HIGH PERFORMANCE COMPUTING

MATTEO MUNINI

10/10/2025

PROJECT TASK 2023

INDEX

- Introduction
- Software and Architecture description
- Project 1
  - Description of the tests
  - Description of algorithms compared and Results presentation
  - Discussion
- Project 2
  - The Mandelbrot set
  - Description of the two strategy
  - Code details
  - Description of type of scaling experiment
  - Results presentation
  - Discussion

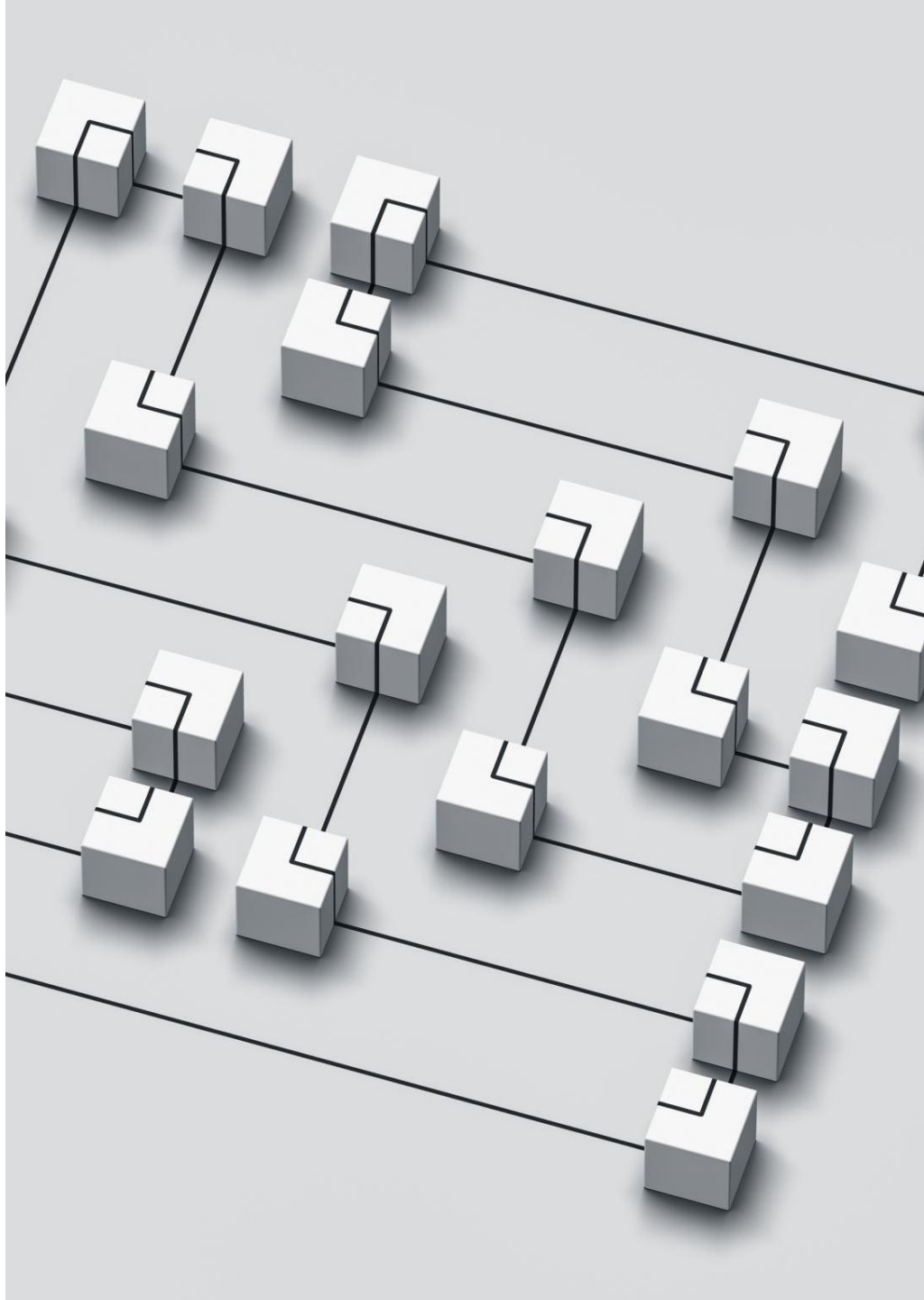
Introduction

PROJECT 1 :

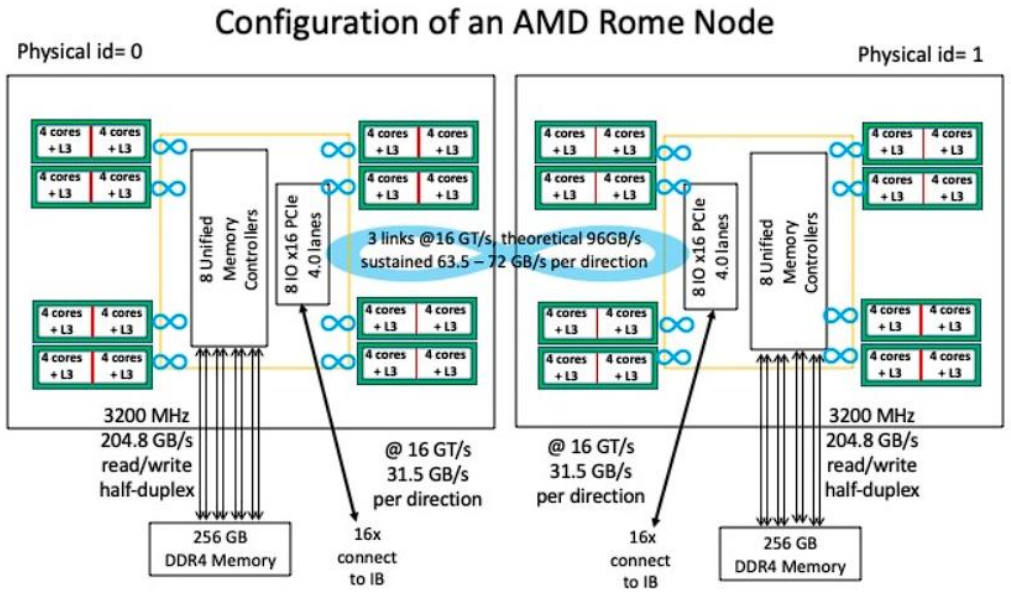
The task is about the comparison between OpenMPI algorithms for collective operation, which in this project will be ***Broadcast*** and ***Reduce***.

PROJECT 2 :

In this project it is asked to define a program which output a PMG file representing the Mandelbrot set using an hybrid MPI+OpenMP strategy.



Software and Architecture description



- *EPYC* partition of the ORFEO HPC cluster.
- This partition comprises 8 compute nodes, each equipped with two AMD EPYC 7H12 CPUs (Rome architecture), providing a total of 128 cores and 512 GiB DDR4 memory per node.
- Each *CPU* is organized into four NUMA nodes, yielding eight NUMA domains per node, and the nodes are interconnected by a 100 Gb/s high-speed network.

Software and Architecture description

- For the benchmarking experiments were executed using SLURM for job submission. The software stack included OpenMPI v4.1.6, OpenMP and OSU Micro-Benchmarks v7.4. The programming language used has been C and Python.


PROJECT 1

Description of the tests

There have been defined 2 kind of tests:

- *FIXED*: The message size if fixed, and the only change allowed is on the number of processes involved. The goal is to understand how the latency changes in this scenario.
- *VARIABLE*: Here both size of message and number of processes changes forming a grid of combination. Interesting to see the latency behavior in this space.

Processes varies from 2 to 256 (max. in 2 EPYC nodes).

It has been asked for exclusive node allocation. MPI processes were mapped by core to minimize NUMA effects and inter-node variability (process affinity).

- SBATCH --nodes=2
- SBATCH --ntasks-per-node=128
- SBATCH --exclusive, and --map-by core.

Communication Time

MPI point-to-point latencies were measured between representative core pairs spanning same

1. CCX
2. CCD
3. NUMA
4. cross-socket
5. internode distances.

Each test binds rank 0 to core 0 and rank 1 to a target most distant core,

Multiple iterations ran for statistical significance.

Size of a 2-byte for each message.

The goal is to simulate communication times between cores

Description of algorithms compared

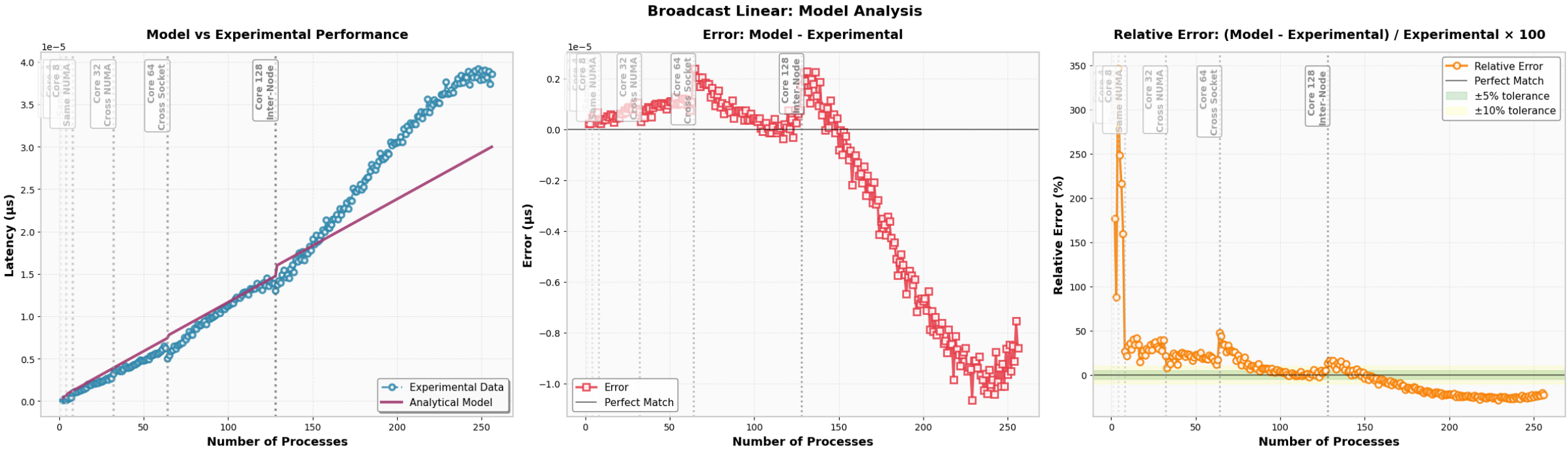
Broadcast Linear

Total communication time:

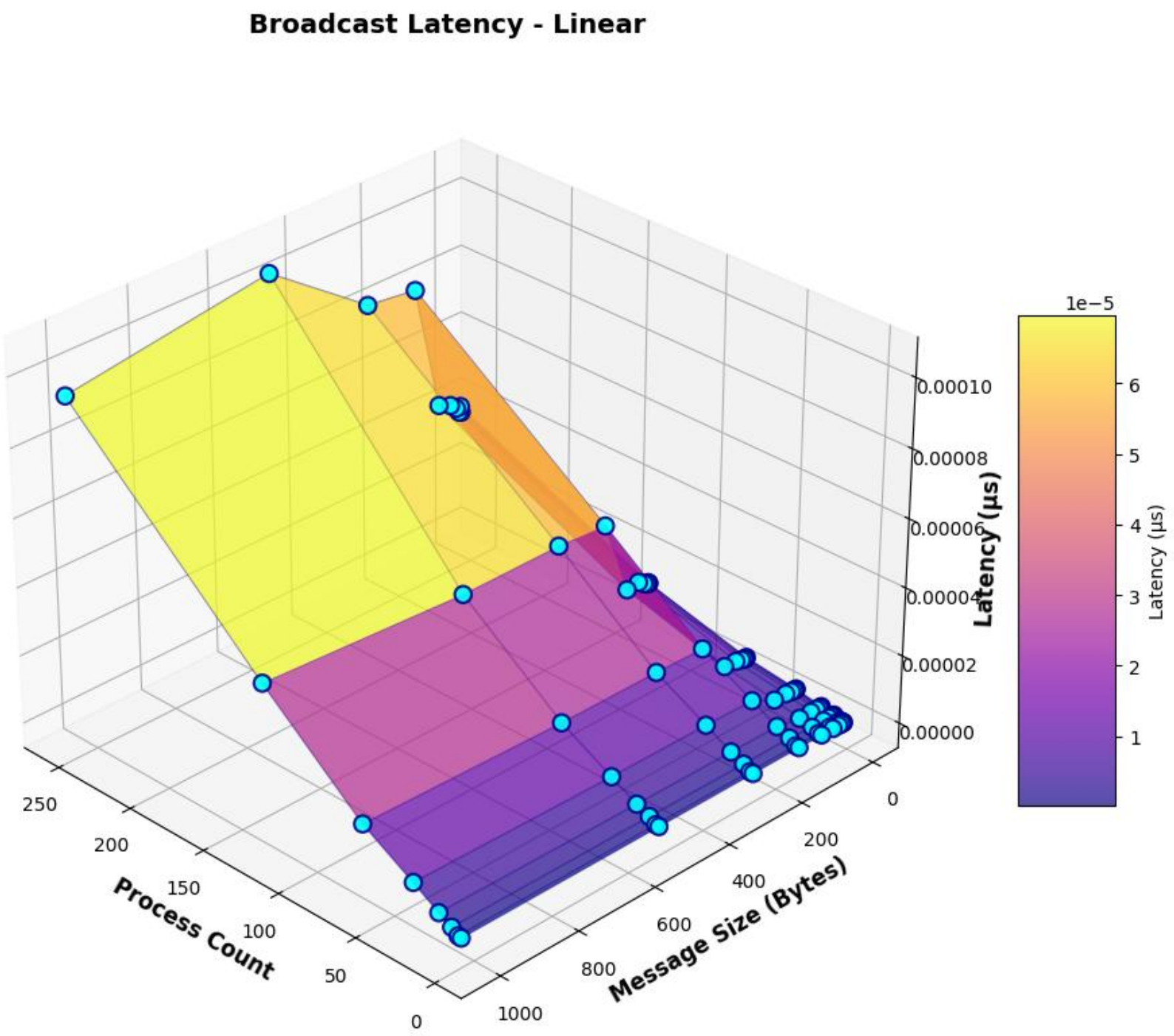
$$T(n) = \max_{p=1}^{n-1} T_{r \to p} + O \cdot n$$

where $T_{r \to p}$ is the time from root to process p, and O is the per-process overhead.

