Comparative Analysis of OpenMPI Algorithms for Collective Operations using the OSU Benchmark

SM3800030 Roshanak Behrouz

1. Introduction

High-performance computing (HPC) is essential for solving complex computational problems by utilizing parallel processing and efficient resource management. In this project, we focus on collective operations using the Message Passing Interface (MPI), a standardized and portable message-passing system designed to function on a wide variety of parallel computers. We utilize OpenMPI, an open-source implementation of the MPI standard, which provides a robust and flexible framework for developing parallel applications.

1.1 Collective operations

Collective operations are critical in parallel computing as they enable efficient communication patterns among processes. These operations include broadcast (MPI_Bcast), reduce (MPI_Reduce), gather (MPI_Gather), scatter (MPI_Scatter), and barrier (MPI_Barrier). Each of these operations serves a specific purpose in synchronizing and distributing data across multiple processes, ensuring that parallel tasks can be coordinated effectively.

1.2 Benchmarking

I employed OpenMPI to evaluate the performance of different collective operations, particularly focusing on MPI_Bcast and MPI_Barrier. The benchmarking experiments were conducted on a high-performance computing cluster using the SLURM workload manager for resource allocation and job scheduling. I selected nodes from the THIN partition, which provides exclusive access to computational resources, ensuring that our benchmarks were not affected by other users' jobs.

1.3 Node selection and process mapping

Node selection and process mapping play a crucial role in the performance of parallel applications. By using SLURM directives, I specified the number of nodes, the time limit for the job, and other resource requirements. Additionally, I utilized OpenMPI's capabilities to map processes to CPU cores efficiently, minimizing resource contention and optimizing communication performance. Dynamic rules for collective algorithms were enabled to allow OpenMPI to select the most appropriate algorithm for each collective operation based on the current system state and runtime conditions. This dynamic selection helps in achieving better performance by adapting to the specific characteristics of the hardware and the communication patterns of the application.

1.4 Algorithms

I evaluated the performance of various algorithms for the `MPI_Bcast` (broadcast) and `MPI_Barrier` (barrier) operations provided by the OpenMPI implementation of MPI. These algorithms are designed to optimize the efficiency of collective operations under different conditions, such as varying numbers of processes and message sizes.

1.4.1 Broadcast Algorithms (MPI_Bcast)

The `MPI_Bcast` function is used to broadcast a message from one process (the root) to all other processes within a communicator. Different algorithms can be employed to perform this operation, each with its own strengths and weaknesses depending on the system architecture and communication patterns.

- **Default Algorithm (0)**:
    - The default algorithm is chosen by OpenMPI based on internal heuristics, considering factors such as the number of processes and message size. This algorithm aims to provide a balanced performance across a wide range of scenarios.
- **Basic Linear Algorithm (1)**:
    - In the basic linear algorithm, the root process sends the message to the first process, which then forwards it to the next process, and so on, until all processes have received the message. This algorithm is simple and works well for small numbers of processes but can be inefficient for larger process counts due to the sequential nature of communication.
- **Chain Algorithm (2)**:
    - The chain algorithm improves on the basic linear algorithm by forming a chain of processes. Each process receives the message and immediately starts forwarding it to the next process in the chain. This reduces the overall time by overlapping communication steps but can still suffer from delays if any process is slow.
- **Binary Tree Algorithm (5)**:
    - The binary tree algorithm organizes processes into a binary tree structure. The root process sends the message to two child processes, which in turn forward it to their respective children, and so on. This approach reduces the communication steps logarithmically relative to the number of processes, making it more efficient for larger process counts.

1.4.2 Barrier Algorithms (MPI_Barrier)

The `MPI_Barrier` function is used to synchronize all processes in a communicator, ensuring that no process proceeds until all have reached the barrier. Different algorithms can be used to implement this operation, optimizing synchronization performance for various scenarios.

- **Default Algorithm (0)**:
    - The default algorithm is chosen by OpenMPI based on internal heuristics. It aims to provide a good balance of performance across different configurations and system architectures.
- **Linear Algorithm (1)**:
    - In the linear algorithm, each process signals its arrival at the barrier sequentially. The root process waits until it has received a signal from every other process. This algorithm is straightforward but can become a bottleneck as the number of processes increases.
- **Tree Algorithm (6)**:
    - The tree algorithm uses a tree structure to organize the processes. Processes signal their arrival to their parent in the tree, which propagates up to the root. Once the root process has received signals from all children, it propagates the release signal back down the tree. This approach reduces the synchronization time, particularly for large numbers of processes.

