[HPC Report Assignment 1]

Luca Pernice

SM3800022

GitHub Repository

[Batch Files]

1. Script Organization

- I structured the scripts with appropriate comments and clear sectioning for readability.
- Each script begins with SLURM directives (**#SBATCH**) for specifying job parameters like job name, number of nodes, tasks per node, time limit, and partition.
- I loaded the required MPI module (**openMPI/4.1.5/gnu/12.2.1**) to ensure the environment is set up correctly.

2. Benchmarking Process

- I initialized variables such as **repetitions** for the number of repetitions to obtain an average and algorithms array to store the algorithms to be tested.
- Both scripts iterate over the number of processes (processes) and the size of the message (**size**) to cover a range of scenarios.
- Inside the loops, I nested another loop to iterate over each algorithm to test them against the specified collective operation.

3. Benchmark Execution

- I used **mpirun** to execute the OSU benchmark binaries (**osu_bcast** for broadcast, **osu_barrier** for barrier) with appropriate options.
- For each iteration, I captured the latency result and appended it to a CSV file for further analysis.

4. MPI Options

- Within the **mpirun** command, I utilized options such as **--map-by core** for task mapping and specified MPI collective tuning parameters (-**-mca coll_tuned_use_dynamic_rules true**) to enable dynamic selection of algorithms.
- Additionally, I specified the algorithm to be used for each collective operation (**--mca coll_tuned_bcast_algorithm** for broadcast, **--mca coll_tuned_barrier_algorithm** for barrier).

5. Output Handling

- I captured the output of the benchmark execution using **tail** and **awk** commands to extract the relevant latency information.
- Results are printed to the console during execution for monitoring and also appended to CSV files (**bcast_algorithms_thin.csv** and **barrier_algorithms_thin.csv**) for later analysis.

6. Scalability Testing

- I designed the scripts to scale across different numbers of processes and message sizes, providing a comprehensive evaluation of the MPI algorithms' performance under varying conditions.

[THIN Partition]

For executing the benchmark, I utilized the THIN nodes available on the cluster. On the cluster's Thin partition, there are 12 Intel-based nodes available. Among them, two nodes are powered by Xeon Gold 6154 CPUs [FAT nodes] , while the remaining ten nodes utilize Xeon Gold 6126 CPUs. Notably, the two Xeon Gold 6154 CPUs have more cores, with 18 cores each, compared to the 12 cores available on the Xeon Gold 6126 CPUs. This difference in core count may influence performance for parallel workloads.

To streamline job allocation and maximize resource utilization, I excluded the Xeon Gold 6154 CPUs from my job submission using the flag **#SBATCH --exclude fat[001-002]**. This directive ensures that my job exclusively utilizes the Xeon Gold 6126 CPUs, which are better suited for my computational requirements.

[Bcast]

1. Basic Linear:

The Basic Linear broadcast algorithm, also known as the naive or simple algorithm, is one of the most straightforward approaches for broadcasting data in a parallel computing environment. In this algorithm, the root process (sender) sends the data directly to each of the other processes (receivers) one by one, typically in a linear fashion. Each receiver waits to receive the data from the root before it can proceed with its computation.

Advantages:

- Simple and easy to implement
- Minimal overhead in terms of communication and coordination.

Disvantages:

- Limited scalability: As the number of processes increases, the time taken for the broadcast operation also increases linearly.
- Inefficient for large-scale systems with a high number of processes due to increased communication overhead.

2. Chain

The Chain broadcast algorithm aims to reduce the communication overhead of the Basic Linear algorithm by organizing processes into a chain-like structure. In this approach, processes are arranged in a linear sequence, with each process responsible for forwarding the data to the next process in the chain until all processes receive the data.

How it works:

1- The root process sends the data to its immediate neighbor.
2- Each process, except the last one, forwards the received data to its next neighbor.
3- The last process in the chain receives the data and completes the broadcast operation.