Results presentation – FIXED Broadcast



Linear Broadcast has an overall excellent match with the theoretical model until 150 processes involved, meaning that with introduction of two nodes a clear divergence appears.



The *Linear algorithm* shows the steepest behavior, meaning very affected by the inter-node links and overhead on root since n−1 processes send messages to it.

Description of algorithms compared

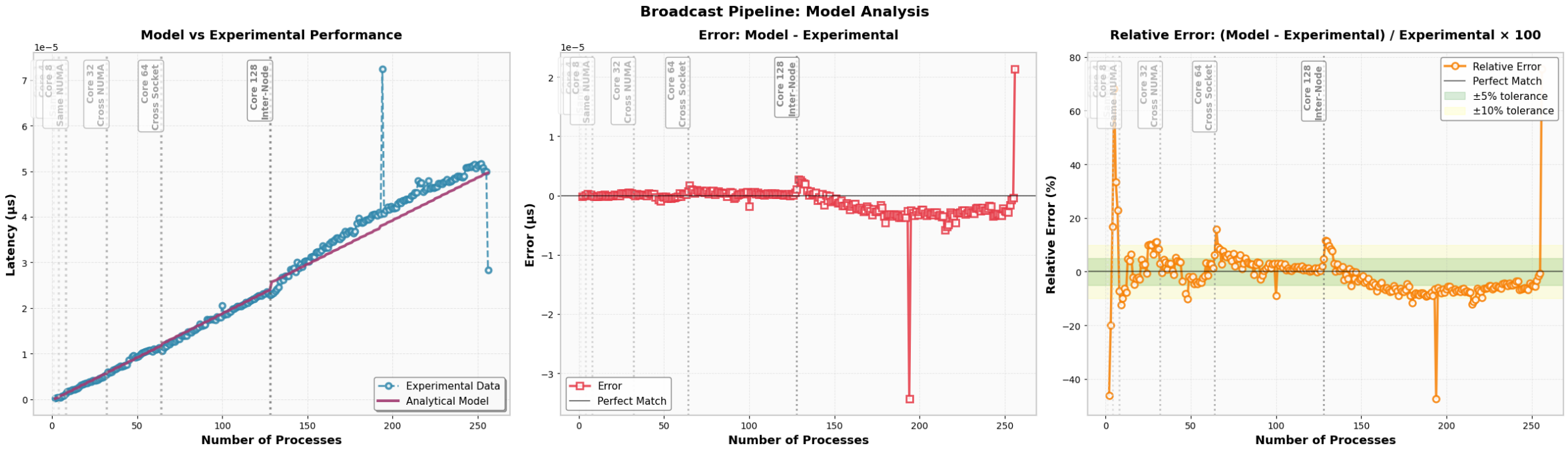
Broadcast Pipeline

Total communication time:

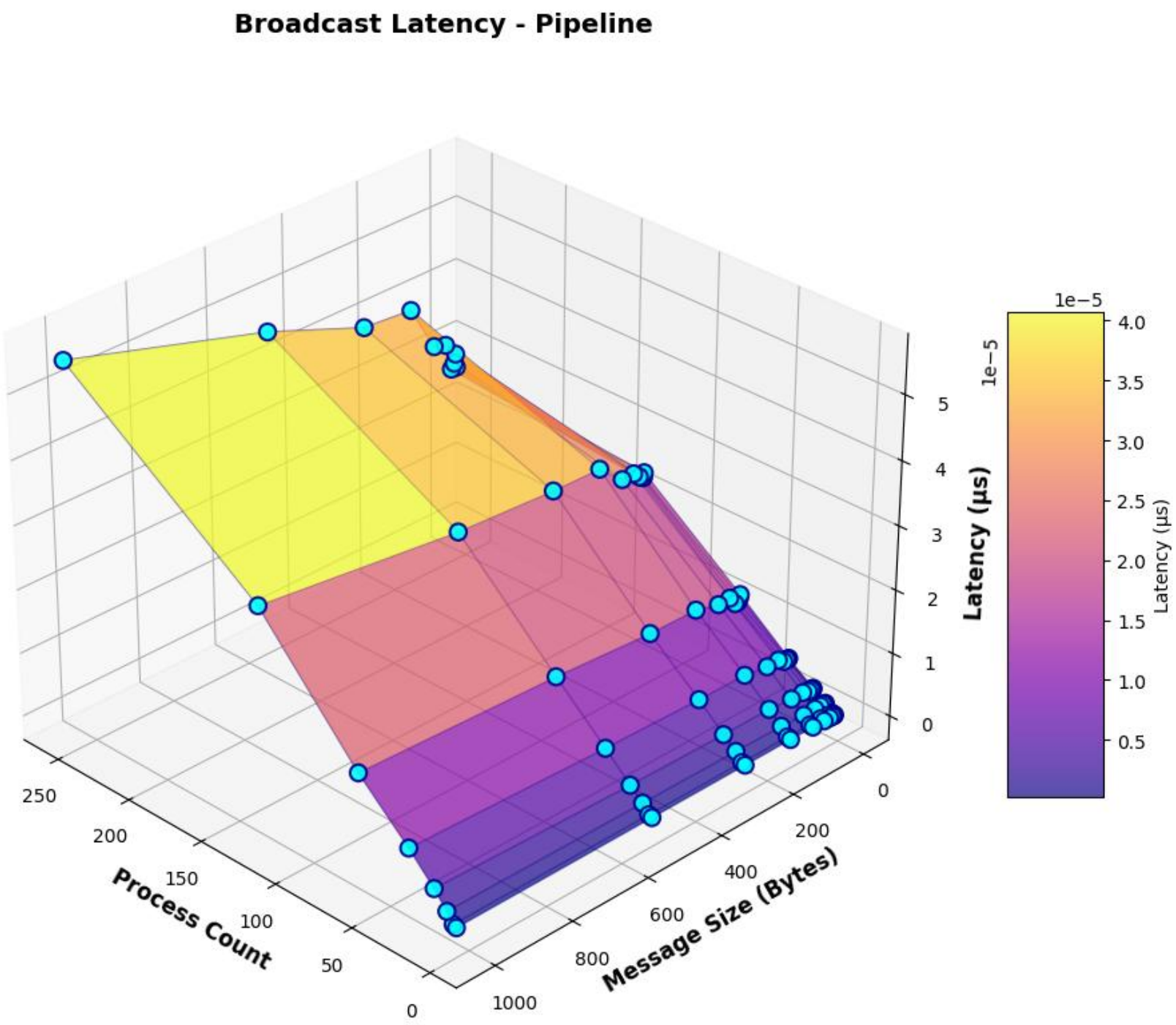
$$T(n) = \sum_{p=0}^{n-2} T_{p \to p+1}$$

where $T_{p \to p+1}$ is the time from process p to $p+1$.

Results presentation – FIXED Broadcast



Pipeline Broadcast has the best concordance between theoretical and experimental data. Only relevant is to observe the presence of some outliers.



Pipeline shows better results than Linear, and the plot shows latency increases with process count, due to sequential message forwarding.

Description of algorithms compared

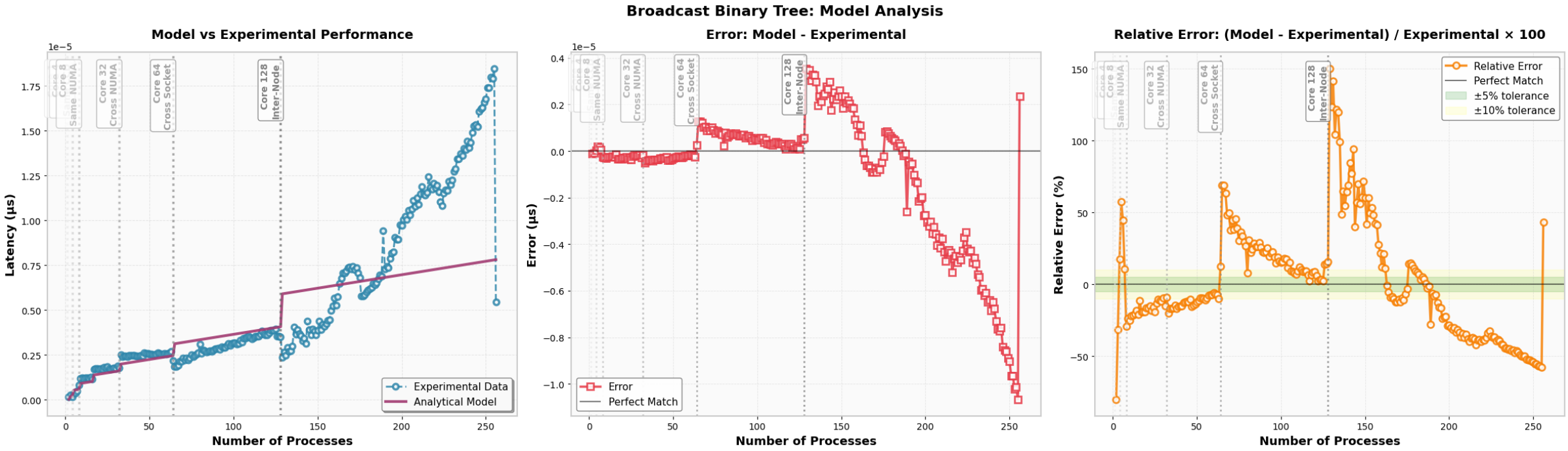
Broadcast Binary

Total communication time:

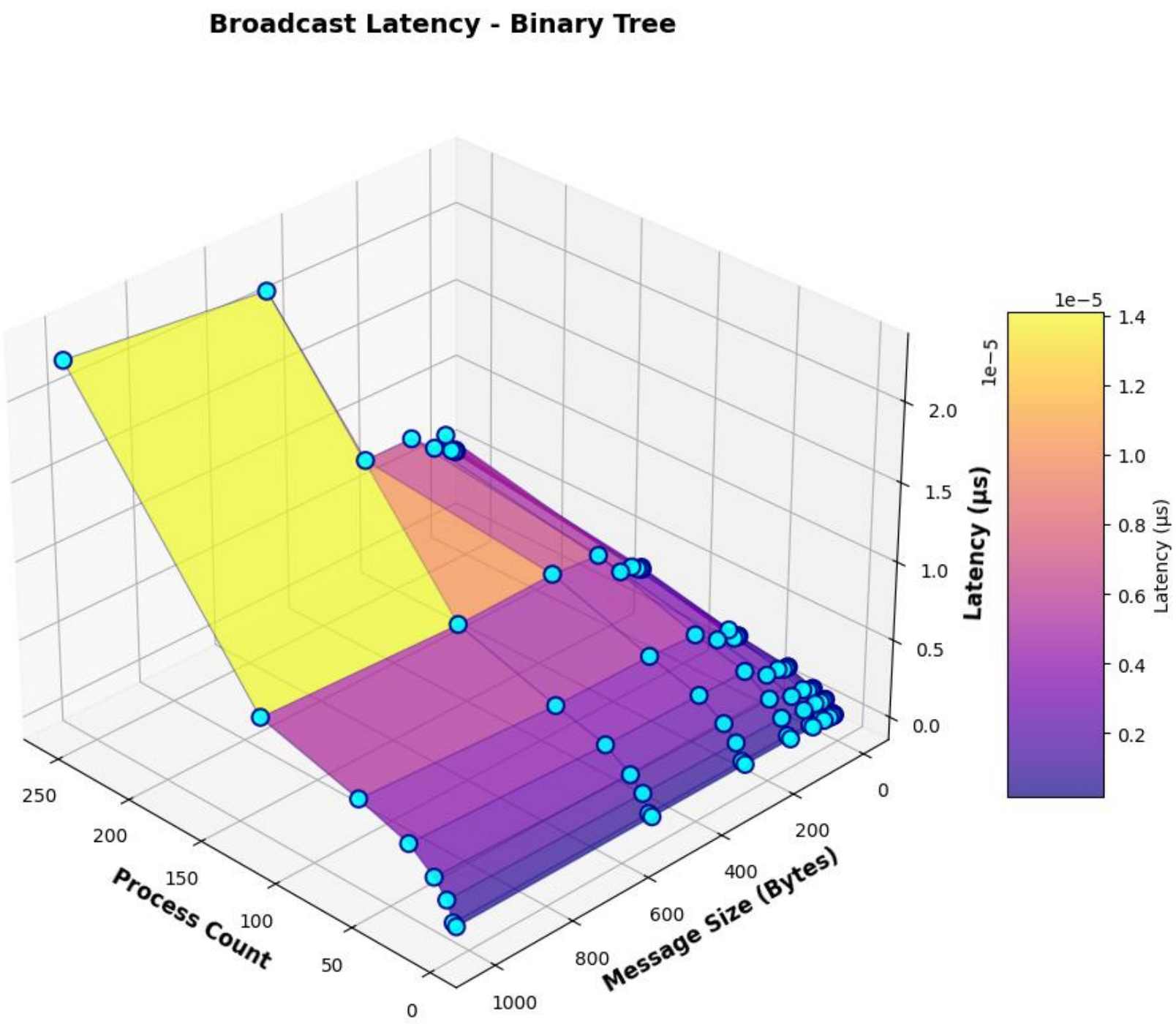
$$T(n) = \sum_{lv=1}^{height_{tree}} \max_{node \in S(lv)} T_{P(node) \to node} + O \cdot n$$

where $height_{tree} = \lceil \log_2(n) \rceil$, $S(lv)$ is the set of nodes at level lv, $P(node)$ the parent node, $T_{P(node) \to node}$ the parent-to-child time, and O the per-process overhead.