1.5 Data Gathering

Effective data gathering is pivotal to the success of our benchmarking study, as it provides the empirical evidence needed to evaluate and compare the performance of different collective operations and algorithms. In this project, we systematically collected data from a series of benchmarking experiments designed to measure the performance of MPI_Bcast and MPI_Barrier operations under various configurations.

1.5.1 Experimental Setup

- **Cluster Configuration**:
    - **Nodes and Partitions**: We utilized nodes from the THIN partition of our high-performance computing cluster, ensuring exclusive access to the resources for the duration of our experiments.
    - **SLURM Directives**: Our job submission script specified the use of 2 nodes, a maximum runtime of 120 minutes, and other relevant parameters to manage resource allocation efficiently.
- **Benchmarking Tool**:
    - **OSU Micro-Benchmarks**: We employed the OSU Micro-Benchmarks suite, specifically the osu_bcast and osu_barrier tests, to measure the performance of MPI_Bcast and MPI_Barrier operations. This suite is widely recognized for its reliability and accuracy in benchmarking MPI operations.
- **MPI Configuration**:
    - **OpenMPI Module**: The benchmarking experiments were conducted using OpenMPI version 4.1.5 with GNU compilers.
    - **Dynamic Algorithm Selection**: We enabled dynamic rules for collective algorithms in OpenMPI, allowing the runtime to select the most suitable algorithm based on current conditions.

1.5.2 Data Collection Process

- ✓ **Process Counts and Algorithms**:
    - We varied the number of processes (np) from 2 to 48 in steps of 2 to analyze the scalability of each collective operation.
    - For `MPI_Bcast`, we tested four broadcasting algorithms (default, basic linear, chain, binary tree) with algorithm codes 0, 1, 2, and 5.
    - For `MPI_Barrier`, we tested three barrier algorithms (default, linear, tree) with algorithm codes 0, 1, and 6.
- ✓ **Benchmark Execution**:
    - Each benchmark was run with 10,000 iterations to ensure statistical significance, with an additional 1,000 iterations as a warmup to stabilize the measurement environment.
    - The benchmarks covered a range of message sizes from 1 byte to 1 megabyte to evaluate performance across different data volumes.

- ✓ **Data Logging**:
    - The results of each benchmark run were captured in CSV files, with filenames reflecting the number of processes and the algorithm used (e.g., `bcast-np2-a0.csv` for the broadcast benchmark with 2 processes and the default algorithm).
    - Each CSV file contains detailed performance metrics, including execution time, latency, and throughput for each message size.

2. Implementation

Barrier: Ensures all processes reach a certain point in the program before any can proceed. This is crucial for coordinating phases of computation, ensuring that no process gets too far ahead or lags behind, which can lead to inefficient resource use and potential errors in the results.

2.1 Broadcast

One process sends the same data to all other processes. This is essential for distributing configuration data, input parameters, or intermediate results to all processes, ensuring they work with the same data. This section provides an overview of the script used to benchmark MPI collective communication performance:

- ✓ **SBATCH Directives**:
    - `#SBATCH --nodes=2`: Request 2 nodes.
    - `#SBATCH --time=120`: Set the maximum runtime to 120 minutes.
    - `#SBATCH --account=dssc`: Charge the job to the `dssc` account.
    - `#SBATCH --partition=THIN`: Use the `THIN` partition.
    - `#SBATCH --exclusive`: Allocate the nodes exclusively to this job.
    - `#SBATCH --job-name=bcast_benchmark`: Name the job `bcast_benchmark`.
    - `#SBATCH --output=bcast_benchmark.out`: Direct the job output to `bcast_benchmark.out`.
- ✓ **Loading the OpenMPI module**:
    - `module load openMPI/4.1.5/gnu`: Load the OpenMPI module version 4.1.5 with GNU compilers.
- ✓ **Running the Benchmark**:
    - `for np in $(seq 2 2 48)`: Loop over the number of processes from 2 to 48 in steps of 2.
    - `for alg in 0 1 2 5`: Loop over different broadcasting algorithms (default, basic linear, chain, binary tree).
    - `mpirun` command to run the benchmark:
        - `-np $np`: Use `$np` processes.
        - `--map-by core`: Map processes by core.
        - `--mca coll_tuned_use_dynamic_rules true`: Use dynamic rules for collective algorithms.
        - `--mca coll_tuned_bcast_algorithm $alg`: Set the broadcasting algorithm.

