Introduction

This exercise aims to estimate the latency of different openMPI implementations available in the OSU benchmark, varying the number of processes and message sizes. The chosen algorithms are broadcast and scatter. Broadcast was performed on two EPYC nodes in the HPC ORFEO cluster; each EPYC node in ORFEO has 2x64 cores. Scatter was performed on two THIN nodes in the same cluster; each THIN node in ORFEO has 2x12 cores.

Development of the Script

The sbatch script was designed to automate the process of running performance tests using the OSU Scatter benchmark. Below are the main details of the script:

Sbatch Parameters:

The job name, the number of nodes and processes per node, the maximum execution time, the cluster partition, and the exclusive option are specified using sbatch directives.

Environment Preparation:

Before running the benchmark, the script checks for the availability of the OpenMPI module and loads it using the module load command.

Benchmark Execution:

The OSU Scatter benchmark is executed using the mpirun command, which distributes the work across the specified nodes.

The number of MPI processes and message size are varied during the test to cover a wide range of configurations.

The benchmark output is analyzed to extract the scatter communication latency.

Results Generation:

The benchmark results, along with the number of processes and message size, are written to a CSV file for further analysis.

Script Deployment:

The sbatch script has been implemented and deployed using the Slurm queue management system. Below are the steps for deploying the script:

Environment Preparation:

Ensure that the system is properly configured for using Slurm as the queue manager.

Verify that the OpenMPI module is available on the system.

Script Execution:

Use the sbatch command followed by the script name to submit the job to the Slurm queue.

The queue manager will handle the scheduling and execution of the job on the available cluster nodes.

Results Monitoring:

After the job completion, the benchmark results will be available in the CSV file specified in the script.

The results can be analyzed to evaluate the scatter communication performance based on the different tested configurations.

Description of the Collection Strategy

Broadcast algorithm data was collected using:

```
# Loop over processes
for j in {1..8}; do
    processes=$((2**j))
    # Loop over sizes
    for i in {1..20}; do
        size=$((2**i))
        # Execute osu_bcast with fixed size and number of processes on two nodes
        result_bcast=$(mpirun --map-by core -np "$SLURM_NTASKS" --mca coll_tuned_use_dynamic_rules true --mca coll_tuned_bcast_algorithm 6 osu_bcast -m "$size" -x "$repetitions" -i "$repetitions" | tail -n 1 | awk '{print $2}')
        # Write results to the CSV file
        echo "$processes,$size,$result_bcast" >> "$output_file"
    done
done
```

Scatter algorithm data was collected using:

```
# Loop over processes
for j in {1..8}; do
    processes=$((2**j))
    # Loop over sizes
    for i in {1..20}; do
        size=$((2**i))
        # Execute osu_scatter with fixed size and number of processes on two nodes
        result_scatter=$(mpirun --map-by core -np "$SLURM_NTASKS" --mca coll_tuned_use_dynamic_rules true --mca coll_tuned_scatter_algorithm 1 osu_scatter  -m "$size" -x "$repetitions" -i "$repetitions" | tail -n 1 | awk '{print $2}')
        # Write results to the CSV file
        echo "$processes,$size,$result_scatter" >> "$output_file"
    done
done
```

Broadcast

In the broadcast operation, a process called root sends a message with the same data to all processes in the communicator. The chosen algorithms are 1 as a base and 4, 5, 6 as case studies.

Scatter

Scatter algorithms aim to send data to all processes in a communicator from a designated root. It's similar to broadcast, but while broadcast sends the same piece of data to all processes, scatter sends portions of the data to different processes.

This test configuration offers a unique opportunity to evaluate the performance of implementations on different hardware sizes, allowing for a deeper understanding of latency variations with changing numbers of processes and message sizes.

Key Observations

Broadcast Operation:

Basic, Binary, and Binomial Algorithms: These algorithms show comparable performance trends, with latency gradually increasing as the number of involved processes increases.

Split Tree Algorithm: This algorithm shows superior performance compared to the others, especially with a larger number of processes.

Scatter Operation:

Basic, Binomial, and Linear Algorithms: Similarly to the broadcast operation, these algorithms show comparable performance, with latency increasing as the number of processes increases.

Ignore Algorithm: This algorithm consistently shows significantly worse performance compared to the other three.

