# High Performance Computing

## Exercises 1 and 2c

**Emanuele Ruoppolo - SM3800049**

*Università degli Studi di Trieste - a.a. 2023-2024*

**25 november 2024**

## Exercise 1

# Performance Evaluation

# OpenMPI collective operations

`broadcast` and `scatter` operations

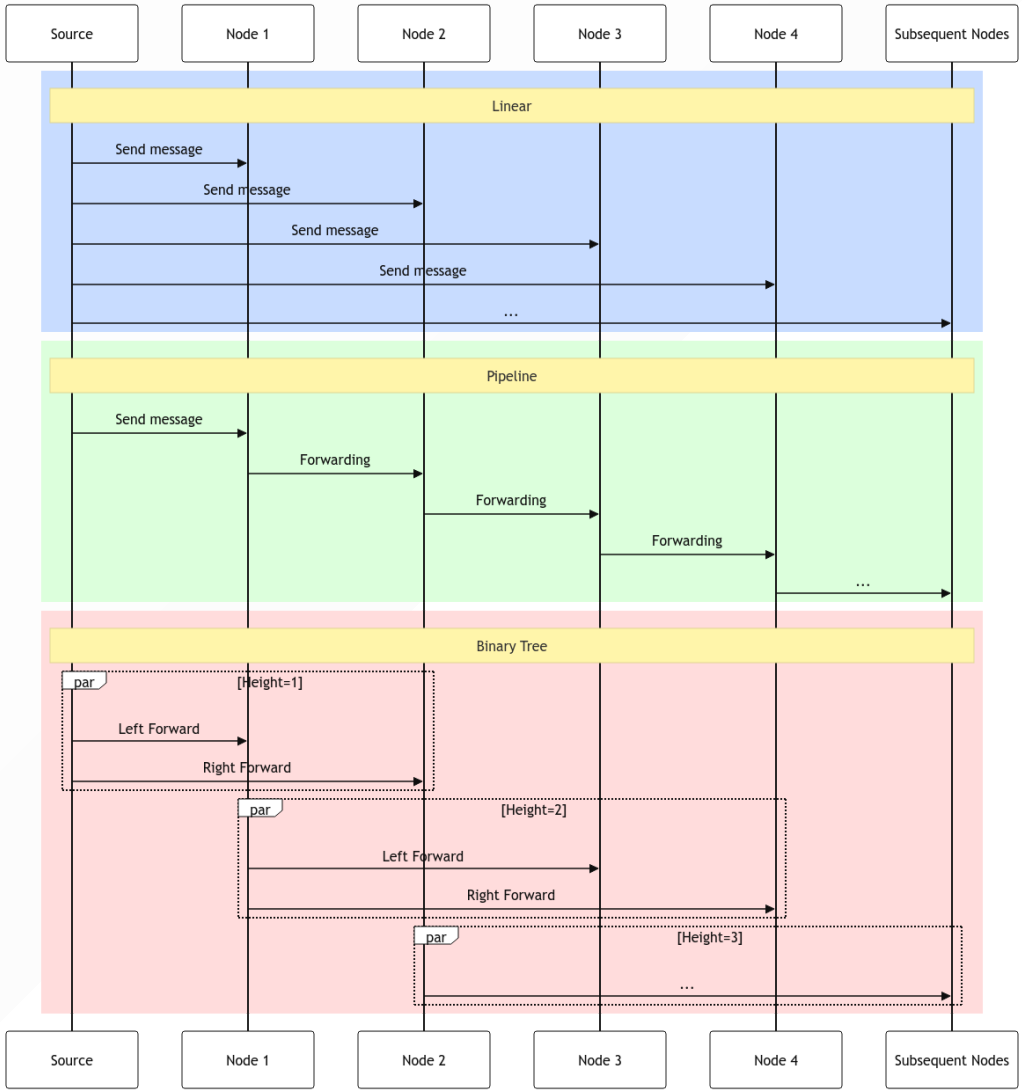
# Problem statement

1. Estimate the latency of default openMPI implementation of two different collective operations
   - Varying the number of processes $\in [2^1 - 2^8]$ and the size of the messages exchanged $\in [2^0 - 2^{20}]$ bytes
   - Comparing the values obtained using different algorithms.
2. Fix the message size, collect data varying the number of processes and build a model to estimate latency of the different algorithms for the two collective operations.