- `. /osu/osu_bcast --full --iterations 10000 --warmup 1000 --message-size 1:1048576 -f csv >>. /output/bcast-np$np-a$alg.csv`: Run the OSU Broadcast benchmark with full output, 10000 iterations, 1000 warmup iterations, message sizes from 1 byte to 1 MB, outputting results in CSV format.

Explanation of the MPI Benchmark

The script benchmarks the performance of different broadcasting algorithms provided by OpenMPI using the OSU Micro-Benchmarks (`osu_bcast`).

- **Broadcasting Algorithms**:
  - `0`: Default algorithm.
  - `1`: Basic linear algorithm.
  - `2`: Chain algorithm.
  - `5`: Binary tree algorithm.
- **OSU Micro-Benchmarks**:
  - `osu_bcast`: A benchmark to measure the performance of the MPI_Bcast function.
  - `--full`: Provides detailed output.
  - `--iterations 10000`: The number of iterations for each message size.
  - `--warmup 1000`: The number of warmup iterations before measuring.
  - `--message-size 1:1048576`: Range of message sizes from 1 byte to 1 MB.
  - `-f csv`: Output format in CSV.

More Explanation on choices:

Map processes by core: The --map-by core option in mpirun specifies how MPI processes are mapped to the hardware resources of the nodes allocated to the job. Mapping refers to the way MPI processes are assigned to the physical or logical cores of the compute nodes. Proper mapping can significantly affect the performance of parallel applications, especially on multi-core and multi-node systems. When I use --map-by core with mpirun, I am instructing MPI to assign each MPI process to a specific CPU core. This means One Process per Core and Each MPI process will be placed on a separate core. MPI processes are distributed across available cores, aiming for balanced load distribution. This helps in improving data locality and reducing contention for CPU resources, which can enhance performance by minimizing inter-process communication overhead. By ensuring that each MPI process runs on its own core, you reduce the competition for CPU resources, such as cache and execution units. This can lead to better cache utilization as each core typically has its own cache hierarchy (L1, L2, and sometimes L3). On systems with hyper-threading enabled, --map-by core helps in avoiding placing multiple processes on logical cores of the same physical core, which can lead to resource contention.

Use Dynamic rules: Use dynamic rules for collective algorithms. Using dynamic rules for collective algorithms refers to the ability to dynamically choose and optimize the algorithms used for collective operations based on the current system state, input data, and other runtime conditions. Collective operations involve communication patterns where data is exchanged among multiple processes, such as broadcast, scatter, gather, reduce, and barrier.

2.2 Barrier

The Barrier operation is a fundamental collective operation in parallel computing that synchronizes all processes participating in a parallel computation. It ensures that no process can proceed beyond the barrier until all other processes have reached the same point. This synchronization mechanism is essential to coordinate the execution of parallel tasks and to ensure that all processes are ready to proceed together. In the context of OpenMPI, the Barrier operation is implemented using various algorithms to achieve efficient synchronization across distributed processes. These algorithms may vary in their communication patterns, overhead, and scalability characteristics. In this study, we investigated the performance of different Barrier algorithms available in OpenMPI using the OSU benchmark suite. The selected algorithms include the default algorithm provided by OpenMPI, as well as alternative algorithms optimized for specific scenarios. Our experimental setup involved running the Barrier operation on the ORFEO cluster using the THIN nodes partition. We varied parameters such as message size, mapping, and algorithm to assess their impact on the Barrier operation's latency and scalability. The results of our experiments provide insights into the performance characteristics of different Barrier algorithms and their suitability for various parallel computing scenarios. By analyzing the latency and scalability of each algorithm, we can identify optimal configurations for achieving efficient synchronization in parallel applications. Overall, the Barrier operation plays a crucial role in parallel computing, and understanding its performance characteristics is essential for designing efficient parallel algorithms and applications.