Advantages:

- Reduced communication overhead compared to the Basic Linear algorithm, especially for large numbers of processes.
- Better scalability as the number of processes increases.

Disvantages:

- Still limited by the linear nature of the chain, which can lead to bottlenecks if the chain becomes too long.
- Requires additional logic for process coordination and message forwarding.

3. Pipeline

The Pipeline broadcast algorithm enhances the Chain algorithm by overlapping communication and computation, thus reducing the overall latency of the broadcast operation. In a pipeline, multiple stages are involved, with each stage responsible for transmitting data to the next stage while simultaneously processing the data it receives.

How it works:

1- The pipeline is divided into multiple stages, with each stage consisting of a subset of processes.
2- Data is transmitted from one stage to the next in a pipelined fashion, with each stage processing the data it receives while forwarding it to the next stage.
3- The broadcast operation completes once the data reaches the final stage of the pipeline.

Advantages:

- Improved performance and reduced latency compared to linear algorithms.
- Efficient utilization of resources by overlapping communication and computation.

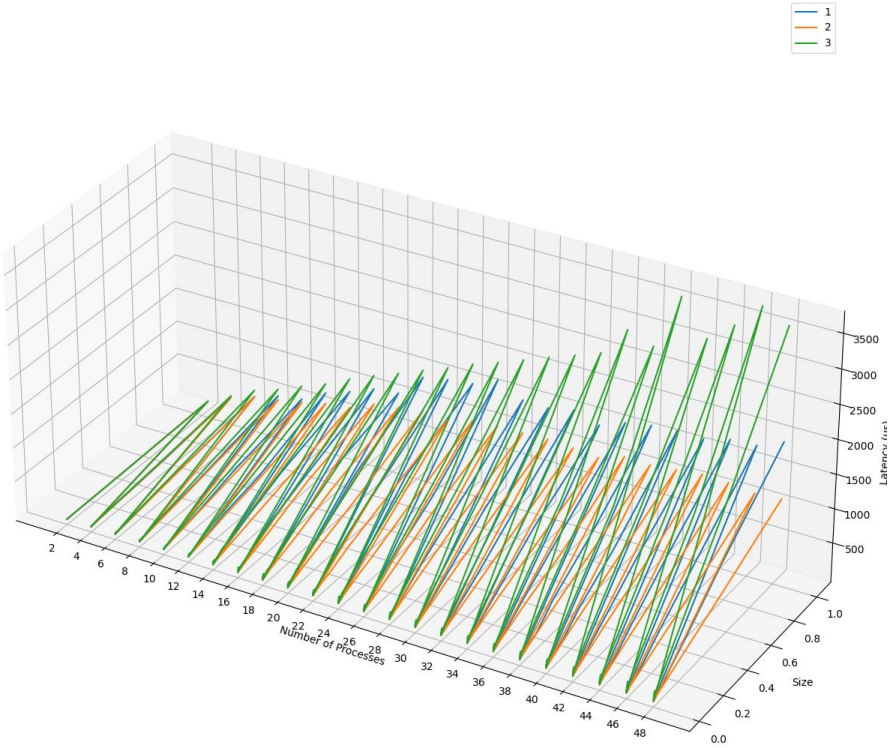
Disvantages:

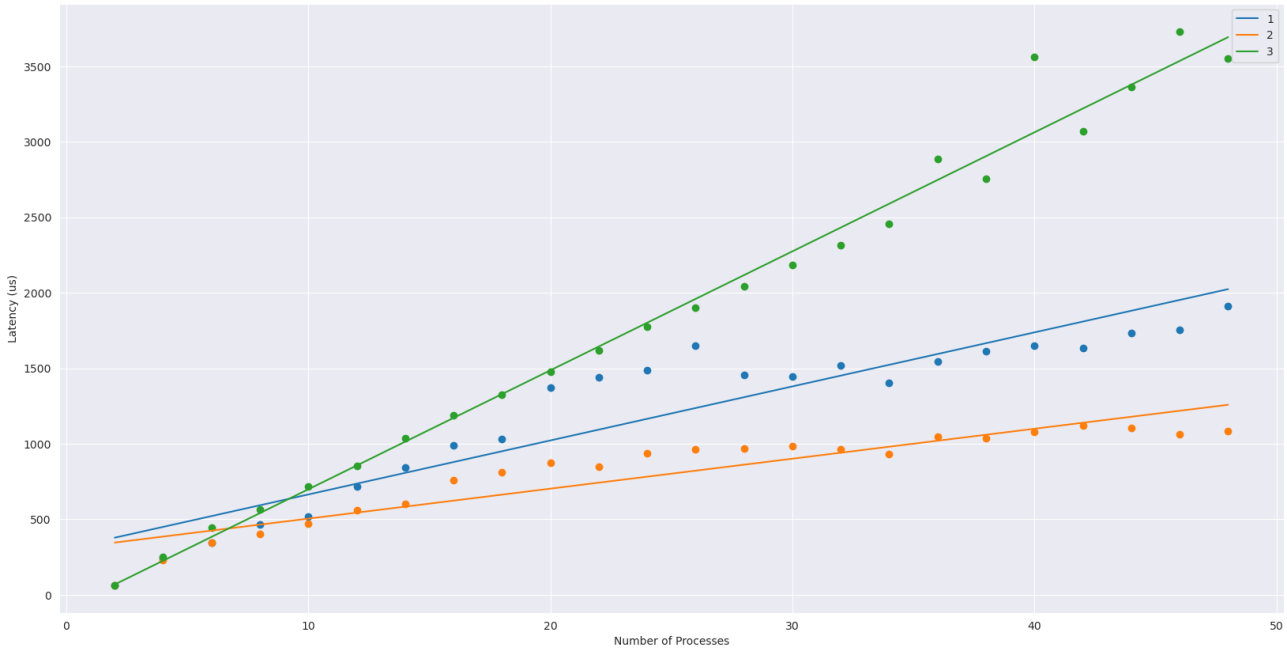
- Complexity increases with the number of stages in the pipeline.
- Sensitive to load imbalance and communication delays between stages, which can impact overall performance.

[Results]

The CSV file containing the collected results is available in the GitHub repository. The following 3D plot displays sizes normalized to the range [0,1], with the actual values being powers of 2, ranging from 2 to 1,048,576. For simplicity and effectiveness, a linear model (Latency ~ Size + Number of Processes) is fitted and displayed in the 2D plot. The coefficients of the fitted linear model are shown in the accompanying table. Notably, there is no evidence of a 'spike' or anomaly when the number of tasks reaches 25, suggesting that the allocation of a new task to the second node does not increase latency as expected. This indicates that internode communication does not significantly impact latency under these conditions.

Algorithm	Bias term (intercept)	Size coefficient	Processes coefficient
1	-80.461191	0.001142	3.943316
2	-44.381542	0.000758	2.278290
3	-183.050744	0.001747	8.431862





[Barrier]

1. Linear Barrier

The Linear Barrier algorithm, also known as the simple or naive barrier, is one of the most straightforward approaches for synchronizing processes in a parallel computing environment. In this algorithm, each process waits until all other processes have reached the barrier before proceeding. This synchronization is typically achieved through a central control point where processes signal their arrival.

How it works:

1. Each process reaches the barrier and waits for a signal from the control point.
2. Once all processes have arrived, the control point sends a signal indicating that all processes can proceed.

Advantages:

- Simple and easy to implement
- Minimal overhead in terms of communication and coordination.

Disvantages:

- Limited scalability: As the number of processes increases, the time taken for synchronization also increases linearly.
- Inefficient for large-scale systems with a high number of processes due to increased communication overhead and contention at the central control point.