Results presentation – FIXED Broadcast Binary Tree



Binary Tree Broadcast has a good fit for low to moderate process counts, but the model struggles again when second node reached. Here, the experimental latency increases more steeply than predicted.



Binary Tree algorithm is the one that shows a generally less steep behavior across all possible grid configurations, and when considering combinations with a high number of processes and maximum message size, the latency remains the lowest in terms of order of magnitude.

Description of algorithms compared

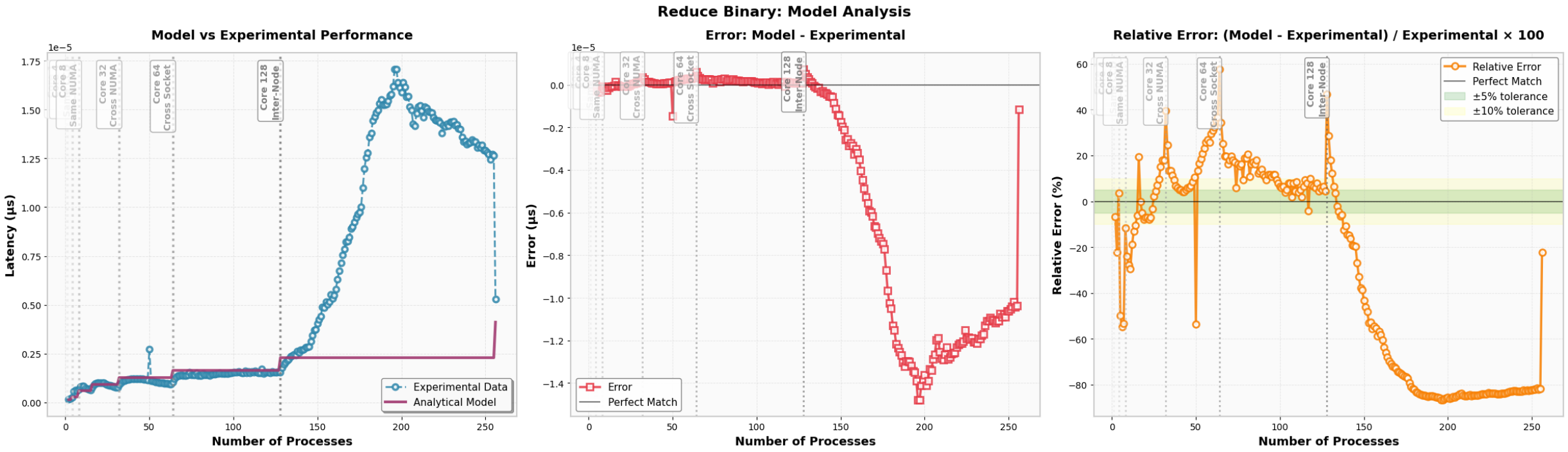
Reduce Binary

Total communication time:

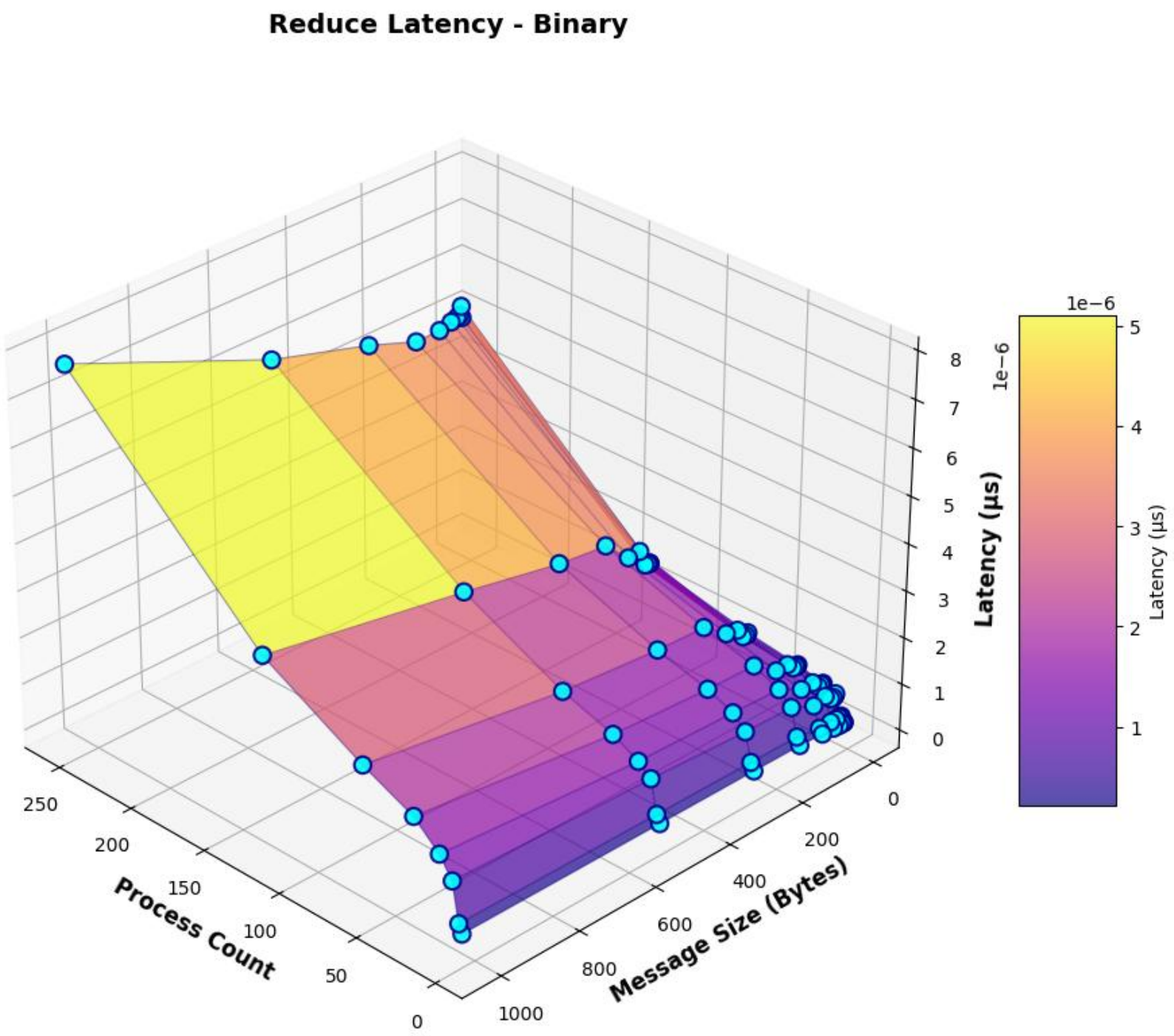
$$T(n) = \sum_{lv=0}^{\log_2 n - 1} \max_{R} \{T(R + 2^{lv} \rightarrow R)\}$$

where lv is the tree level (0 to $\log_2 n - 1$), R the receiver rank, and $R + 2^{lv}$ the sender rank.

Results presentation – FIXED Reduce Binary



Binary Reduce performances show a reasonable fit for lower process counts (<130), but shows systematic deviation afterward. Relative errors remain mainly within ±10–20% but increase significantly at higher process counts.



Binary Reduce is very steeper and it has lowest performance, since high latency over all grid tested..

Description of algorithms compared

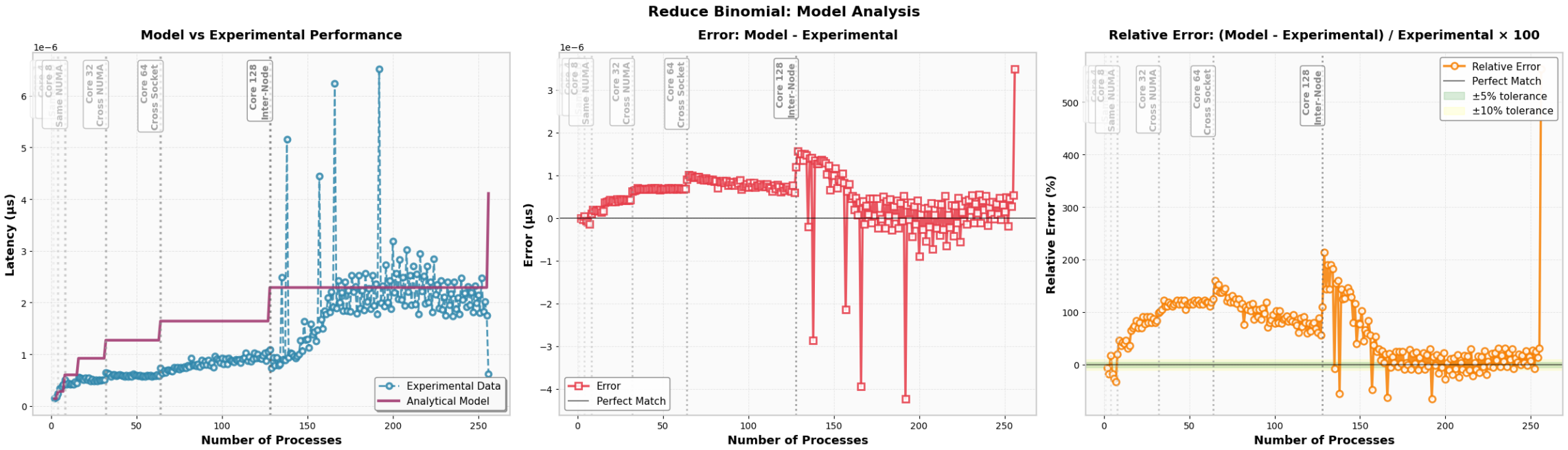
Reduce Binomial

Total communication time:

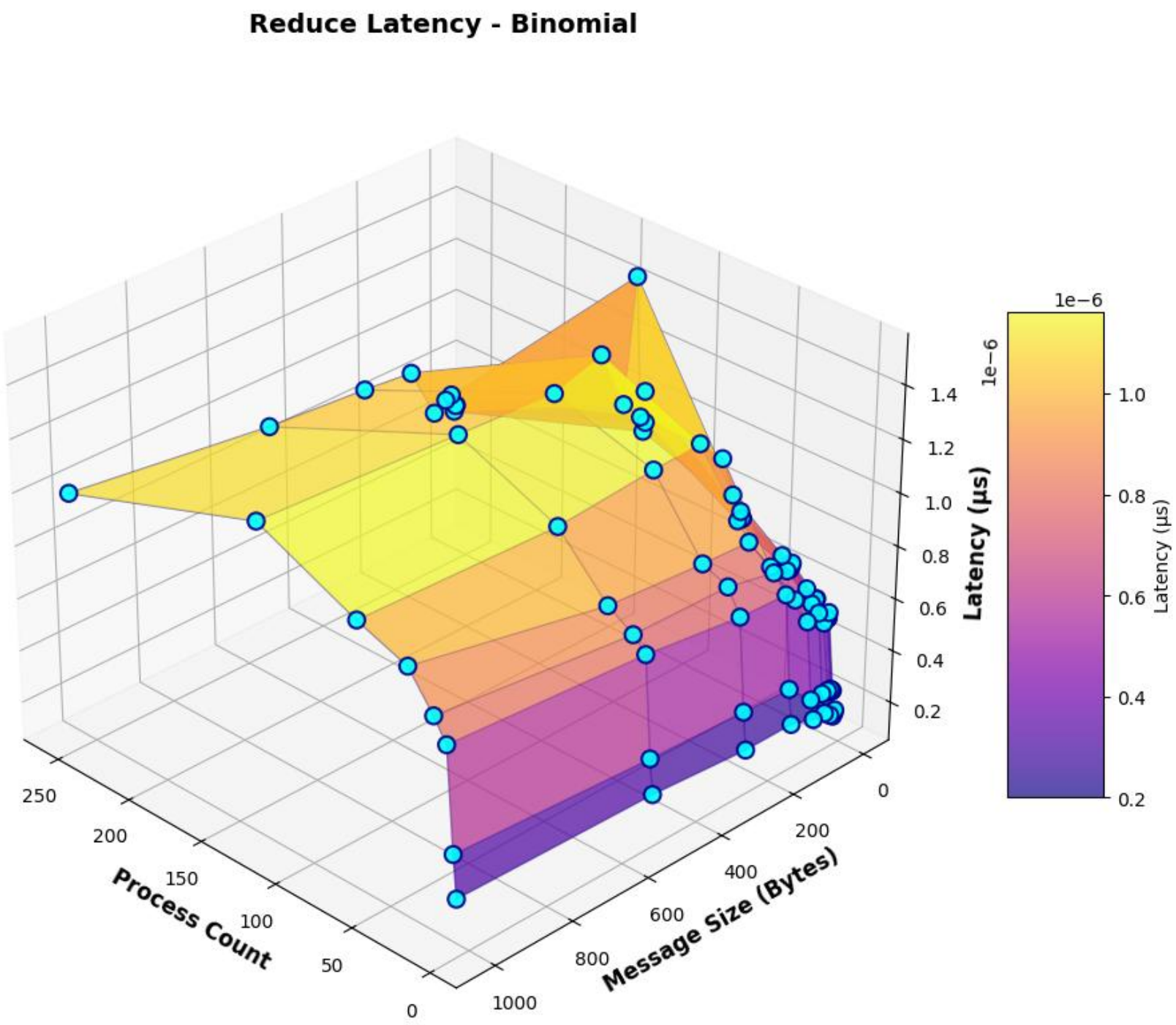
$$T(n) = \sum_{i=0}^{\log_2 n - 1} T(2^i \to 0)$$

where lv is the tree level (0 to $\log_2 n - 1$), R the receiver rank, and $R + 2^{lv}$ the sender rank.