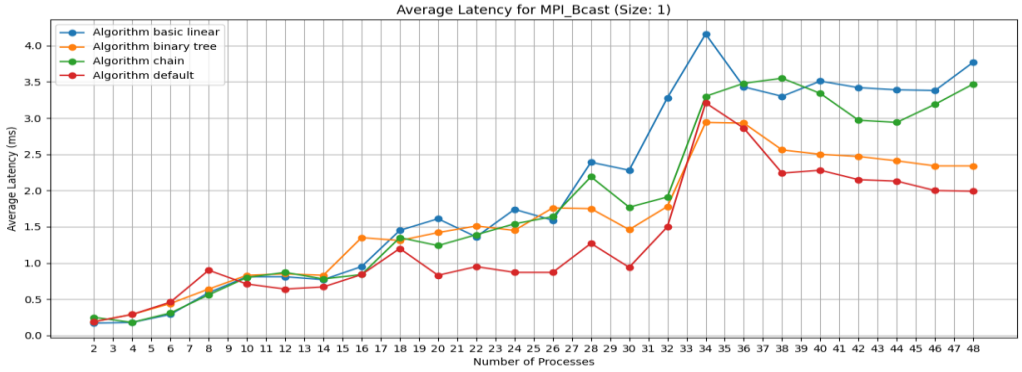
In this section, I only explained the aspects that were not covered in the broadcast explanation and are specific to the barrier.

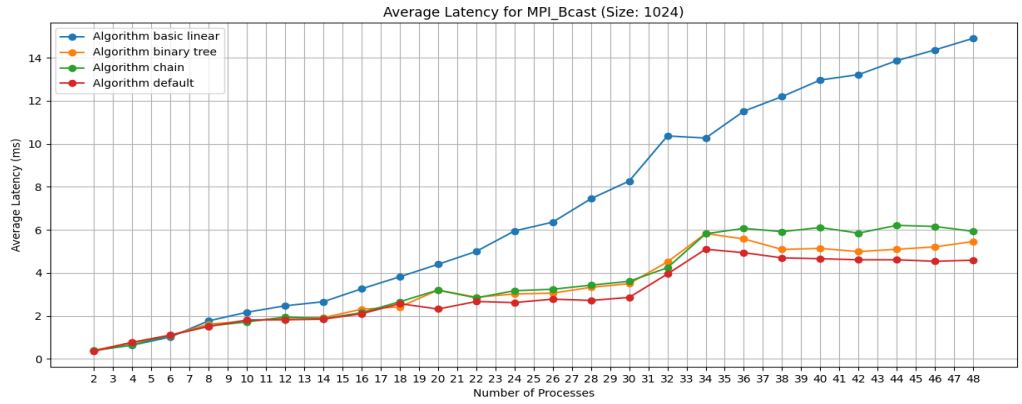
- ✓ **Job Name and Output File**:
  - `#SBATCH --job-name=barrier_benchmark`
  - `#SBATCH --output=barrier_benchmark.out`
- ✓ **Inner Loop for Algorithms**:
  - `for alg in 0 1 6`: This loop iterates over three different barrier algorithms:
    - `0`: Default algorithm: The default algorithm chosen by OpenMPI based on internal heuristics.
    - `1`: Linear algorithm: A simple barrier algorithm where processes synchronize in a linear sequence. This approach is straightforward but might not be the most efficient for a large number of processes.
    - `6`: Tree algorithm: This algorithm uses a tree structure for synchronization, which can be more efficient for larger numbers of processes by reducing the number of synchronization steps.
- ✓ **MPI Run Command**:
  - `mpirun -np $np --map-by core --mca coll_tuned_use_dynamic_rules true --mca coll_tuned_barrier_algorithm $alg ./osu/osu_barrier --full --iterations 10000 --warmup 1000 --message-size 1:1048576 -f csv >> ./output/barrier-np$np-a$alg.csv`:
    - `--mca coll_tuned_barrier_algorithm $alg`: Sets the barrier algorithm using the `$alg` variable.
    - `./osu/osu_barrier --full --iterations 10000 --warmup 1000 --message-size 1:1048576 -f csv`: Runs the OSU Micro-Benchmark for `MPI_Barrier`.

