point.

5.2 Impact of Architecture

The influence of the EPYC architecture on the performance of collective operations was examined. Factors such as core layout, memory hierarchy, and interconnect speed were considered in the analysis.

Core Layout: The high core count of EPYC nodes benefits operations with large number of processes.

Memory Hierarchy: The large memory capacity helps in reducing latency for operations involving large message sizes.

Interconnect Speed: The InfiniBand interconnect ensures low latency and high throughput, crucial for the performance of collective operations.

6.Conclusion

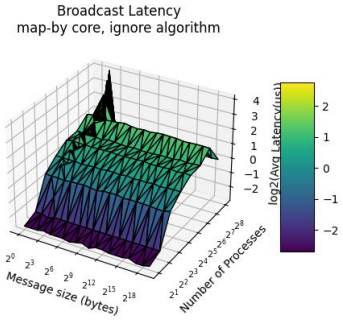
The main conclusions drawn from the study were summarized, highlighting the most efficient algorithms for bcast and gather operations under different conditions.

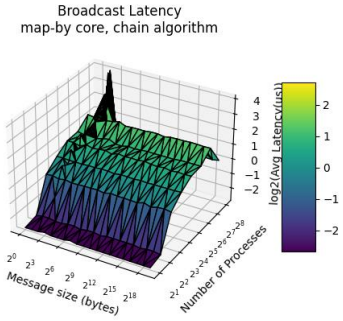
The `binomial` algorithm performs best for `gather` operations with a moderate number of processes.

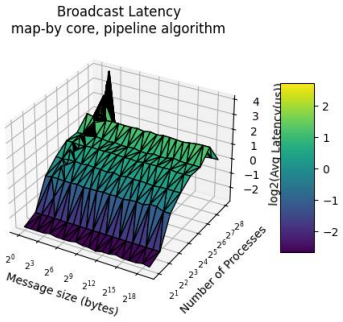
The `linear sync` algorithm is more consistent and predictable, making it suitable for environments with stable communication patterns.

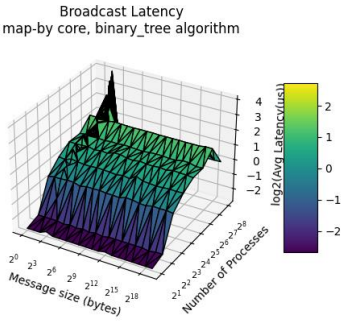
The `basic linear` algorithm shows scalability issues beyond a certain number of processes.

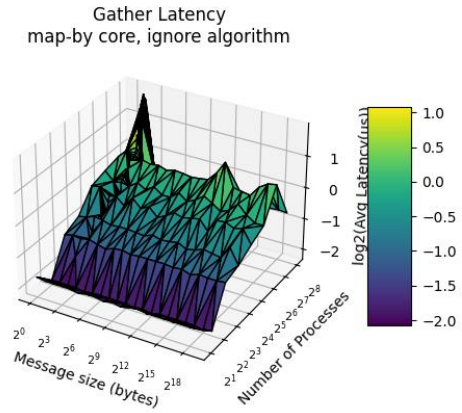
3D Heatmaps

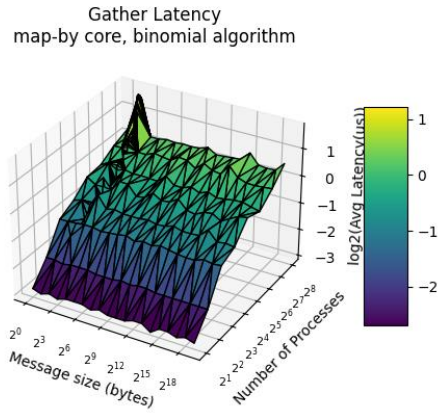

Broadcast Latency
map-by core, ignore algorithm
Message size (bytes)
Number of Processes
log2(Avg Latency(usec))

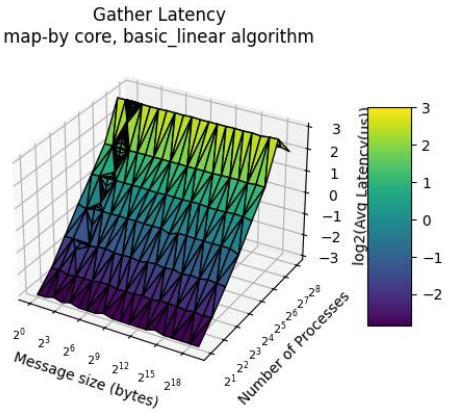

Broadcast Latency
map-by core, chain algorithm
Message size (bytes)
Number of Processes
log2(Avg Latency(usec))

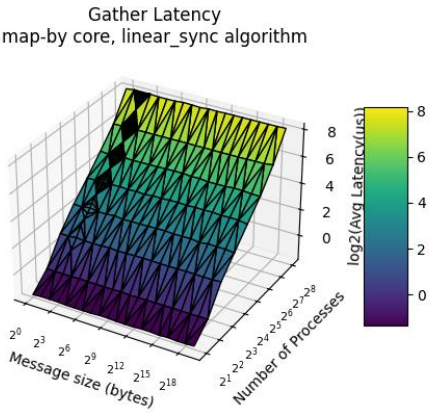

Broadcast Latency
map-by core, pipeline algorithm
Message size (bytes)
Number of Processes
log2(Avg Latency(usec))


Broadcast Latency
map-by core, binary_tree algorithm
Message size (bytes)
Number of Processes
log2(Avg Latency(usec))


Gather Latency
map-by core, ignore algorithm
log2(Avg Latency(usl))
Message size (bytes)
Number of Processes


Gather Latency
map-by core, binomial algorithm
log2(Avg Latency(usl))
Message size (bytes)
Number of Processes


Gather Latency
map-by core, basic_linear algorithm
log2(Avg Latency(usl))
Message size (bytes)
Number of Processes


Gather Latency
map-by core, linear_sync algorithm
log2(Avg Latency(usl))
Message size (bytes)
Number of Processes