High-Performance Computing – Exercise 1

Comparative Analysis of OpenMPI Algorithms for Collective Operations using OSU Benchmark

Tanja Derin [SM3800013]

Data Science and Artificial Intelligence [346SM]
Supervised by Prof. Stefano Cozzini

July 2025

Introduction

Aim of the Exercise

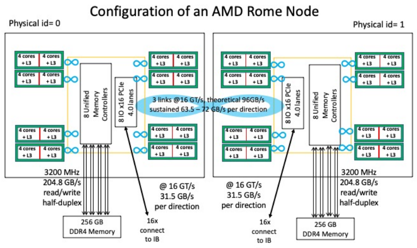
- The goal is to evaluate the performance of different OpenMPI algorithms for collective operations using the OSU Micro-Benchmark suite.
- Focus is placed on `MPI_Bcast` (mandatory) and `MPI_Reduce` (selected), analyzing how latency scales with message size and process count.
- To support interpretation, simple analytical models are developed based on measured point-to-point latencies.

Presentation Structur

- System architecture (EPYC partition of ORFEO)
- Benchmarking setup and parameters
- Results for `MPI_Bcast` and `MPI_Reduce`
- Point-to-point latency and its use in modeling
- Analytical models for broadcast and reduce algorithms

System Architecture

- All benchmarks were executed on the **EPYC partition** of the ORFEO HPC cluster.
- Each node hosts:
  - 2× AMD EPYC 7H12 CPUs (Rome architecture)
  - 128 cores
  - 8 NUMA domains per node
- Architecture impacts communication latency (CCX, CCD, NUMA, inter-node).


Configuration of an AMD Rome Node
Physical id= 0
Physical id= 1
4 cores + L3
8 Unified Memory Controllers
8 IO x16 PCIe 4.0 lanes
3 links @16 GT/s, theoretical 96GB/s sustained 63.5 – 72 GB/s per direction
3200 MHz 204.8 GB/s read/write half-duplex
@ 16 GT/s 31.5 GB/s per direction
16x connect to IB
256 GB DDR4 Memory

Block diagram of AMD EPYC Rome architecture.

Benchmarking Methodology

- Benchmarks executed on 2 full EPYC nodes using SLURM with `--exclusive` flag.
- MPI library: `OpenMPI 4.1.6`; Benchmark suite: `OSU Micro-Benchmarks 7.4`.
- Process placement enforced with `--map-by core` to minimize NUMA effects.

Execution Strategy:

- Message sizes: powers of two from `2B` to `1MiB`.
- Process counts: `2 to 256`.
- Algorithm variants selected via `coll_tuned` module.

Post-processing:

- Latencies extracted from stdout with `awk`.
- Results saved to CSV for plotting and model comparison.

Broadcast Benchmark Results

Tested Algorithms:

- Pipeline
- Binomial Tree
- Chain

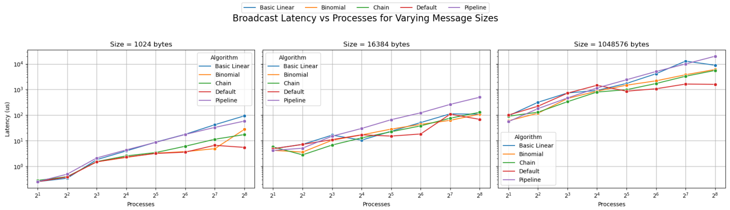
Evaluation Dimensions:

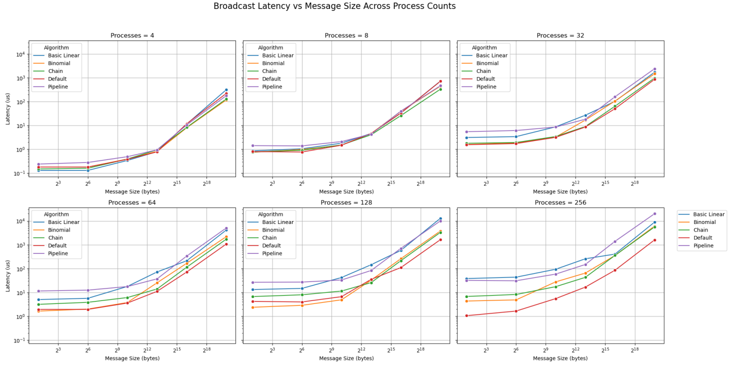
- Latency vs Process Count (for fixed message sizes)
- Latency vs Message Size (for fixed process counts)

Observations:

- Default algorithm generally outperforms others across message sizes and process counts.
- Binomial Tree and Chain show similar, consistently low latency.
- Pipeline and Linear scale poorly, especially beyond 64+ processes.