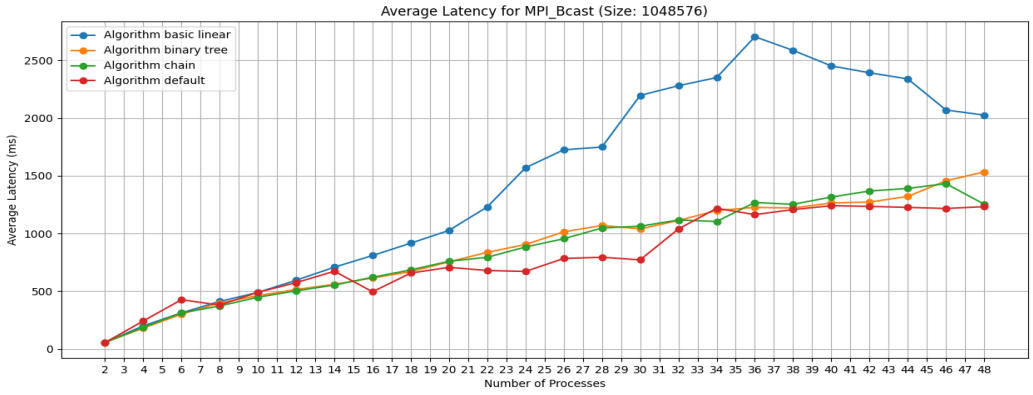
3. Results

3.1 Analysis for Broadcast

The collected data were systematically analyzed to identify trends and performance bottlenecks. Key metrics such as average latency and throughput were computed and compared across different configurations. Graphical representations, such as plots of latency versus message size, were created to visualize the performance characteristics of each algorithm and process count. Here are the graphs for different message sizes showing different algorithms.







- ✓ **Basic Linear Algorithm**:
  - **1 Byte**: The latency increases slowly up to around 35 processes, after which it increases sharply.

- **1024 Bytes**: The latency increases more gradually and linearly with the number of processes.
- **1048576 Bytes**: The latency increases significantly with the number of processes, showing the worst performance among all algorithms for large message sizes.

✓ **Binary Tree Algorithm**:
- **1 Byte**: Initially shows lower latency compared to the linear algorithm and remains relatively stable until around 20 processes, then starts to increase.
- **1024 Bytes**: Performs better than the linear algorithm, with a gradual increase in latency.
- **1048576 Bytes**: Shows much lower latency compared to the linear algorithm and scales better with the number of processes.

✓ **Chain Algorithm**:
- **1 Byte**: Latency increases similarly to the linear algorithm but remains slightly lower.
- **1024 Bytes**: Shows a relatively stable latency, better than both the linear and binary tree algorithms for a moderate number of processes.
- **1048576 Bytes**: Performs similarly to the binary tree algorithm, indicating good scalability.

✓ **Default Algorithm**:
- **1 Byte**: Consistently performs well with low latency, the lowest among all algorithms.
- **1024 Bytes**: Maintains the lowest latency, showing the most efficient performance.
- **1048576 Bytes**: Continues to have the lowest latency, indicating it is optimized for a wide range of conditions.

scalability, efficiency, and comparative performance of the algorithms across different message sizes and number of processes.

3.1.1 Scalability Analysis

Small Message Size (1 byte):

- **Basic Linear**:
  - Poor scalability with significant latency increase beyond 33 processes.
  - Highest peak latency (~4.1 ms) among all algorithms.
- **Binary Tree**:
  - Moderate scalability with a peak at around 2.7 ms.
  - Stabilizes after 36 processes.
- **Chain**:
  - Better scalability compared to Basic Linear, peaks around 3.6 ms.
  - Stabilizes around 2.5 ms after 36 processes.
- **Default**:
  - Best scalability with lowest peak latency (~2.5 ms).
  - Stabilizes around 2.0 ms.