Results presentation – FIXED Reduce Binomial



Binomial Reduce shows a concordance behavior between theoretical and experimental latency which is opposite to the others algorithms, since there is a good concordance once second node is introduced.



Reduce Binomial has the steepest behavior, however, it is the model that has lower latency in general comparing all the grid combinations with more than 50 processes involved.

Description of algorithms compared

Reduce Rabenseifner

Phase 1 – Reduce-Scatter:

$$T_{\mathrm{RS}}(n) = \sum_{lv=0}^{\log_2 n-1} \max_{R}\{T(R \leftrightarrow R + 2^{lv})\}$$

Phase 2 – Allgather:

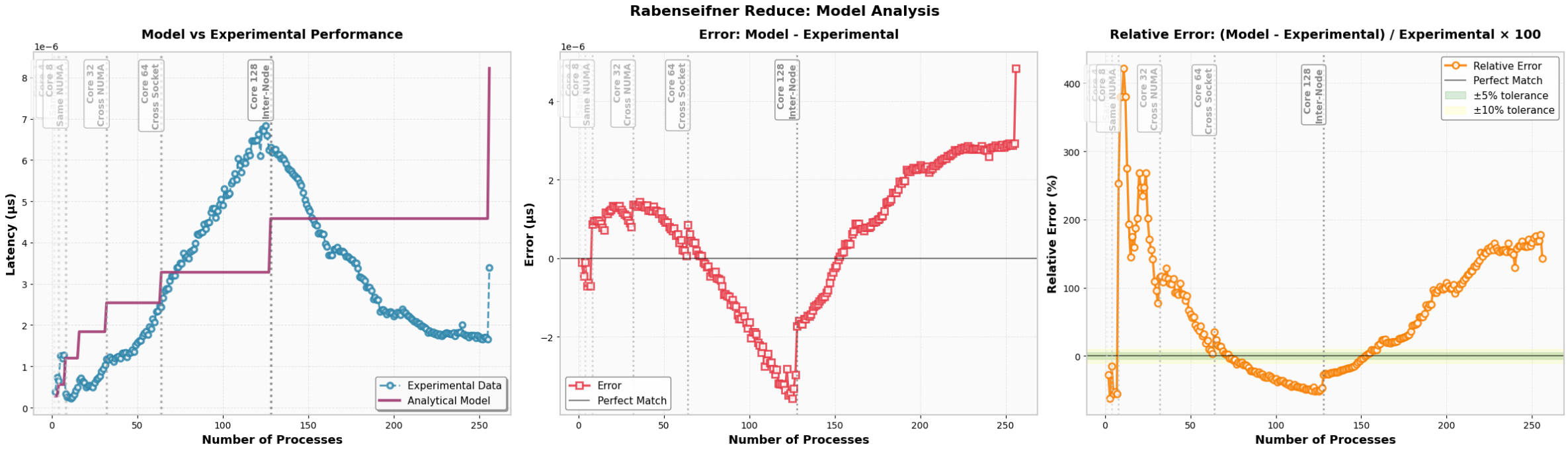
$$T_{\mathrm{AG}}(n) = \sum_{lv=0}^{\log_2 n-1} \max_{R}\{T(R + 2^{lv} \rightarrow R)\}$$

Total latency:

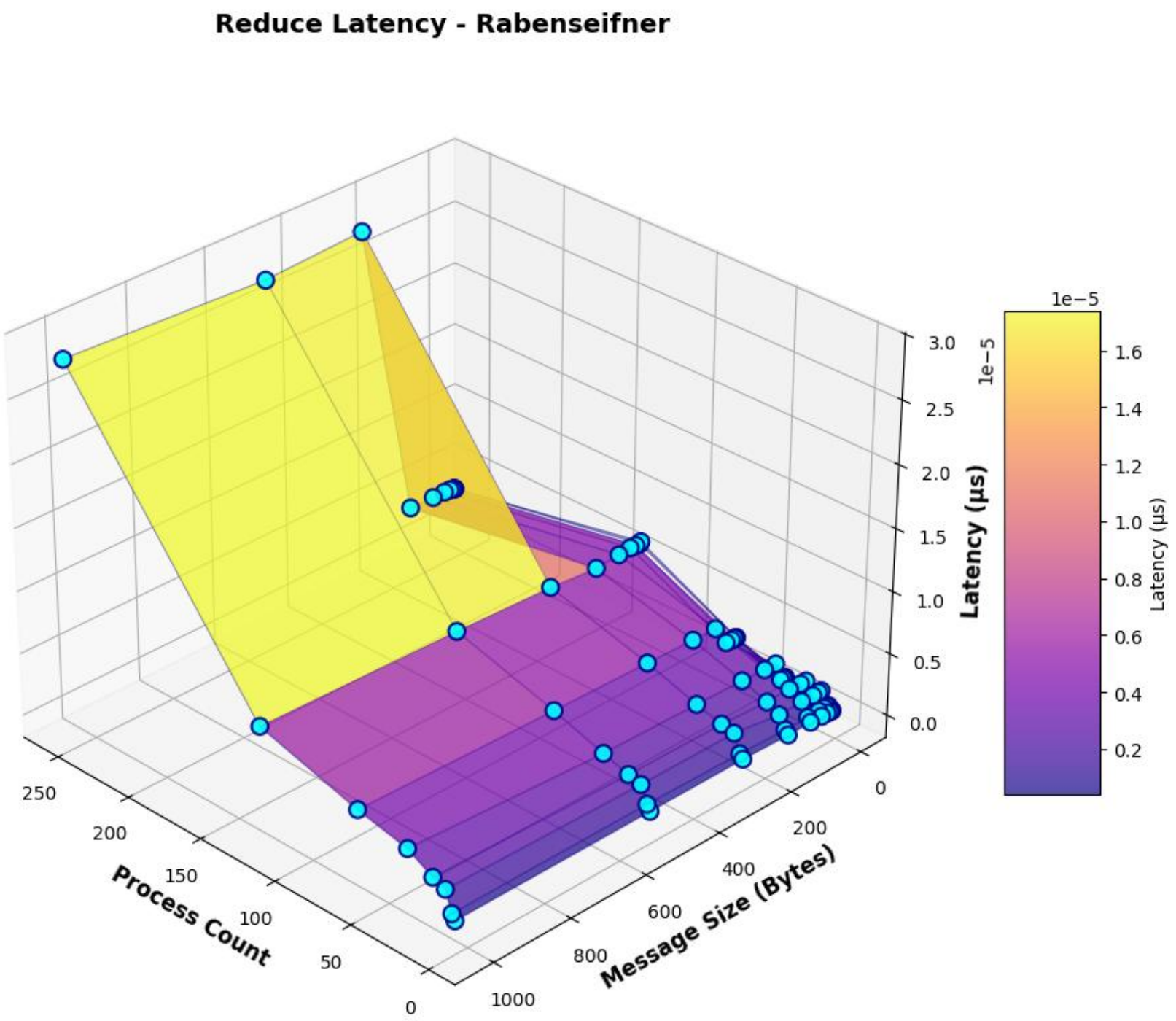
$$T(n) = T_{\mathrm{RS}}(n) + T_{\mathrm{AG}}(n)$$

where lv is the level index (0–$\log_2 n - 1$), R the communicating rank, 2^{lv} the distance, and $\leftrightarrow$ / $\rightarrow$ denote bidirectional and unidirectional exchanges.

Results presentation – FIXED Reduce Rabenseifner



Rabenseifner Reduce theoretical models underestimate latency between 64 to 128 processes range, and then it overestimate it when second node is introduced.



Rabenseifner Reduce shows the highest latency in terms of magnitude.

Discussion

- On *Broadcast Fixed* Setting's experiments there is a general low magnitude of error between experimental data and theoretical predictions.
- On *Reduce Fixed* Setting's experiments there is an higher discordance between theory and experimental data.
- The surface characteristics reveal distinct topology-related thresholds, corresponding to intra-CCD/NUMA, inter-socket, and inter-node communication domains, whose impact intensifies as the number of processes increases.
- Limitation: Size of message, only 2 nodes, not considered bandwidth, overlap and queueing terms

PROJECT 2

The Mandelbrot set

The Mandelbrot set M consists of all complex numbers c for which the sequence defined by the iteration:

$$Z_{k+1} = Z_k^2 + C, \quad Z_0 = 0, \quad C = X + iY$$

If by applying the following iteration N times c does not diverge then it is an element of the set. X-axis is real part, Y-axis is imaginary. To map the point into the image:

$$X = X_{\min} + \frac{i}{W-1}(X_{\max} - X_{\min}), \quad Y = Y_{\min} + \frac{j}{H-1}(Y_{\max} - Y_{\min})$$

The Mandelbrot set

```
static inline unsigned char compute_mandelbrot(double real_c, double imag_c, int iterations)
{
    double real_z = real_c, imag_z = imag_c;
    for (int iter = 0; iter < iterations; ++iter) {
        double real_sq = real_z * real_z, imag_sq = imag_z * imag_z;
        if (real_sq + imag_sq > 4.0) return (unsigned char)iter;
        imag_z = 2.0 * real_z * imag_z + imag_c;
        real_z = real_sq - imag_sq + real_c;
    }
    return (unsigned char)iterations;
}
```

Description of the two strategy

- Hybrid two-level parallelization strategy with *MPI* for distributed memory across ranks and *OpenMP* for shared memory within each rank.
- Rank 0 handles all file I/O, while ranks *1* to *n* focus solely on computation:
  - ensuring clear separation between computation and I/O (centralized approach)
  - simplifies debugging
  - guarantees correct PMG output formatting
  - facilitates load-balanced data collection.
  - However, it is important to highlight some ***cons***, indeed it may introduce bottlenecks at rank 0, such as memory saturation, idle processes during I/O, or network congestion during data aggregation

Code Details

Initializes the MPI environment (must be 1°)

```
MPI_Init(&argc, &argv);
int process_id, total_processes;
MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
MPI_Comm_size(MPI_COMM_WORLD, &total_processes);
```

The MPI Comm rank function assigns each process a unique identifier (rank) ranging from 0 to n-1

The MPI Comm size function retrieves the total number of processes running in parallel, stored in total processes, providing each process with knowledge of the overall system size.

```
double start_time = MPI_Wtime();
// ... computation ...
double computation_time = MPI_Wtime() - start_time;
```

parallel computation (OpenMP loops), MPI communication, synchronization overhead (barriers), and load imbalance (waiting for the slowest process)

```
int base_strips = img_height / total_processes;      // ← height
int extra_strips = img_height % total_processes;      // ← height
int local_strips = base_strips + (process_id < extra_strips ? 1 : 0);
int strip_offset = process_id * base_strips + (process_id < extra_strips ? process_id : extra_strips);
```

Almost evenly distributed rows (cols)
base strips: base number of rows (cols) per process,
extra strips: assigns leftover rows to the first few ranks.

