High-Performance Computing - Exercise 1

Comparative Analysis of OpenMPI Algorithms for Collective Operations using OSU benchmark

Andela Cvetinovic [SM3800009]
Data Science and Artificial Intelligence [SM38-23]
Supervised professor: Stefano Cozzini

Abstract

This project aims to evaluate the latency of the OpenMPI implementation for two collective operations, broadcast and reduce by varying the number of processes, message sizes, and distribution algorithms.

1. Introduction

In high-performance computing, it is essential to understand the efficiency and performance of parallel processing libraries. This project aims to explore the collective operations of OpenMPI, examining its various algorithms and their interaction with the hardware architecture, specifically ORFEO. OSU benchmarks have been used to evaluate the performance of two selected blocking operations - broadcasts and reduction.

1.1. Collective operations

Collective communication routines are predefined operations in parallel computing that involve communication among multiple processes within a group. Unlike point-to-point communication, where data is transferred between two specific processes, collective communication involves data exchange among all processes in a group, typically involving synchronization. These routines are essential in HPC for efficient data distribution, collection, and synchronization. Parallel computing utilizes multiple computing resources simultaneously—such as multiple CPUs, cores, or distributed systems—to solve computational problems. Effective communication between these resources is vital for the efficient execution of parallel programs and to prevent conflicts between operations. The following collective operations are used to keep compute resources supplied with data and instructions and to maintain synchronization:

- **Broadcast**
  A single process transmits the same data to all other processes. This operation is used usually for distributing configuration data, or input parameters when the root process has a block of data that needs to be entirely copied to all the other processes.
- **Barrier**
  Ensures that all processes reach a specific point in the program before any can proceed. It is critical for synchronizing phases of computation, preventing any process from advancing too far ahead or lagging behind, which can lead to inefficient use of resources and potential errors in the results.
- **Gather**
  Collects data from multiple processes and assembles it into a single process. It is used when partial results from various processes need to be combined into one dataset. For example, if each process is computing

a segment of a larger problem, the gather operation will collect these segments into one process to form a complete result.

- **Scatter**
  Distributes data from one process to multiple processes. It is used when a single dataset needs to be divided and sent to different processes for parallel processing. For example, if one process holds the entire dataset, the scatter operation will send portions of this data to other processes so that each can work on its part independently.
  While gather and scatter perform opposite functions, they are frequently used together in workflows that involve both distributing initial data to processes (scatter) and then collecting the results back from those processes (gather).
- **Reduce**
  Combines data from all processes into a single result using a specified operation (such as sum, maximum, or logical AND). It is important for aggregating results, such as summing values from multiple processes to obtain a total, finding the maximum value, or performing other collective computations that consolidate data from all processes into one final outcome.

1.2. OpenMPI

Open Message Passing Interface is a high-performance, open-source implementation of the MPI standard. OpenMPI offers a versatile and robust framework for creating and running parallel applications, making it an essential tool in HPC. Key features include scalability, portability, and modularity.

1.3. Distribution Algorithms

Of several algorithms provided by OpenMPI for distributing messages among processes, the chosen ones are linear, chain, and tree for both operations.

- **Basic_Linear**
  The simplest one, where the responsibility of sending the message to all other processes rests solely with one process. The message is sent in a linear sequence: the root process begins by sending the message to the first process, then proceeds to send it to the second process, and continues this pattern until all processes have received the message. The total data transmitted amounts to $(n-1) \cdot \mathrm{m}$, where n is the number of processes and m is the message size.
- **Chain**
  The chain algorithm improves upon the basic linear by having each process send the message to its immediate neighbor in sequence, starting from the root process. Instead of the root sending messages to all processes, it sends them to the next process, which then forwards it to the next, and so forth, amounting to total data transmitted to $n \cdot \mathrm{m}$. This ensures that each node is actively sending data to another node at any given time. Consequently, the root process can begin working on its own data immediately after sending the message to the first process.
- **Binary_tree**
  In this algorithm, the total data is divided into pieces, each designated for a specific node. The root process sends the initial chunks to two nodes, which then forward them to their respective children, continuing this pattern until all nodes receive the data. This results in a total of log_2(n) messages. The total data transmitted is log_2(n) $\cdot\ n \cdot \mathrm{m}$ /2. A drawback is the redundancy in data transfer due to overlapping communication paths, potentially causing network bandwidth issues as the number of processes increases. For broadcast operation, redundancy is minimized since each process requires the same data.