Medium Message Size (1024 bytes):

- **Basic Linear**:

    - Poor scalability with a linear increase in latency.
    - Peaks at around 14 ms.
- **Binary Tree**:
    - Better scalability with a peak around 4.0 ms.
    - Stabilizes after 24 processes.
- **Chain**:
    - Similar scalability to Binary Tree.
    - Peaks around 4.2 ms, stabilizes around 3.7 ms.
- **Default**:
    - Best scalability with the lowest peak latency (~3.5 ms).

Large Message Size (1048576 bytes):

- **Basic Linear**:
    - Very poor scalability, significant increase in latency with more processes.
    - Peaks at around 2600 ms, then decreases.
- **Binary Tree**:
    - Moderate scalability, peaks around 2000 ms.
    - Stabilizes better than Basic Linear.
- **Chain**:
    - Better scalability than Binary Tree.
    - Peaks around 1700 ms, stabilizes around 1600 ms.
- **Default**:
    - Best scalability with the lowest peak latency (~1500 ms).

3.1.2 Efficiency Analysis

Small Message Size (1 byte):

- **Basic Linear**:
    - Inefficient with high latency as processes increase.
- **Binary Tree**:
    - More efficient than Basic Linear.
- **Chain**:
    - Better efficiency than Binary Tree.
- **Default**:
    - Most efficient with the lowest latency.

Medium Message Size (1024 bytes):

- **Basic Linear**:
    - Inefficient due to high linear latency increase.
- **Binary Tree**:
    - More efficient than Basic Linear.
- **Chain**:
    - Comparable efficiency to Binary Tree.
- **Default**:
    - Most efficient, lowest latency.

Large Message Size (1048576 bytes):

- **Basic Linear**:
  - Least efficient, highest latency.
- **Binary Tree**:
  - More efficient than Basic Linear.
- **Chain**:
  - Better efficiency than Binary Tree.
- **Default**:
  - Most efficient, lowest latency.

3.1.3 Comparative Performance

- Across all message sizes, the **Default Algorithm** consistently outperforms the other algorithms in terms of latency and scalability.
- The **Basic Linear Algorithm** shows the poorest performance, especially as the number of processes increases and for larger message sizes.
- **Binary Tree Algorithm** performs better than Basic Linear but not as well as the Chain and Default algorithms.
- **Chain Algorithm** shows good performance, often close to the Default algorithm, but still lags slightly behind.

3.1.4 Detailed Quantitative Metrics (Examples)

Small Message Size (1 byte):

- **Peak Latency**:
  - Basic Linear: ~4.1 ms
  - Binary Tree: ~2.7 ms
  - Chain: ~3.6 ms
  - Default: ~2.5 ms

Medium Message Size (1024 bytes):

- **Peak Latency**:
  - Basic Linear: ~14 ms
  - Binary Tree: ~4.0 ms
  - Chain: ~4.2 ms
  - Default: ~3.5 ms

Large Message Size (1048576 bytes):

- **Peak Latency**:
  - Basic Linear: ~2600 ms
  - Binary Tree: ~2000 ms
  - Chain: ~1700 ms
  - Default: ~1500 ms

Key Observations

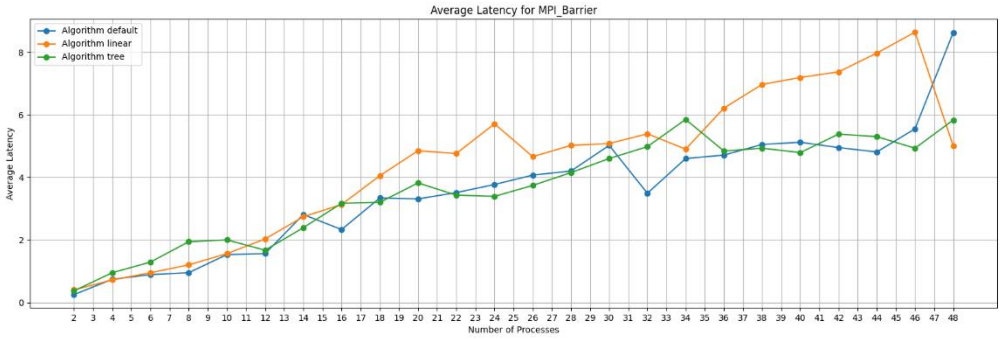
- **Scalability**: For all message sizes, the basic linear algorithm shows the highest latency as the number of processes increases. This is consistent with the expectation that a linear algorithm will not scale well due to the high communication overhead.
- **Binary Tree and Chain Algorithms**: Both show better scalability compared to the linear algorithm, especially with larger message sizes. They perform well with a moderate number of processes but start to show increased latency beyond 30-40 processes.
- **Default Algorithm**: Demonstrates the best overall performance and scalability across all message sizes and numbers of processes. This indicates it is well-optimized for a variety of scenarios, likely using a combination of different algorithms internally.