Each process computes its workload with local strips and starting position using strip offset.
This ensures all rows are processed exactly once, limits imbalance to one row, and maintains contiguous memory for efficient computation and communication

Code Details

Each MPI process allocates a heap buffer of size (local_strips × width (height)) as unsigned char in row/col-major order to store grayscale pixels efficiently and free it later to prevent memory leaks.

```
unsigned char *local_buffer = (unsigned char *)malloc(local_strips * img_width);
```

```
#pragma omp parallel for schedule(dynamic)
for (int strip = 0; strip < local_strips; ++strip) {
    int global_strip = strip_offset + strip;
    double coord_y = min_y + global_strip * (max_y - min_y) / img_height; // ← Y fixed per strip
    for (int pixel = 0; pixel < img_width; ++pixel) {                     // ← iterate X
        double coord_x = min_x + pixel * (max_x - min_x) / img_width;
        local_buffer[strip * img_width + pixel] = compute_mandelbrot(coord_x, coord_y, max_iterations);
    }
}
```

Outer loop distributes rows among threads
Inner iterations map coordinates to [min x, max x] and [min y, max y]

Results are stored in the local buffer using row/col-major indexing for efficient memory access.

```
for (int proc = 0, offset = 0; proc < total_processes; ++proc) {
    int strips = base_strips + (proc < extra_strips ? 1 : 0);
    receive_sizes[proc] = strips * img_width;   // ← width
    data_offsets[proc] = offset;
    offset += strips * img_width;               // ← width
}
```

rank 0 constructs two arrays: receive sizes and data offsets, which define how data from each process fits into final image.

MPI_Gatherv collects variable-sized buffers from all processes to the root, assembling the full image from each process's local data (sent as MPI_UNSIGNED_CHAR) using receive sizes and offsets to handle uneven workloads efficiently.

```
MPI_Gatherv(local_buffer, local_strips * img_width, MPI_UNSIGNED_CHAR,   // ← width
            final_image, receive_sizes, data_offsets, MPI_UNSIGNED_CHAR,
            0, MPI_COMM_WORLD);
```

Theoretical background for evaluation

Amdahl's Law: Predicts maximum speedup for a fixed problem size. For a serial fraction $\%s$ and P processes:

$$S_A(P) = \frac{1}{\%s + \frac{1-\%s}{P}}, \quad E_A(P) = \frac{S_A(P)}{P}.$$

Gustafson's Law: Considers scaling problem size with P:

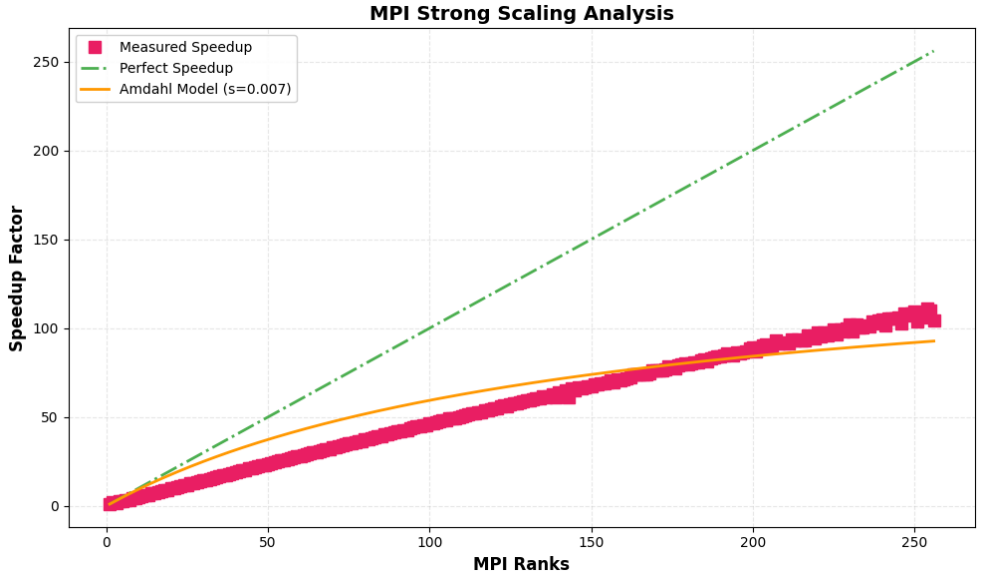
$$S_G(P) = P - \%s(P-1), \quad E_G(P) = \frac{S_G(P)}{P}.$$

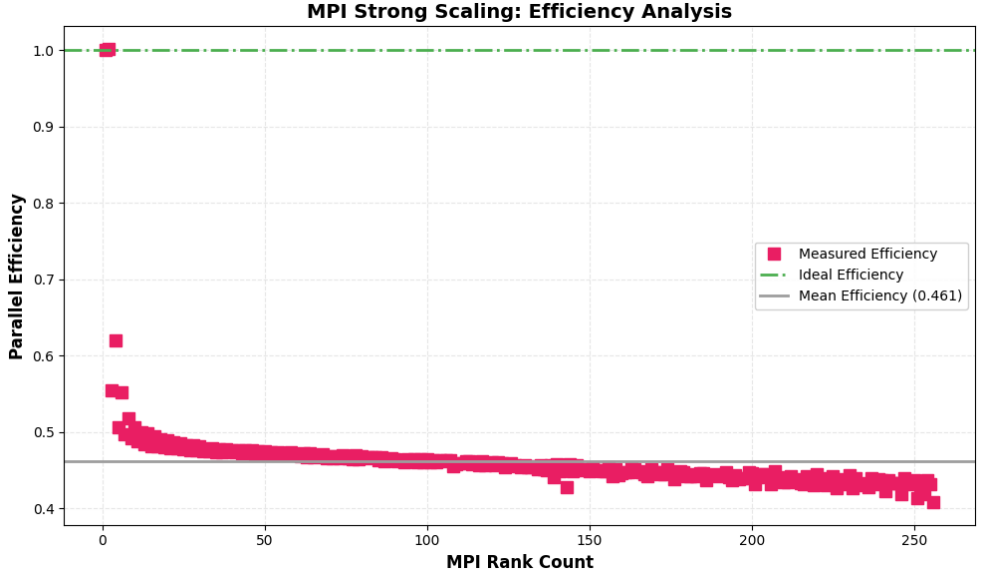
Description of type of scaling experiment

- *MPI Strong Scaling*: Up to 256 processes on 2 nodes compute subsets of a 10K×10K Mandelbrot set with OMP NUM THREADS=1 to isolate MPI performance.
- *MPI Weak Scaling*: Keeps 106 √ pixels per core, scaling the grid as CORES × 106 (1K×1K to 16K×16K pixels). Processor affinity uses --map-by core --bind-to core.
- *OpenMP Strong Scaling*: Up to 128 threads on one node for a 10K×10K problem, with OMP NUM THREADS, OMP PLACES, OMP PROC BIND and --map-by socket --bind-to none for NUMA-aware execution.
- *OpenMP Weak Scaling*: Keeps 106 pixels per thread, scaling the grid as n = √ THREADS × 106 (1K×1K to 11K×11K pixels).