Broadcast Latency: Results





Reduce Benchmark Results

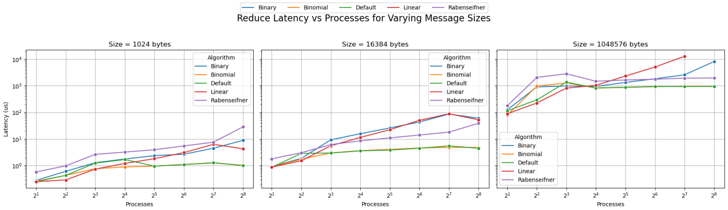
Tested Algorithms:

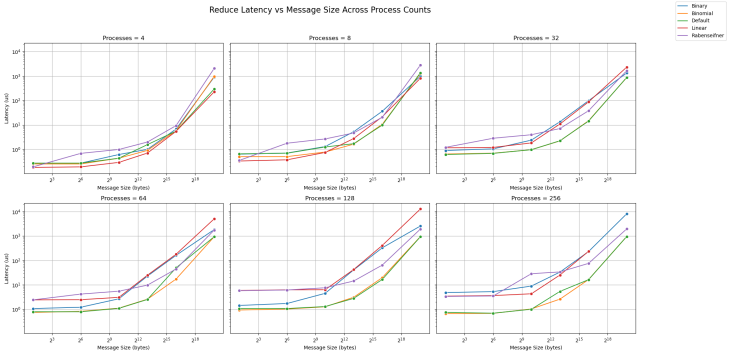
- Binomial Tree
- Rabenseifner
- Binary

Observations:

- **B**inomial and Default show best scalability.
- Large messages Rabenseifner improves noticeably.
- Linear and Binary degrade beyond 16 processes.
- Default implementation balances performance well across sizes.

Reduce: Latency Results





Point-to-Point Latency

- Captures architectural bottlenecks (CCX, CCD, NUMA, sockets, inter-node)
- To model collective algorithms using realistic communication costs.

Method:

- Used `osu_latency` to measure time between core 0 and selected targets.
- Created latency matrix $L(i, j)$ to represent communication cost between any two processes.

Example Measurements (2-byte messages):

- Same CCX (Core 1): 0.15 μs
- same CCD (Core 4): 0.32 μs
- Same NUMA (Core 8): 0.35 μs
- Cross NUMA (Core 32): 0.43 μs
- Cross socket (Core 64): 0.69 μs
- Inter-node (Core 128): 1.82 μs

Broadcast Algorithm Models

`MPI_Bcast` distributes a message from a root process to all others.

- **Pipeline**: message forwarded in sequence.

$$T_{\text{pipeline}}(P) = \sum_{i=0}^{P-2} \text{latency}(i \to i+1)$$

- **Binomial Tree**: root sends in $\log_2 P$ steps to ranks 2^i.

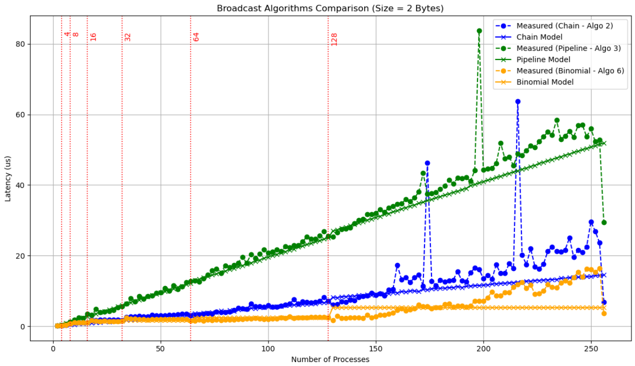
$$T_{\text{binomial}}(P) = \sum_{i=0}^{\log_2 P-1} \text{latency}(0 \to 2^i)$$

- **Chain (Parallel)**: processes divided into f chains from root.

$$T_{\text{chain}}(P,f) = \max_{j=1,\dots,f} \left\{ \text{latency}(0 \to p_{j,1}) + \sum_{k=2}^{n_j} \text{latency}(p_{j,k-1} \to p_{j,k}) \right\}$$

Broadcast Models vs Measurements

- **Pipeline**: Latency grows linearly with process count. Model fits well.
- **Binomial Tree**: Lowest latency and best scalability. Matches measurements closely.
- **Chain (Parallel)**: Outperforms Pipeline, especially at scale. Spikes likely due to runtime variability.



Reduce Algorithm Models

`MPI_Reduce` aggregates values from all processes to a designated root.