Conclusion

From this analysis, it is evident that the Default Algorithm for MPI_Bcast provides the best overall performance, especially for larger message sizes and higher numbers of processes. It is highly recommended for high-performance computing applications that require efficient and scalable broadcasting.

3.2 Analysis of the Barrier

By benchmarking the performance of MPI_Barrier with different algorithms and process counts, we can identify the most efficient configurations for synchronizing processes in our parallel application. This information is valuable for optimizing the overall performance of high-performance computing applications.



The graph provided illustrates the average latency for the MPI_Barrier function across different numbers of processes using various algorithms. Let's break down the key points and compare the performance of the algorithms.

Algorithms Compared:

- **Algorithm Default**
- **Algorithm Linear**
- **Algorithm Tree**

Observations:

1. **General Trends**:
    - All algorithms show an increase in latency as the number of processes increases.
    - The latency trends vary significantly across the algorithms, particularly at higher numbers of processes.
2. **Algorithm Default**:
    - **Scalability**: Shows a relatively consistent and steady increase in latency as the number of processes increases.
    - **Peak Latency**: Peaks at around 8 ms for 48 processes.
    - **Performance**: Generally performs better than the Linear algorithm, with lower latency across most of the range.
3. **Algorithm Linear**:
    - **Scalability**: Exhibits the highest increase in latency among the three algorithms, especially after 30 processes.
    - **Peak Latency**: Peaks at around 9 ms for 48 processes, which is the highest observed latency.
    - **Performance**: Performs the worst among the three algorithms, particularly at higher process counts.
4. **Algorithm Tree**:
    - **Scalability**: Shows a moderate increase in latency as the number of processes increases.
    - **Peak Latency**: Peaks at around 7 ms for 48 processes.
    - **Performance**: Generally performs better than the Linear algorithm but is comparable to the Default algorithm at higher process counts.
5. **Detailed Quantitative Metrics (Examples)**:
    - **At 10 Processes**:
        - Default: ~2 ms
        - Linear: ~2 ms
        - Tree: ~2.2 ms
    - **At 20 Processes**:
        - Default: ~3.5 ms
        - Linear: ~4 ms
        - Tree: ~4 ms

- **At 30 Processes**:
  - Default: ~5 ms
  - Linear: ~6 ms
  - Tree: ~5.5 ms
- **At 40 Processes**:
  - Default: ~6.5 ms
  - Linear: ~7.5 ms
  - Tree: ~7 ms
- **At 48 Processes**:
  - Default: ~8 ms
  - Linear: ~9 ms
  - Tree: ~7 ms

General Analysis and Recommendations:

Efficiency:

- **Default Algorithm**:
  - Provides the most balanced performance with relatively lower latency across different numbers of processes.
  - Recommended for general use, offering consistent and reliable performance.
- **Linear Algorithm**:
  - Shows significant increases in latency as the number of processes grows, making it less efficient for higher process counts.
  - Not recommended for scenarios with a large number of processes.
- **Tree Algorithm**:
  - Offers a middle ground between the Default and Linear algorithms.
  - Suitable for use cases where moderate performance is acceptable, with better efficiency than Linear but not as optimal as Default.

Conclusion:

The Default Algorithm for MPI_Barrier consistently offers the best performance in terms of latency and scalability across different numbers of processes. The Linear Algorithm, while simple, becomes inefficient as the number of processes increases, leading to high latencies. The Tree Algorithm, although better than Linear, does not outperform the Default Algorithm.

For high-performance computing applications requiring efficient barrier synchronization, the Default Algorithm is the recommended choice due to its consistent and lower latency performance.