# Setup

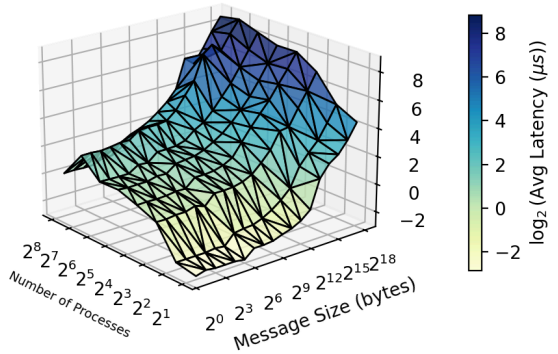
- $2\times$ **AMD EPYC 7H12** nodes on ORFEO cluster, 256 cores in total
- `map-by core` policy
- multiple iterations (10000) to get stable results
- bash scripts to run the jobs
- data analysis on jupyter notebooks

# `broadcast`: linear, pipeline, binary tree

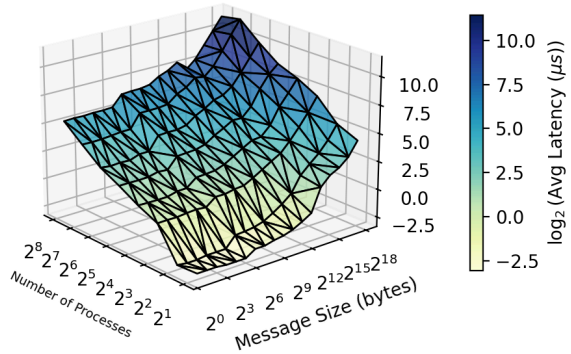


# Overall Analysis

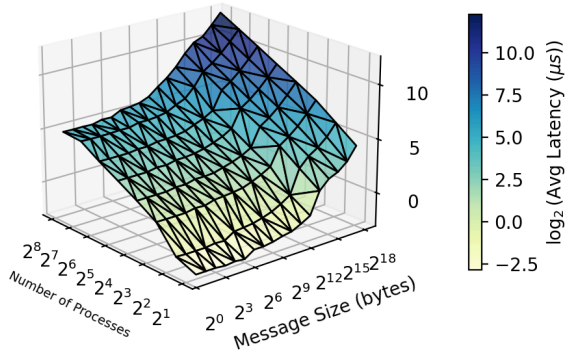
Broadcast Latency
map-by core, default algorithm