2. Double Ring Barrier

The Double Ring Barrier algorithm improves upon the Linear Barrier by introducing a more structured approach to the synchronization process. In this algorithm, processes are organized into two rings, an inner ring and an outer ring. Each process signals its arrival to the processes in both rings, ensuring that synchronization occurs in a more distributed manner.

How it works:

1. Processes are arranged in two rings: an inner ring and an outer ring.
2. Each process signals its arrival to the processes in both rings.
3. Once all processes in both rings have signaled their arrival, synchronization is achieved, and processes can proceed.

Advantages:

- Reduced communication overhead compared to the Linear Barrier, especially for large numbers of processes.
- Better scalability due to the distributed nature of synchronization across the two rings.

Disvantages:

- Requires additional logic for organizing processes into rings and coordinating the synchronization process.
- Complexity increases with the number of processes and rings, which can impact implementation and performance.

3. Recursive Doubling Barrier

The Recursive Doubling Barrier algorithm further enhances the scalability and efficiency of barrier synchronization by leveraging a recursive approach. In this algorithm, processes are organized into a binary tree structure, and synchronization occurs by recursively combining pairs of processes until all processes have synchronized.

How it works:

1- Processes are arranged in a binary tree structure.

2- Synchronization occurs by recursively combining pairs of processes, with each pair synchronizing at their parent node in the tree.
3- Once all processes have synchronized at the root of the tree, synchronization is achieved, and processes can proceed.

Advantages:

- Improved performance and reduced latency compared to linear or ring-based algorithms.
- Efficient utilization of resources by leveraging a hierarchical structure for synchronization.

Disvantages:

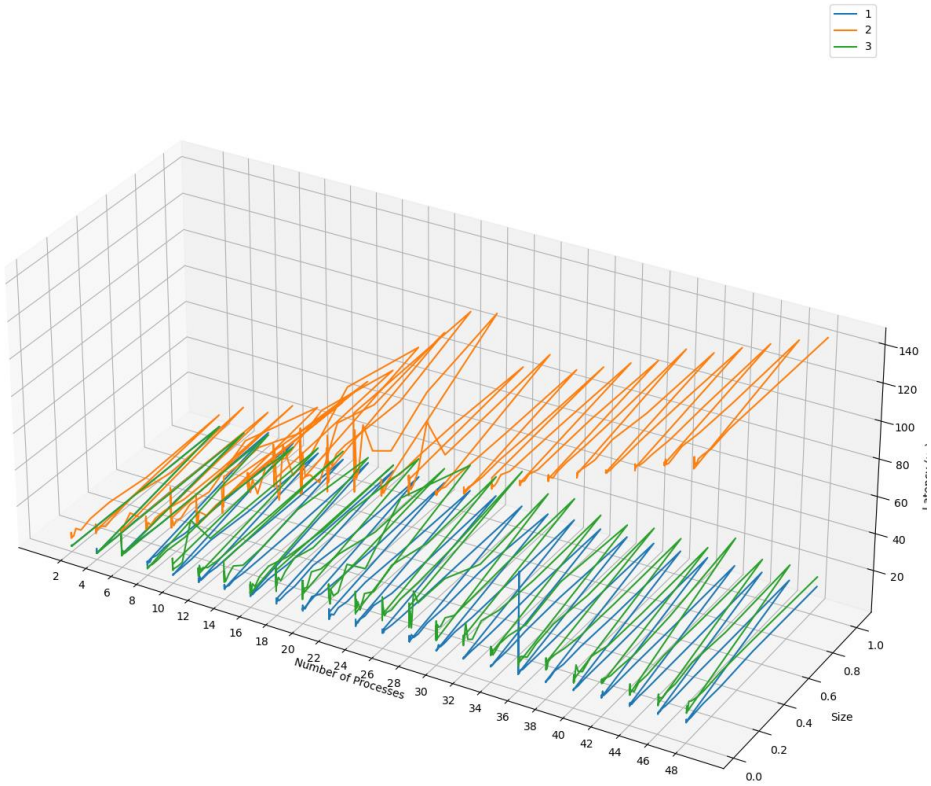
- Complexity increases with the depth of the binary tree and the number of processes, which can impact implementation and performance.
- Sensitive to load imbalance and communication delays between processes, especially in deep trees or for irregular process distributions.