- **Binary Tree**: reduction proceeds in $\log_2 P$ steps.

$$T_{\text{binary}}(P) = \sum_{i=0}^{\log_2 P - 1} \max_r \left\{ \text{latency}(r + 2^i \to r) \right\}$$

- **Binomial Tree**: rank 2^i sends to root at each step.

$$T_{\text{binomial}}(P) = \sum_{i=0}^{\log_2 P - 1} \text{latency}(2^i \to 0)$$

Reduce Algorithm Models

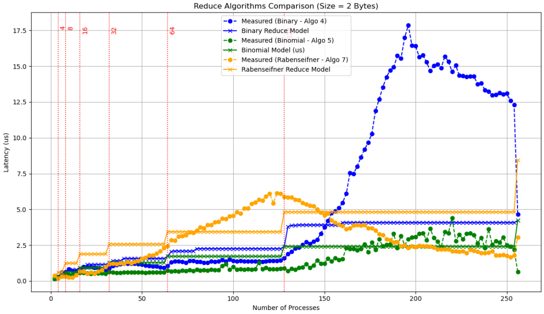
`MPI_Reduce` aggregates values from all processes to a designated root.

- **Rabenseifner**: combines reduce-scatter and gather phases.

$$T_{\text{rabenseifner}}(P) = \sum_{i=0}^{\log_2 P-1} \max_r \left\{\text{latency}(r \to r+2^i)\right\} + \sum_{i=0}^{\log_2 P-1} \max_r \left\{\text{latency}(r+2^i \to r)\right\}$$

Reduce: Model vs Measurement

- **Binomial Tree** shows the best agreement and lowest latency mostly
- **Binary Tree** performs worse at scale, latency increases sharply beyond 128 processes.
- **Rabenseifner** matches measured data in some points



High-Performance Computing – Exercise 2c

Hybrid Parallel Mandelbrot Set Computation: MPI + OpenMP Scaling Analysis

Tanja Derin [SM3800013]

Data Science and Artificial Intelligence [346SM]
Supervised by Prof. Luca Tornatore

July 2025

Introduction: Mandelbrot Set Computation

The **Mandelbrot set** is a complex fractal defined by the iterative function:

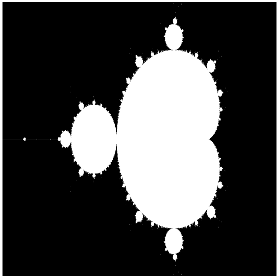
$$f_c(z) = z^2 + c \quad \text{with} \quad z_0 = 0,\ c \in \mathbb{C}$$

A point c belongs to the set if the sequence $\{z_n\}$ remains bounded ($|z_n| \leq 2$) after a finite number of iterations. A cutoff $I_{\max}$ is used in practice.
each pixel can be computed independently

This project implements a **hybrid MPI + OpenMP** approach:

- **MPI** handles domain decomposition across nodes (process-level parallelism)
- **OpenMP** parallelizes pixel computations within each process within a node
- Outputs are saved as `.pgm` images

Goal: Evaluate the effectiveness and scalability of hybrid parallelism on a multi-node HPC cluster Orfeo.


Example output

MPI Domain Decomposition

Strategy: Row-wise decomposition of the 2D image domain across MPI processes.

When rows don't divide evenly, the extra rows are assigned to the first processes.

```
int rows_per_process = ny / size;
int remainder = ny % size;
int start_row = rank * rows_per_process + (rank <
   remainder ? rank : remainder);
int local_rows = rows_per_process + (rank < remainder
   ? 1 : 0);
```

Parallel File Output with MPI-IO

Each process writes its image block directly using `MPI_File_write_at`.
Key steps:

- Rank 0 writes the PGM header
- All ranks synchronize using `MPI_Barrier()`
- Each rank writes its block at a calculated file offset

Listing: MPI parallel write after header

```
MPI_Offset offset = header_size + (MPI_Offset)(rank *
    local_rows * xsize);
MPI_File_write_at(file, offset, image, local_rows *
    xsize, MPI_UNSIGNED_CHAR, &status);
```

OpenMP Thread-Level Parallelism

Within each process:

- OpenMP parallelizes the loop over image rows
- Dynamic scheduling improves load balance

Listing: Parallel pixel computation with OpenMP

```
#pragma omp parallel for schedule(dynamic)
for (int j = 0; j < local_rows; j++) {
    for (int i = 0; i < nx; i++) {
        double cx = xL + i * dx;
        double cy = yL + (start_row + j) * dy;
        int iter = compute_mandelbrot(cx, cy, Imax);
        local_image[j * nx + i] = (iter == Imax ? 255
           : 0);}}
```

Test Setup

Test Environment: ORFEO Cluster – EPYC Partition

- OpenMP: up to 128 threads on 1 node
- MPI: up to 256 processes across 2 nodes

Scalability Formulas
Amdahl's Law – Strong Scaling

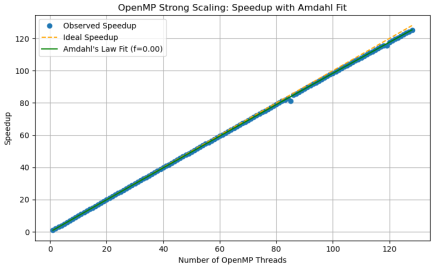
$$S(p) = \frac{1}{f + \frac{1-f}{p}}, \qquad E(p) = \frac{S(p)}{p}$$