2. Methods

2.1. Node selection

The assignment requires using two identical nodes to evaluate selected collective operations with varying parameters. The ORFEO cluster provides THIN, FAT, and EPYC partitions and architectural differences between them significantly impact how the problem is approached. The THIN partition was selected due to its simpler architecture, i.e., more efficient memory access lower latency within the node, and high availability. THIN is formed by 10 nodes, each with 2 Intel Xeon Gold 6126 processors and 768 GB of RAM. The nodes are interconnected with a 100 Gbit HDR InfiniBand network, which allows fast communication between the nodes with low latency. Each node in THIN has 12 cores and one NUMA node per CPU. This single NUMA per CPU means that each CPU has direct access to its own local memory, reducing latency for memory access. When a process on a CPU accesses its local memory, it is faster compared to accessing memory from another CPU (which would require additional steps and higher latency).

2.2. Benchmark Execution & Performance Metrics Collection

The tool used to measure the performance and compare different OpenMPI algorithms for broadcast and barrier operations was the OSU benchmark. Processes have been mapped by the core to leverage the THIN node architecture (1 NUMA node per CPU). This maximizes computational resources and isolates processes, ensuring consistent results and simplified analysis. This approach guarantees that the benchmarking results accurately reflect the true performance of the algorithms under test. Then, performance metrics run from benchmarks were collected. Bash scripts of the following test parameters were submitted as jobs:

Test Parameters	
# Warmup iterations	1000
# Iterations	10000
Message size	1 byte – 1 MB
# Processes	2 - 48
Algorithms	Basic linear, chain, binary tree

2.3. Performance Metrics Analysis

To analyze the data, a Python notebook was created to process the output files, load the information, and plot it. Basic mathematical models were developed to understand the behavior of each distribution algorithm.

3. Results

The primary goal is to evaluate the performance of MPI operations broadcast and barrier on ORFEO nodes and understand how latency scales with message size.

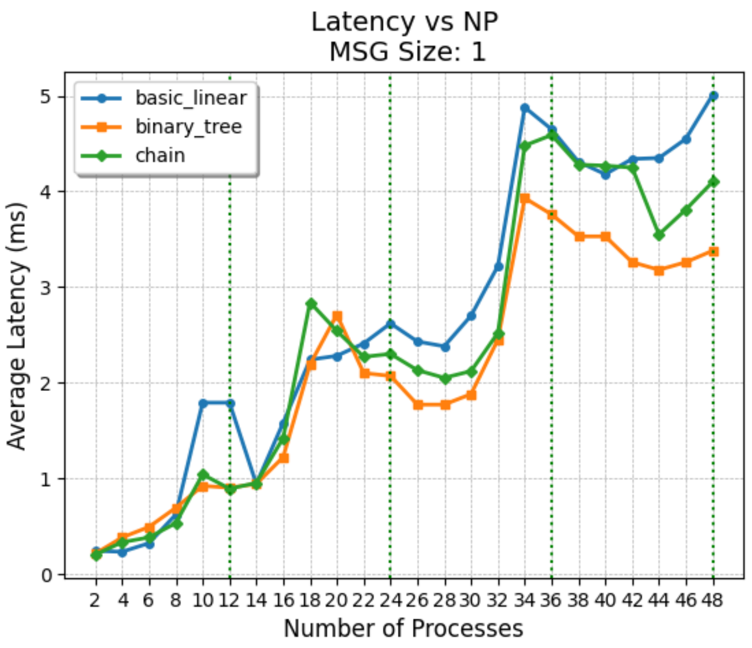
3.1. Broadcast

The results for this are reported considering two different message sizes: 1 byte, and 1 Mb. What could have been expected before performing experiments is a significant increase in latency when the MPI processes fill the socket and even more so when they fill the node. This increase occurs because communication between processes on different sockets or nodes must be conducted through the network, which is slower than using shared memory and involves additional steps and overhead, leading to higher latency compared to the faster, direct access of shared memory within a single socket.