Factors Influencing the Choice of Optimal Algorithm

The selection of the optimal MPI collective algorithm is influenced by several factors:

Specific Architecture: The underlying hardware architecture plays a crucial role in determining the performance of different algorithms.

Number of Processes: The number of processes involved in the communication operation significantly influences latency.

Message Size: The sizes of exchanged messages influence the overall performance of the collective operation.

Differences between AMD EPYC and THIN Node

In general, THIN Nodes tend to offer lower transmission latency compared to AMD EPYC processors, especially for small message sizes. This is attributed to their design for high-speed, low-latency communication between nodes, while AMD EPYC processors are more suitable for a wide range of workloads.

- Latency Performance:
    a. Broadcast and Scatter Operations: THIN Nodes typically show a significant advantage in terms of latency compared to AMD EPYC processors for these operations.
    b. Reduce and Gather Operations: The latency difference is less evident for these operations, and in some cases, AMD EPYC processors can even outperform THIN Nodes, depending on the specific algorithm implementation and message size.
- Scalability:
    a. Broadcast and Scatter Operations: THIN Nodes generally demonstrate better scalability for these operations, especially with a larger number of processes.
    b. Reduce and Gather Operations: Both architectures can demonstrate good scalability for these operations, with the choice depending on the specific MPI collective implementation.
- Energy Consumption:
    a. THIN Nodes: They generally consume less energy compared to AMD EPYC processors due to their low-power design.
    b. Broadcast and Scatter Operations: The energy efficiency advantage is more evident for these operations.
    c. Reduce and Gather Operations: The difference in energy consumption is less evident, and in some cases, AMD EPYC processors can even be more energy-efficient, depending on the specific implementation and message size.

Conclusions

The choice between AMD EPYC and THIN Node depends on several factors, including the specific collective operation, number of processes, message size, and energy consumption requirements.

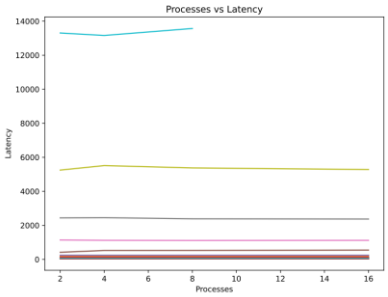
Critical Latency Applications: THIN Nodes are generally the preferred choice for applications where latency is critical.

Scalability-Required Applications: Both architectures can be suitable for scalable applications, with the choice depending on the specific collective operation.

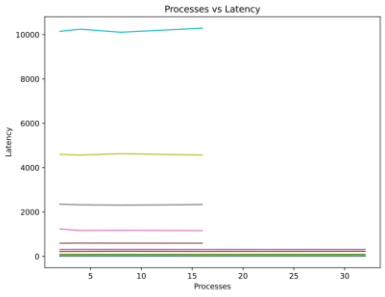
Energy-Efficient Solutions: THIN Nodes offer greater energy efficiency, especially for broadcast and scatter operations.