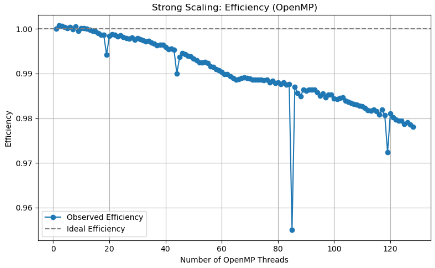
Gustafson's Law – Weak Scaling

$$S(p) = p - f \cdot (p-1), \qquad E(p) = \frac{S(p)}{p}$$

- f: serial fraction of the workload
- p: number of processes or threads
- Models fitted to empirical performance data

OpenMP Strong Scaling

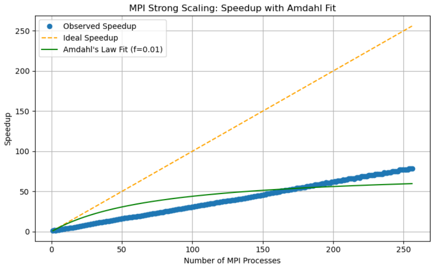


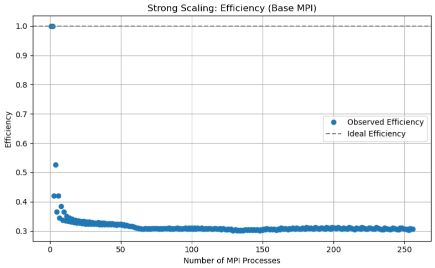


Observations:

- OpenMP shows near-linear speedup, closely matching the ideal speedup line
- The Amdahl's Law fit yields a serial fraction $f = 0.00$, suggesting that the workload is almost entirely parallelizable
- The efficiency remains very high up

MPI Strong Scaling

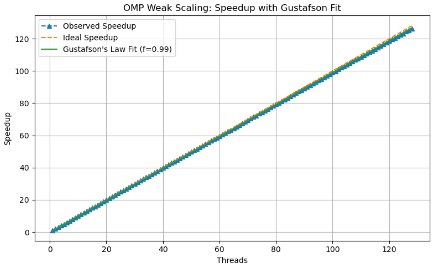


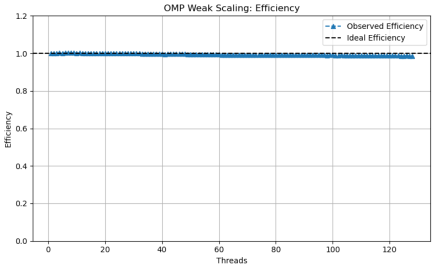


Observations:

- Sublinear speedup with growing number of processes
- Amdahl fit reveals serial fraction $f \approx 0.01$
- Efficiency drops from near to 30%
- Bottlenecks from imbalance, communication, synchronization

OpenMP Weak Scaling





Observations:

– workload was scaled proportionally: each thread handled 1M pixels

– Speedup fits Gustafson's Law with $f = 0.99$

– Efficiency remains high even up to 128 threads, indicating excellent parallel scalability.

MPI Weak Scaling

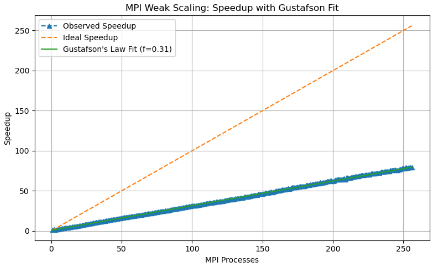


Figure: Speedup (Gustafson Fit)

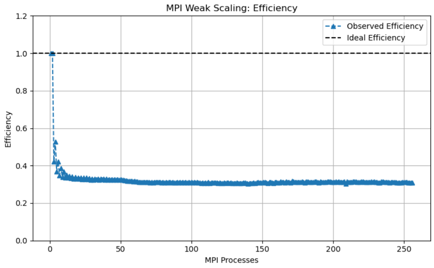


Figure: Efficiency vs. Processes

- Gustafson's Law fit yields $f = 0.31$, indicating that approximately 31% of the workload behaves serially
- Efficiency rapidly drops below 40% and approaches 30% at 256 processes.

Conclusion and Outlook

Summary of Findings:

– OpenMP achieved near-ideal speedup and efficiency within a single node.

– MPI scalability was constrained by poor load balance from static row-wise domain decomposition.

– Amdahl's and Gustafson's Laws accurately modeled performance limits.

Possible Improvements:

– Implement dynamic load balancing with master–worker scheduling.

– Use non-blocking MPI.

Hybrid parallelism is effective, but adaptive workload distribution is key to achieving high efficiency at scale.