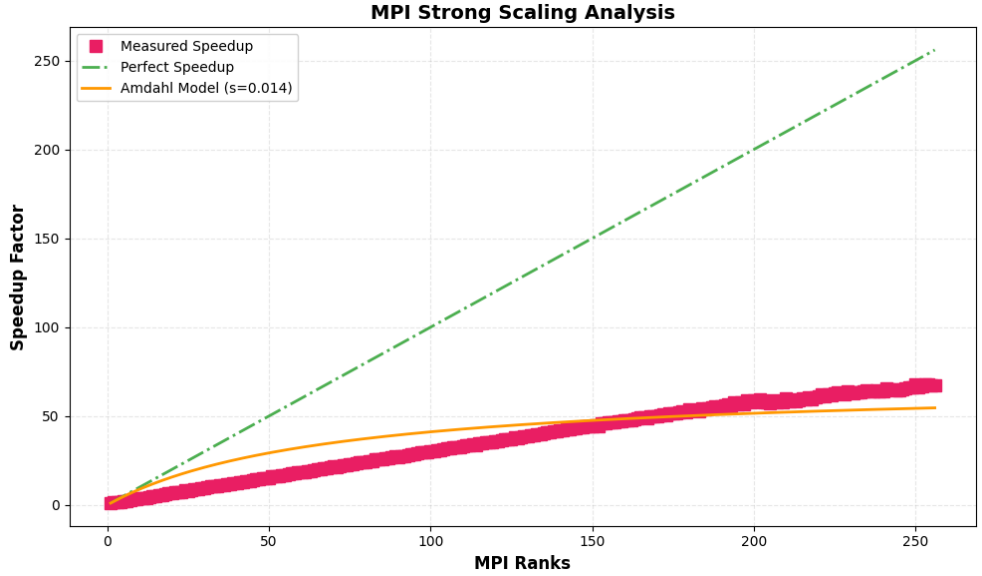
Results presentation – *MPI Strong*

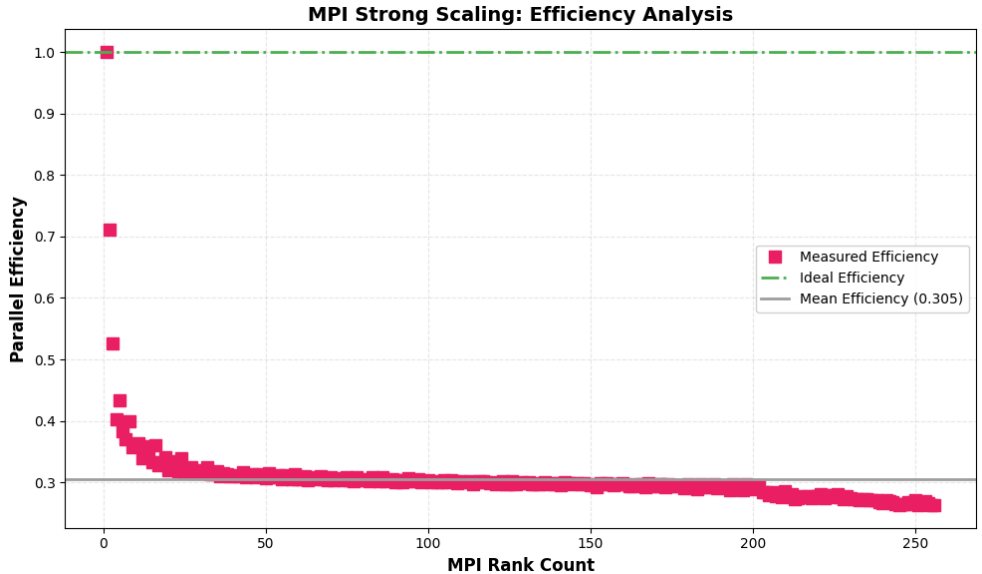
ROW





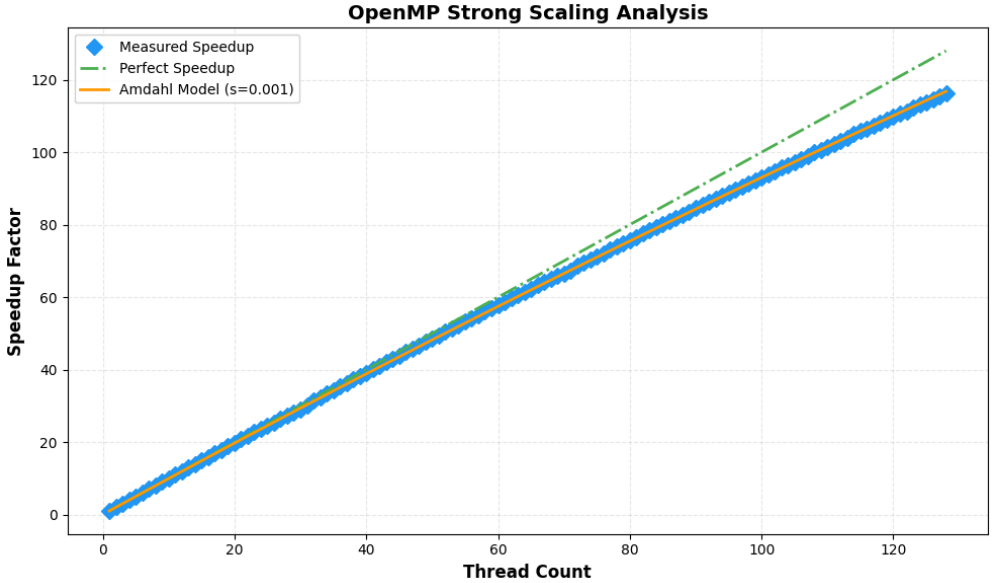
COLUMN

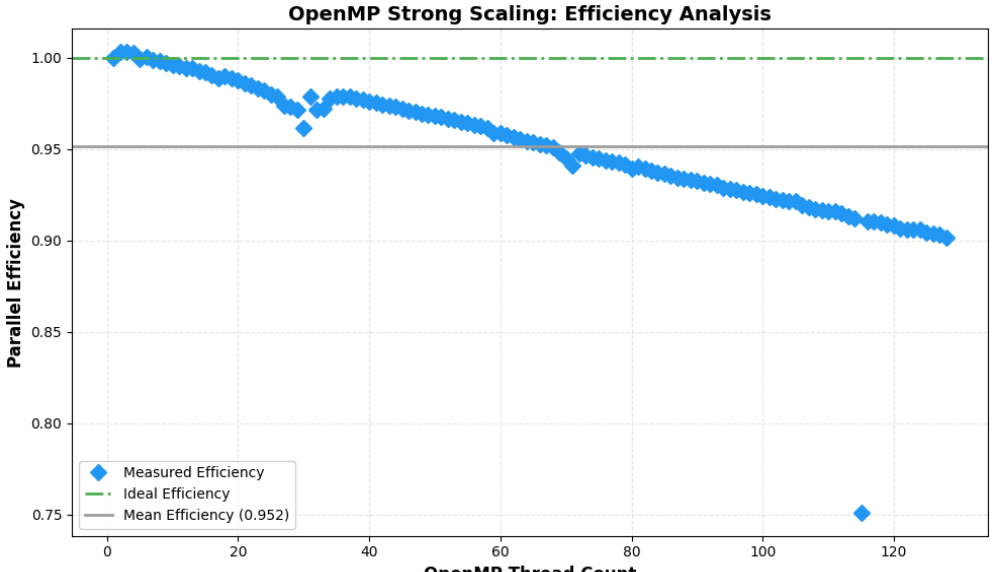




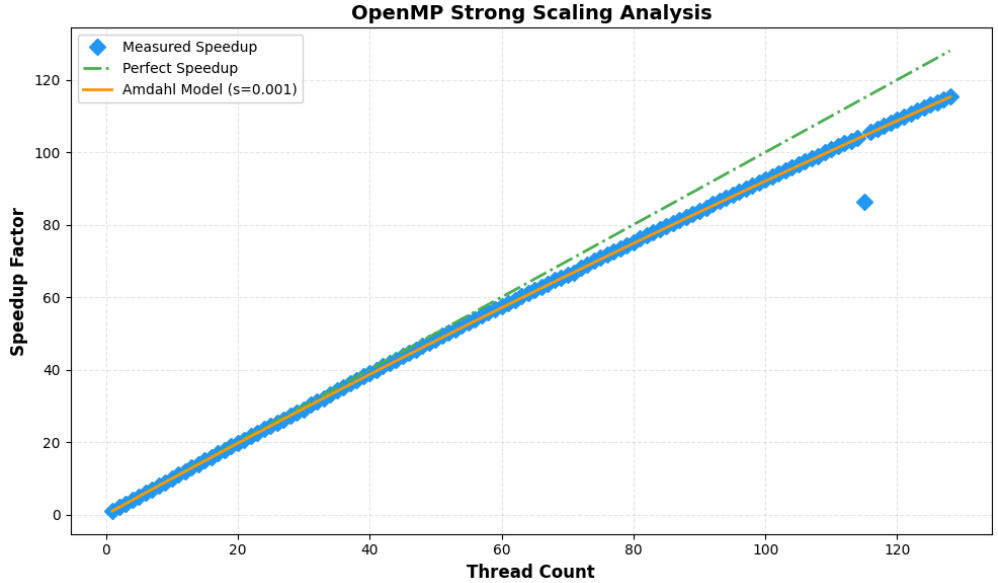
Results presentation – *OMP Strong*

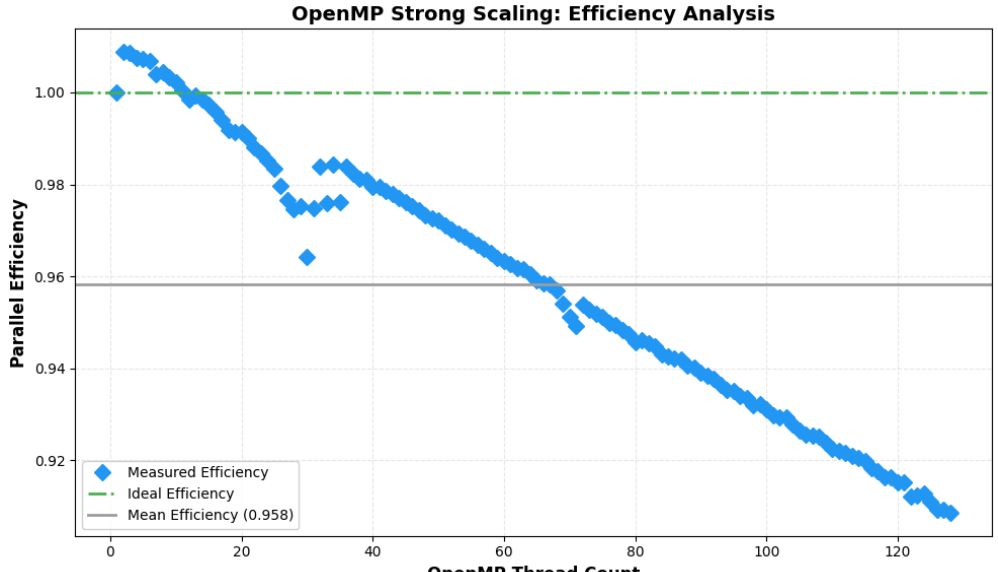
ROW





COLUMN





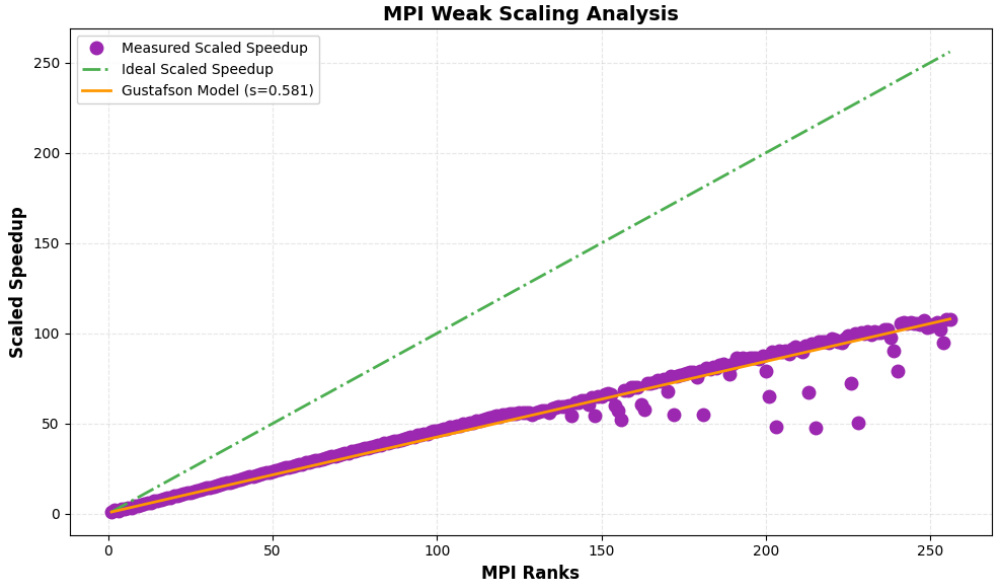
Discussion on Strong scaling

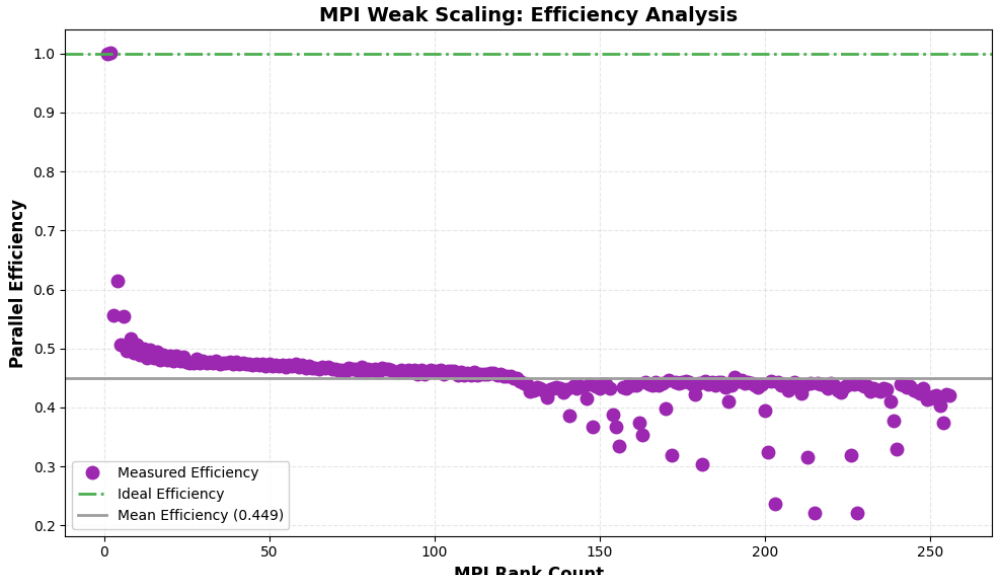
- *Row Strong Scaling Summary*:
  - MPI strong scaling is limited and there is evidence of a serial fraction due to communication and synchronization overhead.
  - OpenMP scales much better and a linear behavior of the speedup (low serial fraction).
- *Column Strong Scaling Summary*:
  - MPI strong scaling shows an average efficiency around 30.5%, showing that communication, synchronization, and load imbalance dominate performance.
  - OpenMP performs even here much better, maintaining an average efficiency up to 95%, with speedup that exhibits a linear behavior.

Overall, OpenMP achieves an higher average efficiency than MPI highlighting that memory access patterns and communication overhead, rather than algorithmic serial fractions, limit strong scaling at high processor counts.