3.1.1. Broadcast: Small message (1 byte)

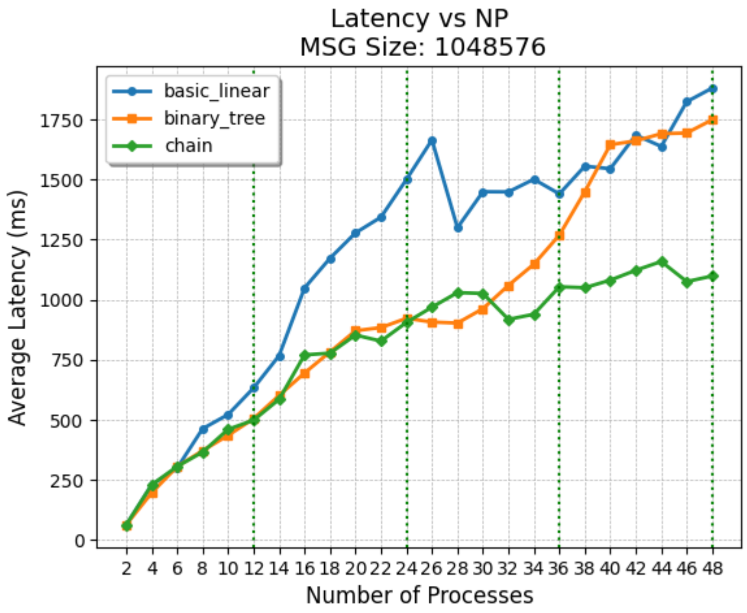
The latency for all three algorithms starts low but increases as the number of processes increases. The binary algorithm performs the best overall. Due to the small message size, the differences between the algorithms are not obviously notable but will become more noticeable when a larger message is tested.

As long as processes are within the same CPU (socket), they can share cache, leading to lower latency. However, once all cores within the first CPU are utilized and processes start to run on the second CPU, inter-CPU communication (between two CPUs) requires maintaining cache coherence, which results in a rapid increase in latency observed as the number of processes grows from 12 to 24 (since there are 12 cores per CPU). After that, the positive impact of the cache again helps reduce latency until the second CPU is fully utilized. When processes start to run on a different node, latency drastically increases again due to the slower network communication. Once more, the cache of the second node helps reduce latency (for processes 36-48). When the processes no longer fit the first node, basic linear and chain exhibit a steady increase in latency with a noticeable jump (around 24 processes) and continue to increase more sharply.



3.1.2. Broadcast: Large message (1 MB)

Basic linear performs even worse with larger message sizes, indicating inefficiencies in handling large data in a linear broadcast fashion, while both binary tree and chain algorithms handle the larger message size more efficiently. However, the chain algorithm demonstrates the best scalability so would be recommended for highly scalable systems. The overhead is more noticeable for smaller messages because it takes up a larger share of the total time, overshadowing the actual message transmission. This is especially clear in the first plot, where the smaller message size amplifies the impact of the overhead compared to this plot for larger message.

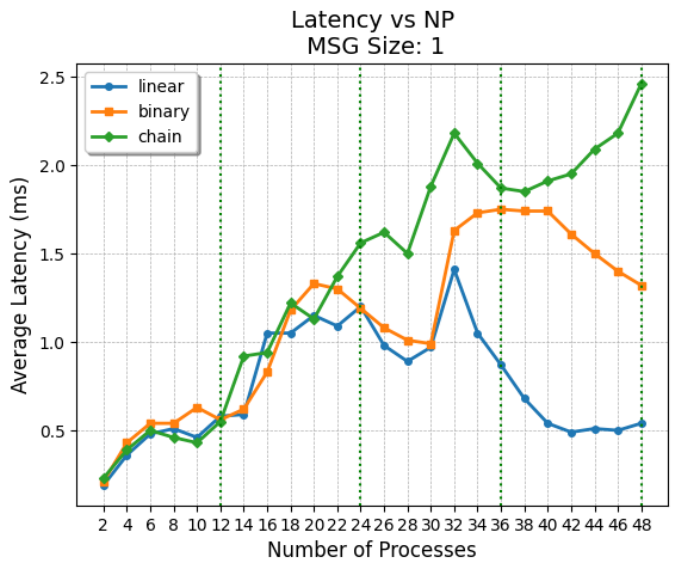


3.2. Reduce

The analysis is performed on the same message sizes 1 byte and 1 Mb, as well as the expected increase in latency for default MPI_reduce which is summation.