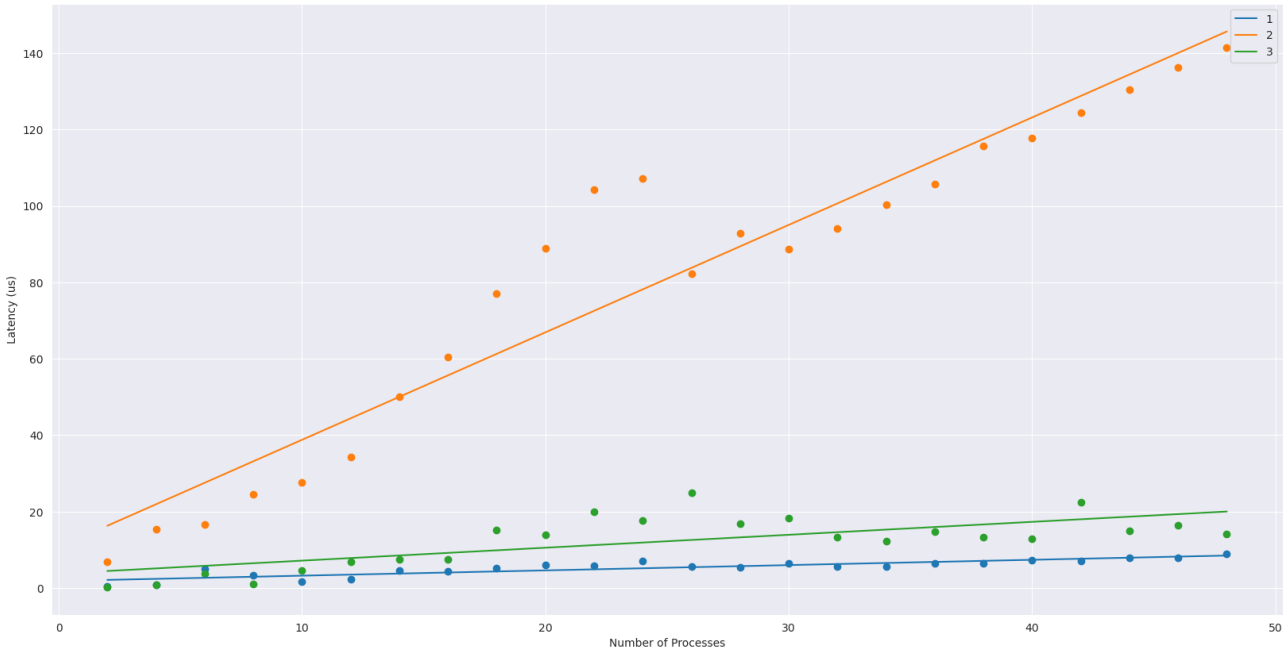
[Results]

The CSV file with all the data is available in the GitHub repository. The observations and conclusions for the broadcast (Bcast) results are applicable here as well. The table below presents the coefficients of the fitted linear models for the barrier algorithms:

Algorithm	Bias term (intercept)	Size coefficient	Processes coefficient
1	1.639068	-2.320911e-07	0.152075
2	9.068090	1.216480e-06	2.827884
3	4.528924	1.488068e-07	0.303596

These results highlight the relationship between latency, message size, and the number of processes. Similar to the broadcast results, no significant 'spike' or anomaly is observed when the number of tasks reaches 25, indicating that the allocation of tasks to a second node does not adversely affect latency. This suggests that internode communication does not substantially impact performance in this context.


1
2
3
Number of Processes
Size


1
2
3
Latency (us)
Number of Processes