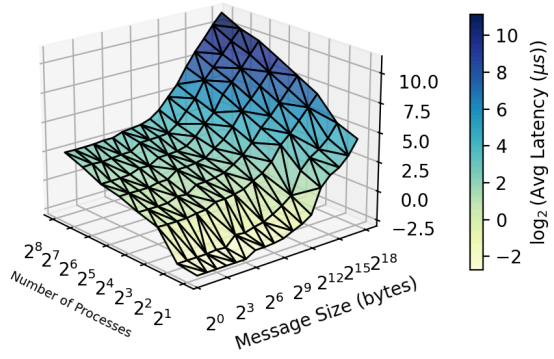

Broadcast Latency
map-by core, basic_linear algorithm

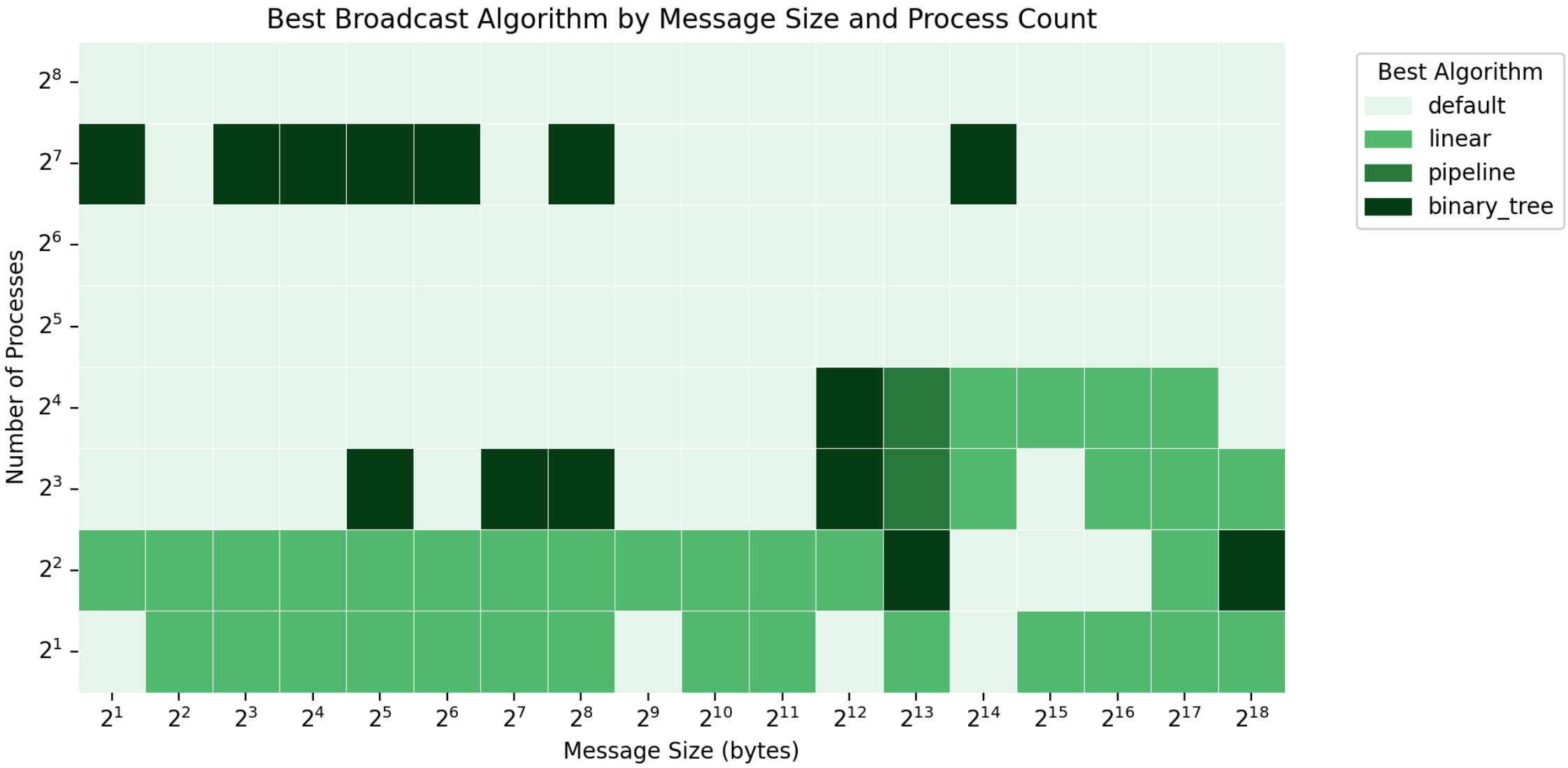



Broadcast Latency
map-by core, pipeline algorithm

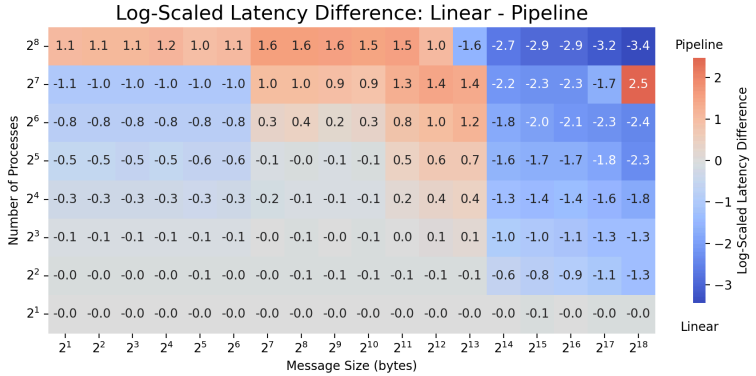


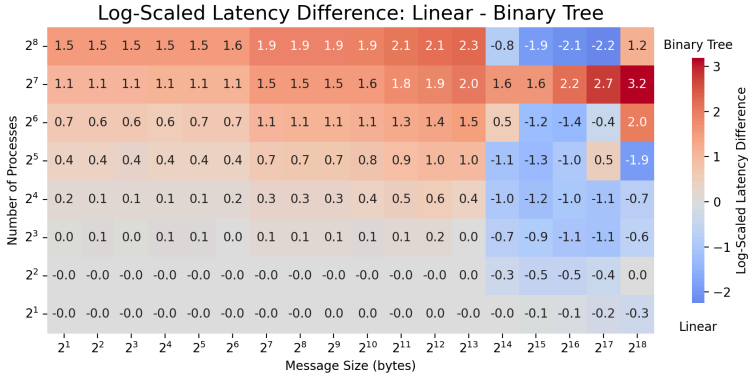

Broadcast Latency
map-by core, binary_tree algorithm

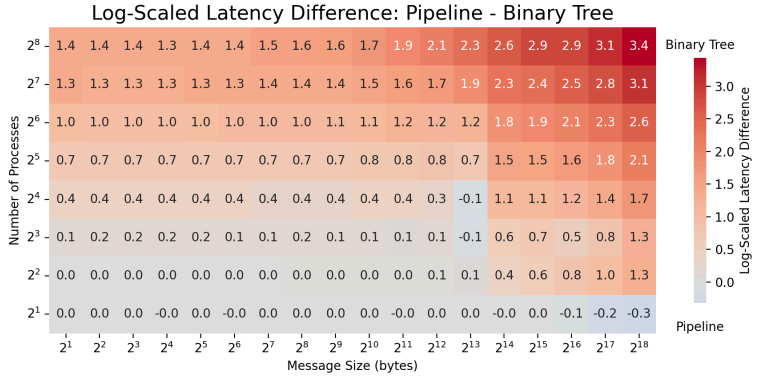



Best Broadcast Algorithm by Message Size and Process Count
Number of Processes