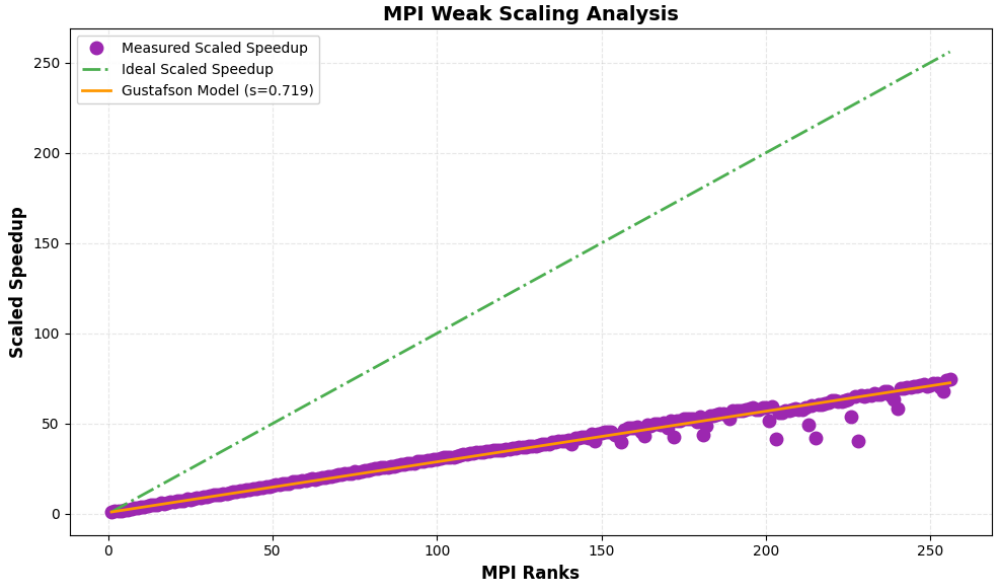
Results presentation – *MPI Weak*

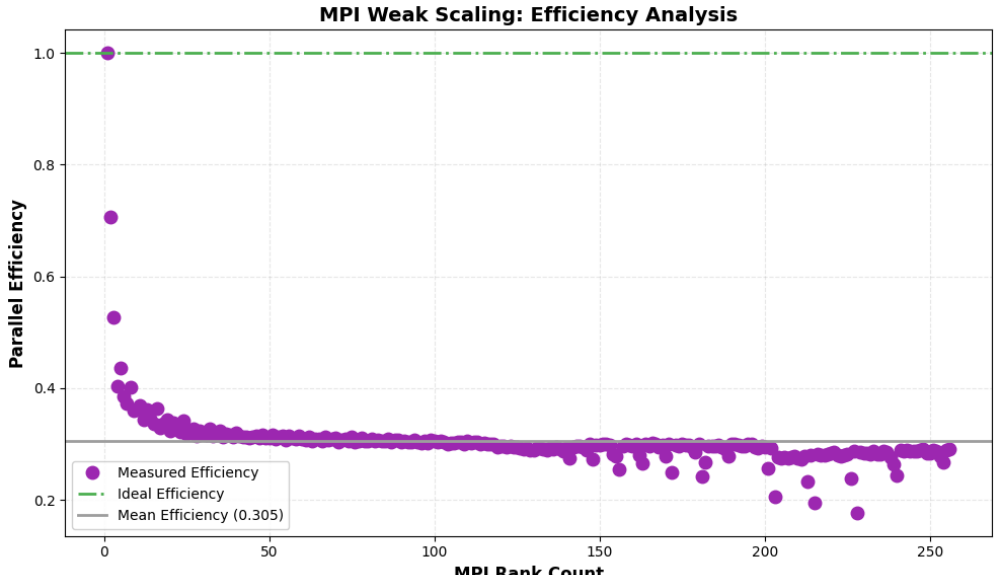
ROW





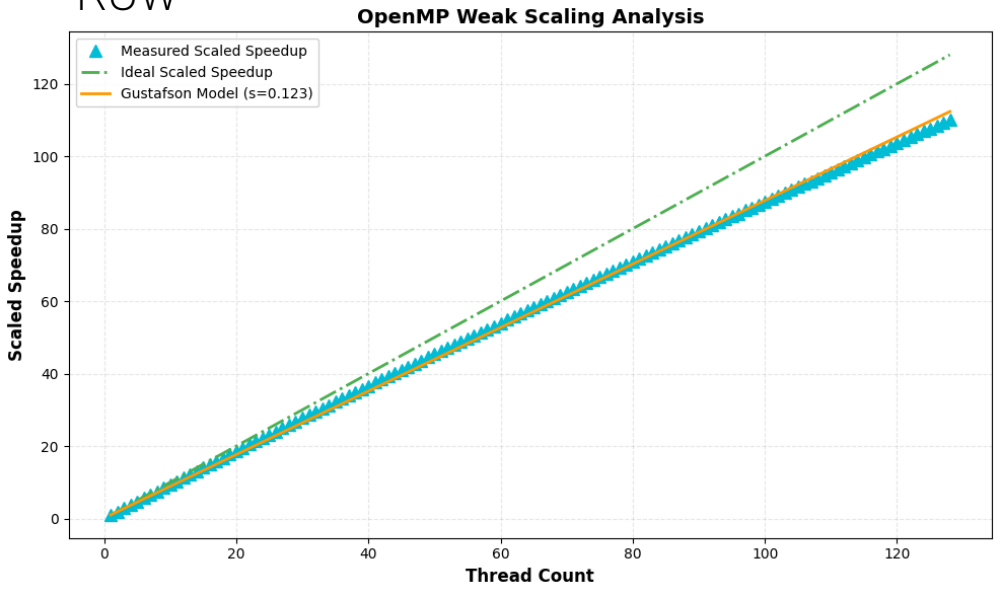
COLUMN

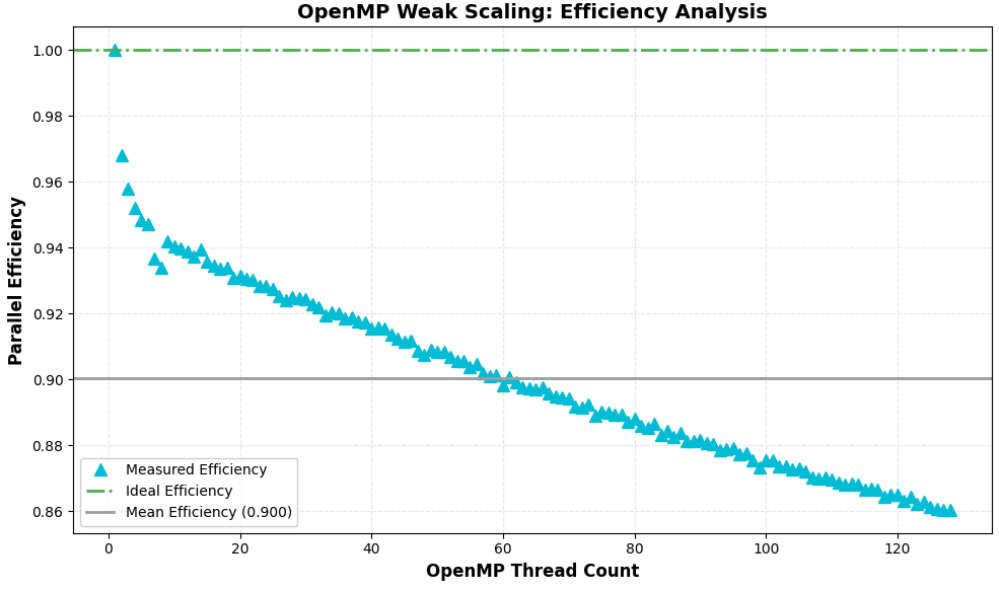




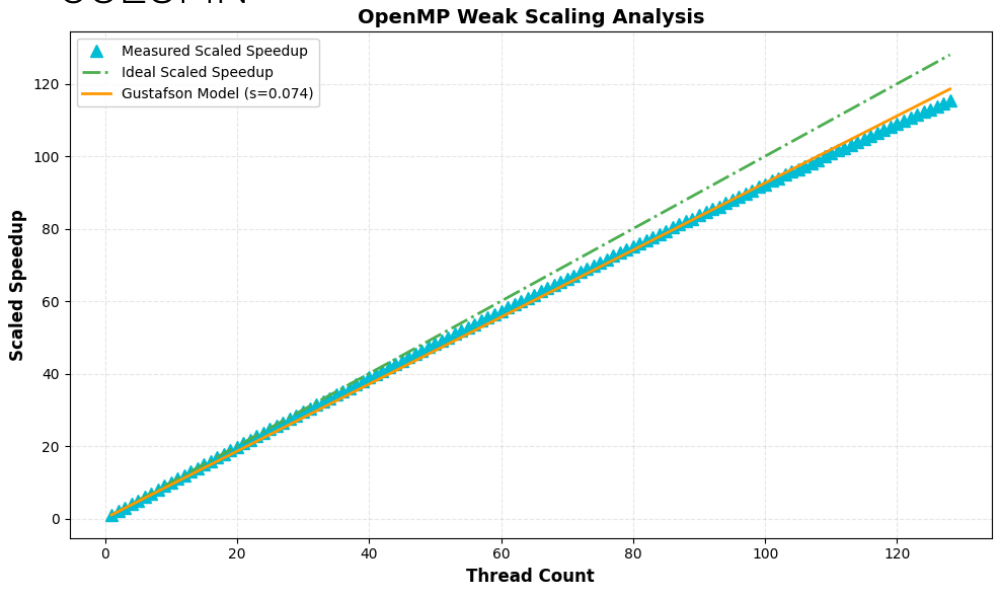
Results presentation – *OMP Weak*

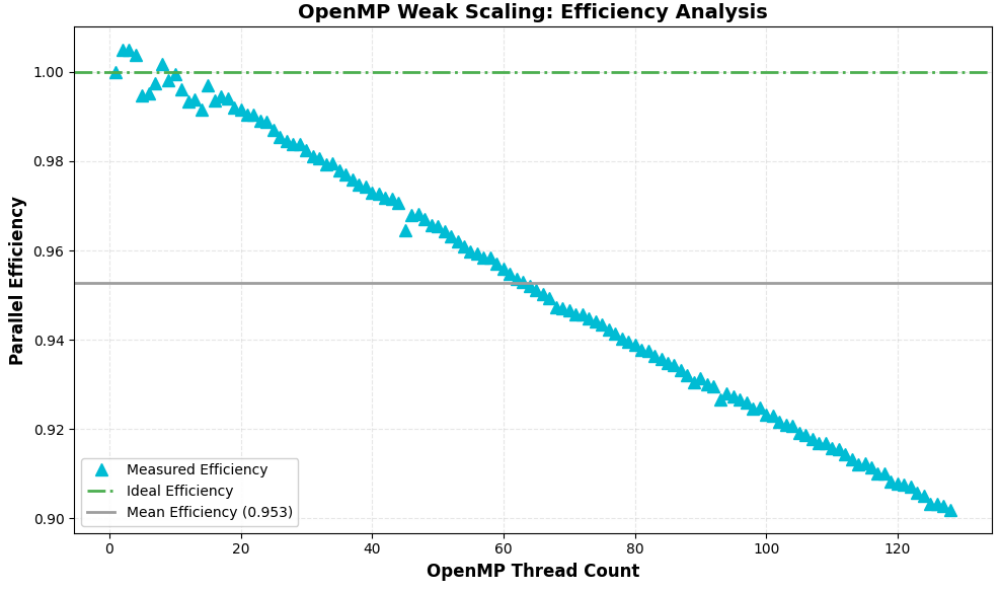
ROW





COLUMN





Discussion on Weak Scailing

- *Row Weak Scaling Summary*:
  - MPI weak scaling shows an increase in variability in efficiency with the number of ranks increased. There is evidence of a growing overhead with problem size and likely caused by network, memory, or load imbalance issues.
  - OpenMP weak scaling remains and an higher serial fraction than strong scaling.
- *Column Weak Scaling Summary*:
  - MPI weak scaling suffers performance degradation, with a mean of efficiency at 30.5%. Gustafson's Law could indicate that overhead dominates and grows faster than computation.
  - OpenMP, as in all the other case, maintains better weak scaling than MPI, with efficiency on average at 95% and the serial fraction is substantially lower than MPI. It is also lower than the row strategy.

Overall, OpenMP outperforms MPI, but both implementations show reduced efficiency in weak scaling due to overheads that increase with problem size and thread count.

Supplementary

This implementation uses hierarchical parallelization:

- MPI performs 2D block-based decomposition across processes
- OpenMP distributes work cyclically among threads

- Because the Mandelbrot set exhibits spatial coherence (neighboring pixels have similar iteration counts), contiguous blocks can be uneven in computation cost; cyclic distribution mitigates this by spreading high-iteration pixels evenly across threads for better load balance.

Code Details

```
void find_process_grid(int total_processes, int *px, int *py) {
    int best_px = 1;
    int min_diff = total_processes;

    for (int i = 1; i <= (int)sqrt(total_processes) + 1; ++i) {
        if (total_processes % i == 0) {
            int j = total_processes / i;
            int diff = abs(i - j);
            if (diff < min_diff) {
                min_diff = diff;
                best_px = i;
            }
        }
    }

    *px = best_px;
    *py = total_processes / best_px;
}
```

This function factorizes the total number of MPI processes into a nearly square 2D grid by finding factor pairs with the smallest difference. It returns grid dimensions (px, py) such that px * py = total_processes, creating a balanced decomposition that minimizes communication overhead and improves load balance

Code Details

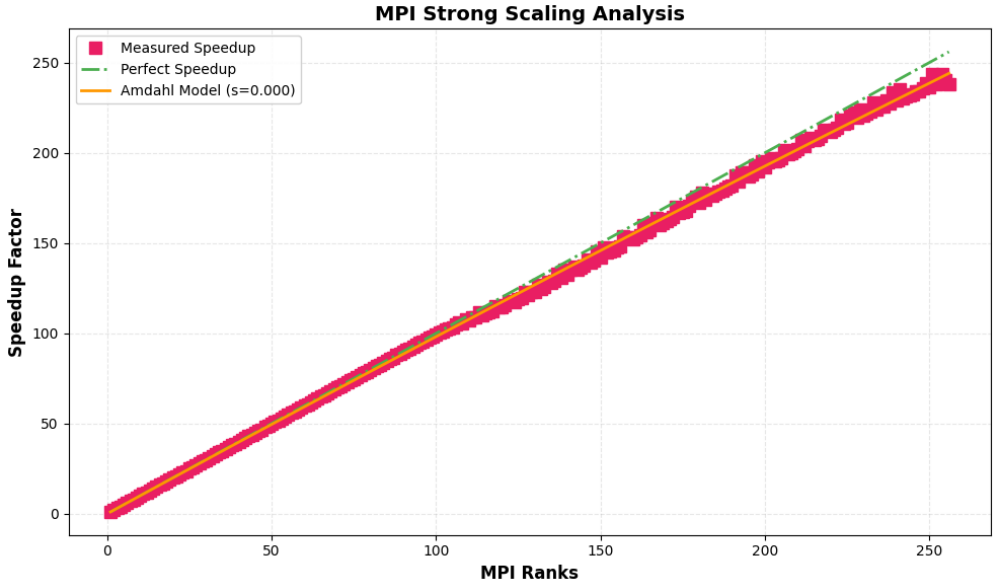
```
int total_pixels = local_width * local_height;
long long total_iterations = 0;

#pragma omp parallel reduction(+:total_iterations)
{
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();

    for (int pixel_idx = thread_id; pixel_idx < total_pixels; pixel_idx += num_threads) {
        int local_x = pixel_idx % local_width;
        int local_y = pixel_idx / local_width;

        int global_x = width_offset + local_x;
        int global_y = height_offset + local_y;

        double coord_x = min_x + global_x * (max_x - min_x) / img_width;
        double coord_y = min_y + global_y * (max_y - min_y) / img_height;

        // Mandelbrot computation (inlined)
        double real_z = coord_x, imag_z = coord_y;
        int iter;
        for (iter = 0; iter < max_iterations; ++iter) {
            double real_sq = real_z * real_z, imag_sq = imag_z * imag_z;
            if (real_sq + imag_sq > 4.0) break;
            imag_z = 2.0 * real_z * imag_z + coord_y;
            real_z = real_sq - imag_sq + coord_x;
        }

        total_iterations += iter;
        local_buffer[local_y * local_width + local_x] = (unsigned char)iter;
    }
}
```

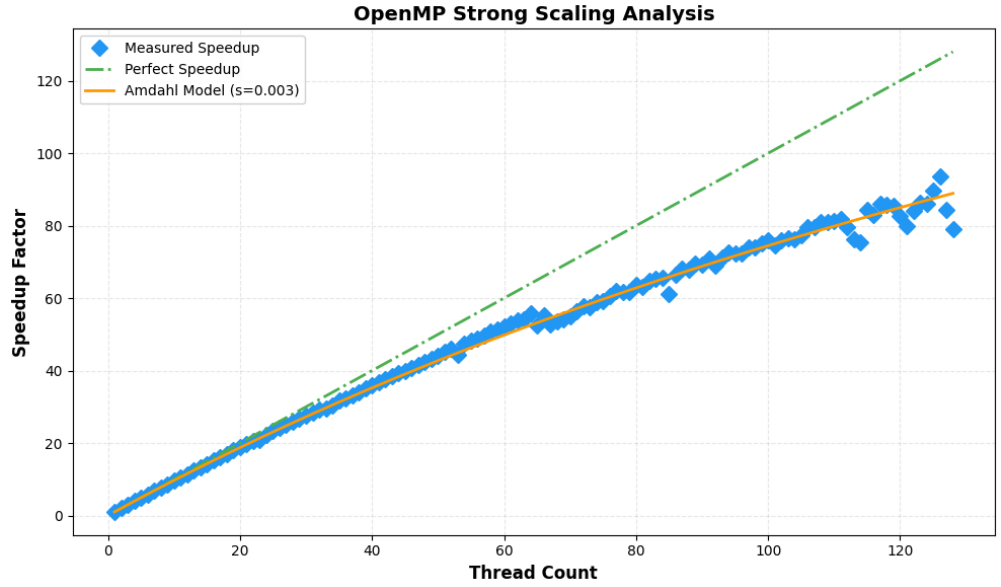
results are stored in row-major order in the local buffer