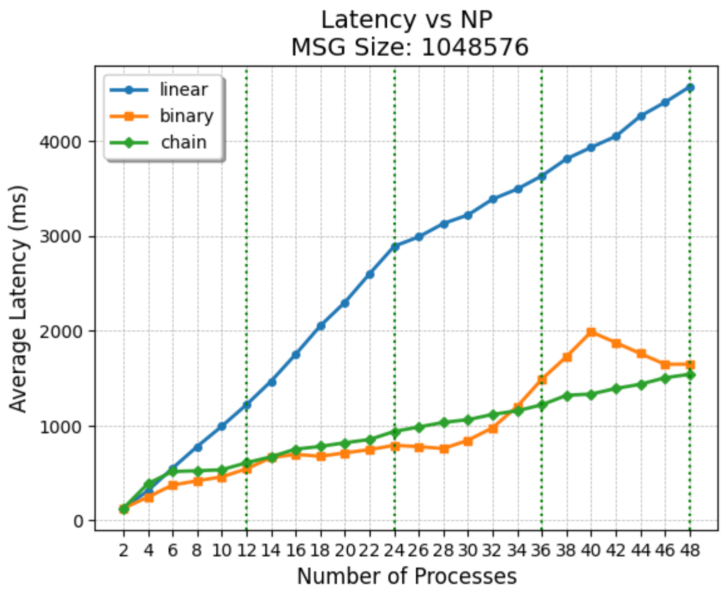
3.2.1. Reduce: Small message (1 byte)

Again as expected, the latency for all three algorithms starts low but increases as the number of processes increases. However, for small messages, the linear algorithm outperforms the others. This is because it aggregates all the data in one go, without breaking it into smaller chunks. In contrast, the chain and binary tree algorithms divide the data into smaller pieces and perform multiple aggregation steps, which adds overhead and increases latency. By avoiding these additional steps, the linear algorithm can process small messages more efficiently.



3.2.2. Reduce: Large message (1 MB)

The chain and binary tree algorithms distribute the computational load more evenly across processes, making them more efficient for sum operations with larger message sizes. This is because these algorithms break down the data into smaller chunks and process them in parallel, reducing the burden on any single process. In contrast, the linear algorithm requires the root process to handle the entire aggregation, leading to increased latency as the message size grows. The chain algorithm, in particular, shows a more stable performance compared to the binary tree algorithm. The stability of the chain algorithm is evident from the graph, where it consistently maintains lower latency across different numbers of processes. This stability can be attributed to its systematic way of passing data through a series of steps, ensuring that each process handles a manageable portion of the workload. Therefore, based on the current analysis, the chain algorithm is recommended for sum operations with larger message sizes due to its efficiency and stability.



4. Conclusions & Key Findings

The analysis of OpenMPI's broadcast and reduce operations reveals that the binary tree algorithm performs best for small messages (1 byte) due to efficient communication patterns, with latency increasing as processes grow and being lower within the same CPU due to shared cache usage. For large messages (1 MB), the basic linear algorithm performs poorly, while the chain and binary tree algorithms manage larger sizes more effectively, with the chain algorithm demonstrating superior scalability and stability, making it the preferred choice for highly scalable systems. In reduce operations, the linear algorithm outperforms others for small messages by aggregating data in a single step, whereas the chain and binary tree algorithms distribute the computational load more evenly for large messages, with the chain algorithm showing stable performance across different process counts. These findings underscore the importance of choosing appropriate algorithms based on message size and process count to optimize parallel computing performance, with the chain algorithm being the most efficient and stable option overall. For future improvements, the focus should be on adaptive and hybrid algorithms, dynamic load balancing, network optimization, cache coherence strategies, extensive scalability testing, energy efficiency, and continuous software and algorithm enhancements.