Epyc Node graphs

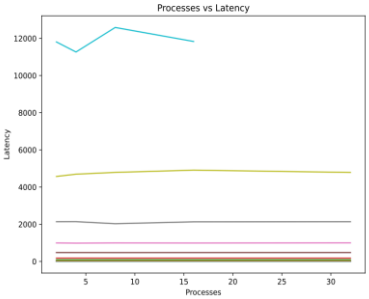
Bcast basic



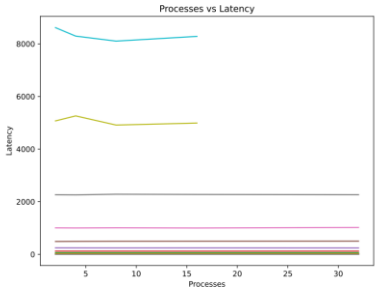
Bcast binary



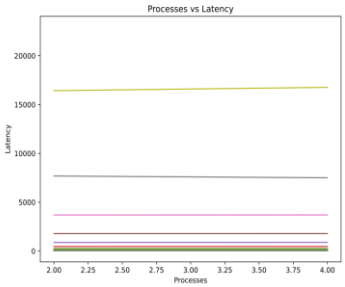
Bcast binomial



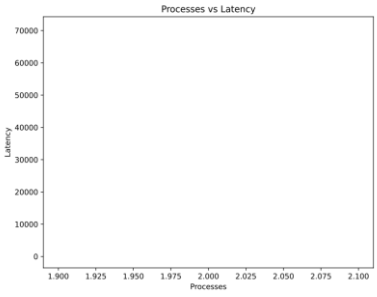
Bcast split



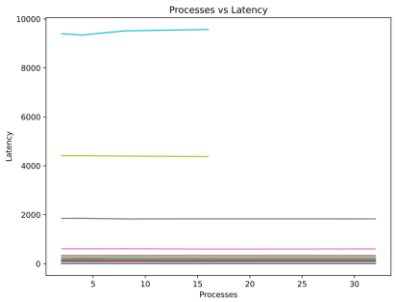
Scatter basic



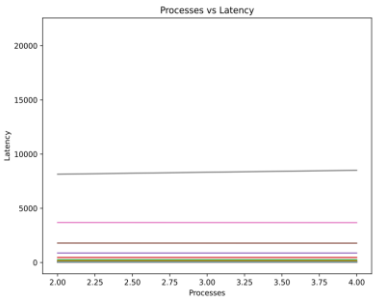
Scatter binomial



Scatter ignore

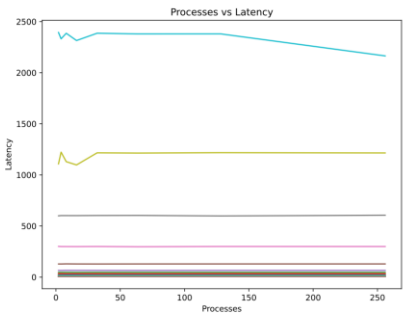


Scatter linear

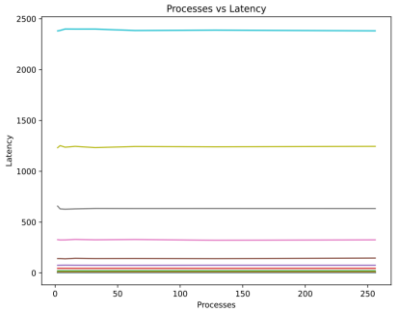


Nodo Thin

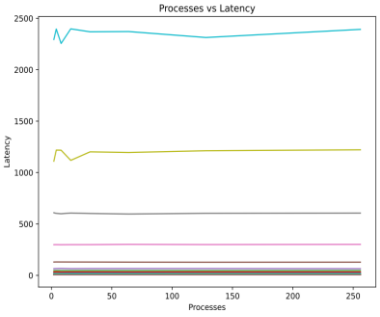
Bcast basic



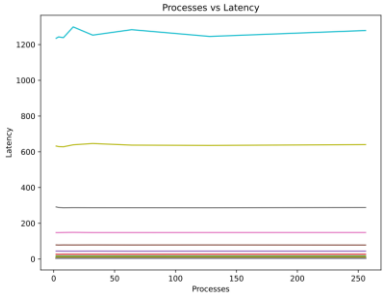
Bcast binary



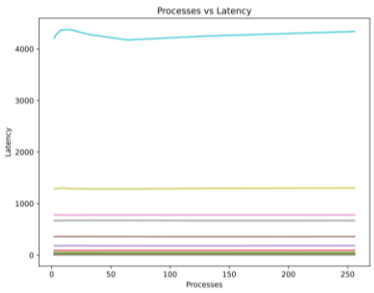
Bcast binomial



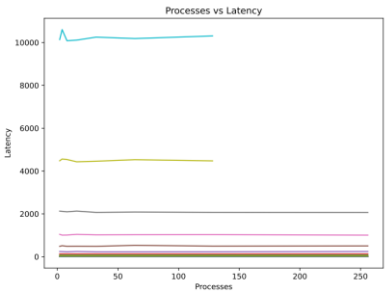
Bcast split



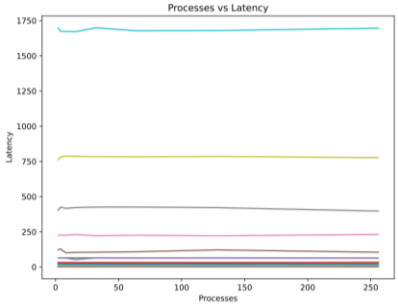
Scatter basic linear



Scatter binomial



Scatter ignore



Scatter linear nb