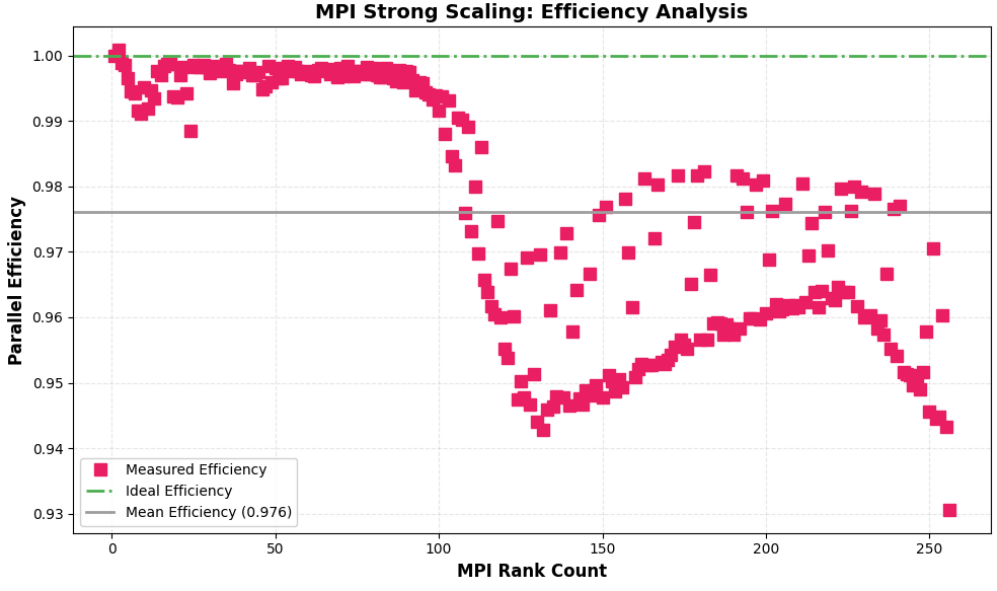
```
long long total_iterations = 0;

#pragma omp parallel reduction(+:total_iterations)
{
    // ... within parallel region ...
    total_iterations += iter;
}

long long global_iterations = 0;
MPI_Reduce(&total_iterations, &global_iterations, 1, MPI_LONG_LONG,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

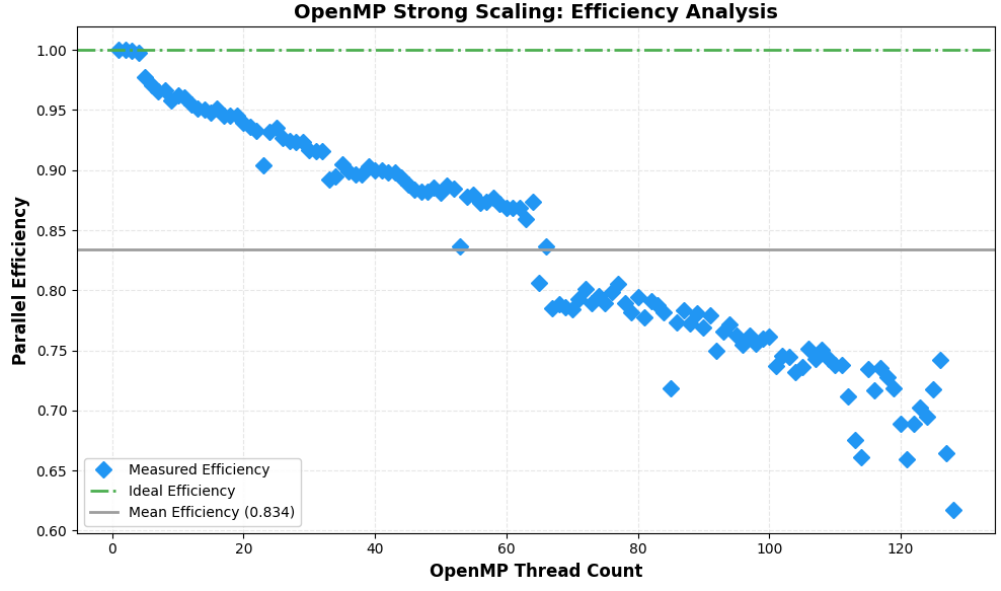
```
unsigned char *reconstructed_image = (unsigned char *)malloc(img_width * img_height);

for (int proc = 0; proc < total_processes; ++proc) {
    int p_x = proc % px;
    int p_y = proc / px;

    int block_w = base_block_width + (p_x < extra_width ? 1 : 0);
    int block_h = base_block_height + (p_y < extra_height ? 1 : 0);
    int w_off = p_x * base_block_width + (p_x < extra_width ? p_x : extra_width);
    int h_off = p_y * base_block_height + (p_y < extra_height ? p_y : extra_height);

    for (int by = 0; by < block_h; ++by) {
        for (int bx = 0; bx < block_w; ++bx) {
            int global_pos = (h_off + by) * img_width + (w_off + bx);
            int block_pos = by * block_w + bx;
            reconstructed_image[global_pos] = final_image[displacements[proc] + block_pos];
        }
    }
}
```

Results presentation – *Strong*

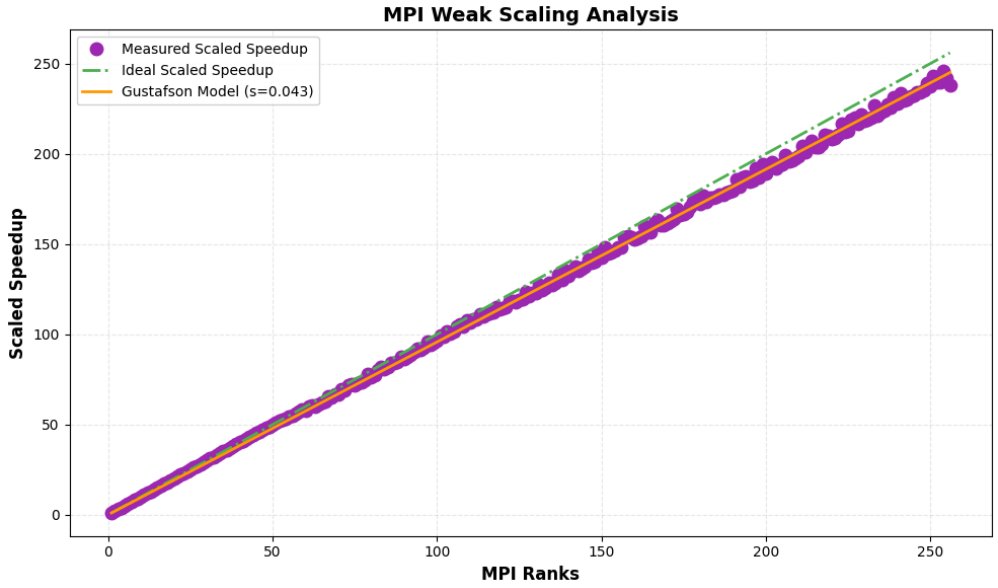


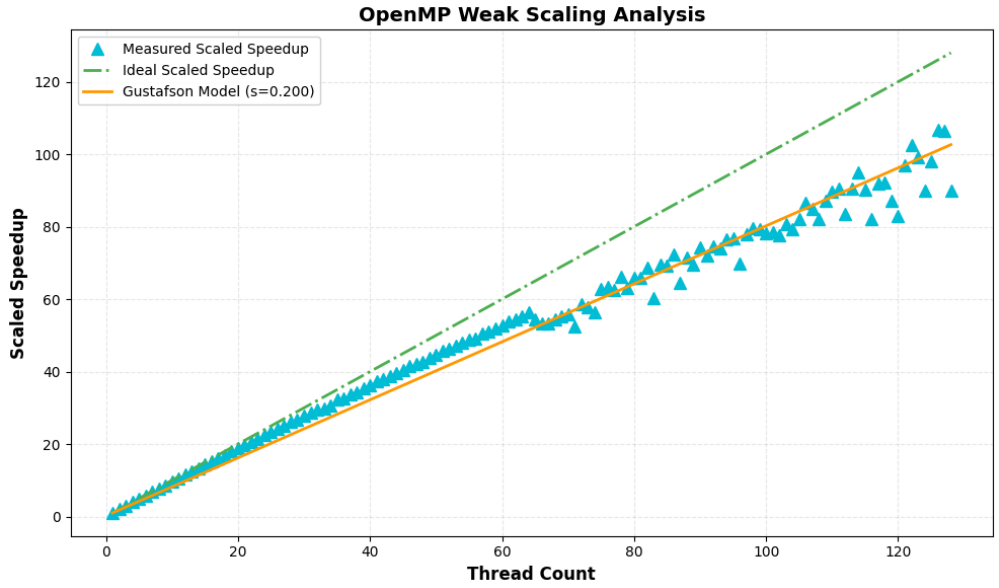


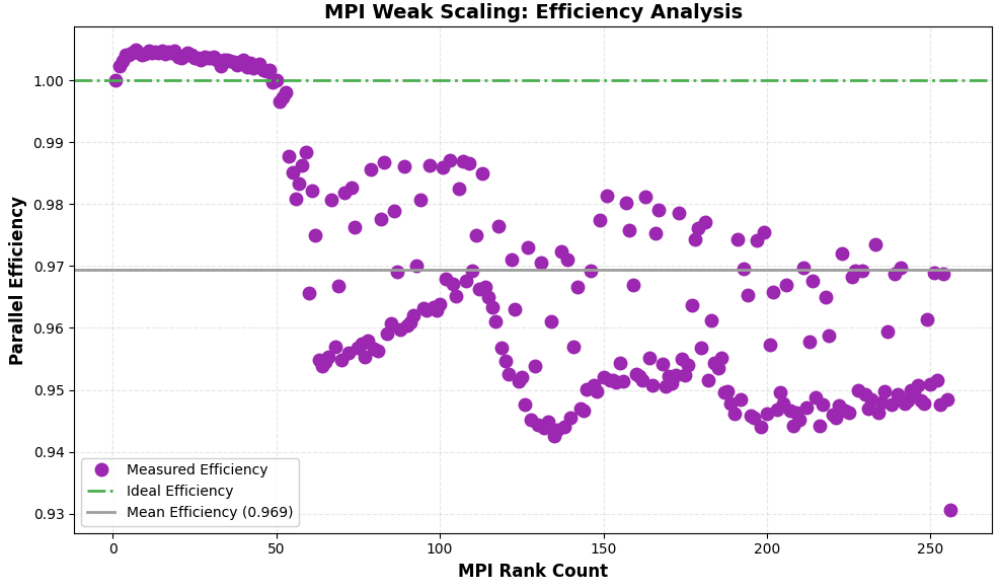


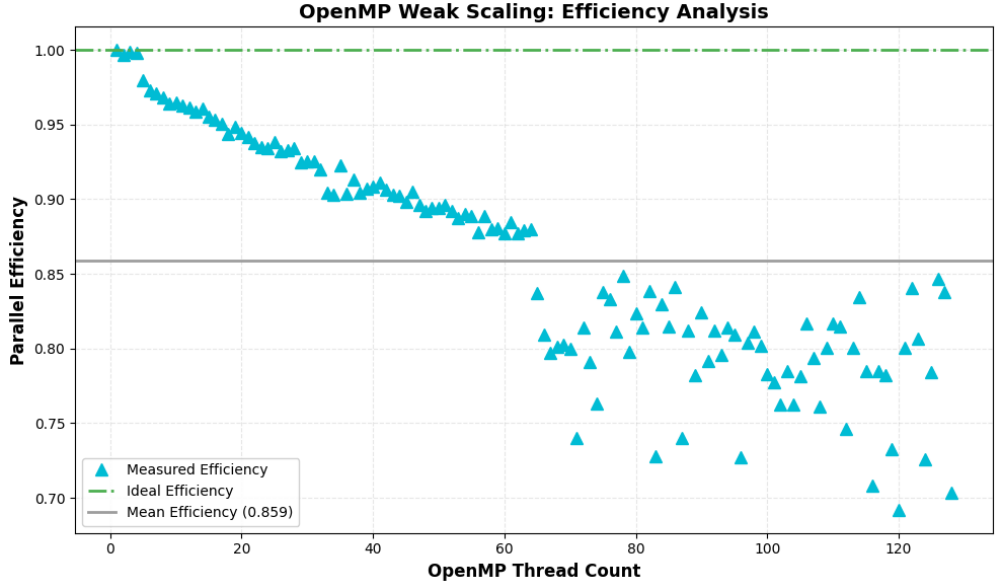


Results presentation – *Weak*









Conclusion

- Increase the size of messages and the number of nodes
- Try to do an analysis on complex real life problems
- Better scaling on MPI with 2d strategy
- Better scaling on OpenMP for row