$2^8$
$2^7$
$2^6$
$2^5$
$2^4$
$2^3$
$2^2$
$2^1$
$2^1$ $2^2$ $2^3$ $2^4$ $2^5$ $2^6$ $2^7$ $2^8$ $2^9$ $2^{10}$ $2^{11}$ $2^{12}$ $2^{13}$ $2^{14}$ $2^{15}$ $2^{16}$ $2^{17}$ $2^{18}$
Message Size (bytes)
Best Algorithm
default
linear
pipeline
binary_tree

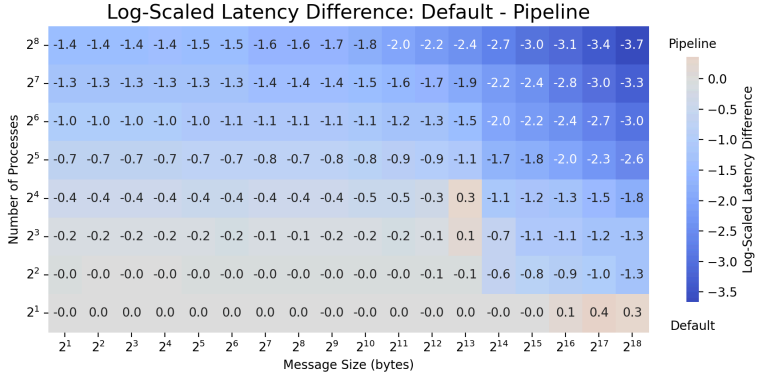
# Log-Scaled latency differences between Broadcast algorithms

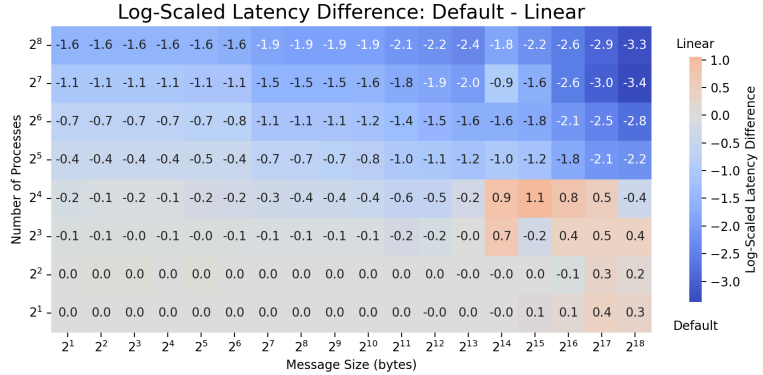


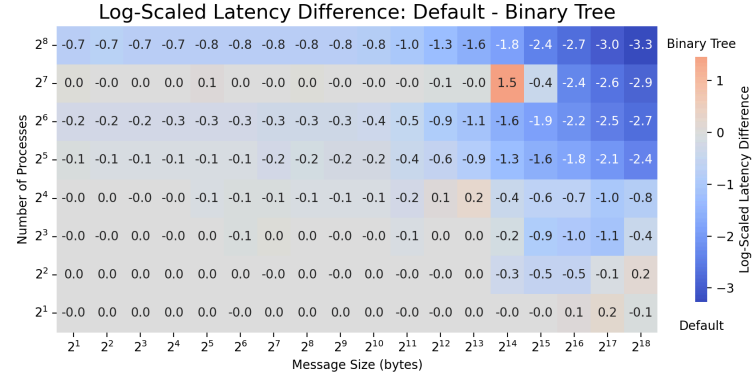



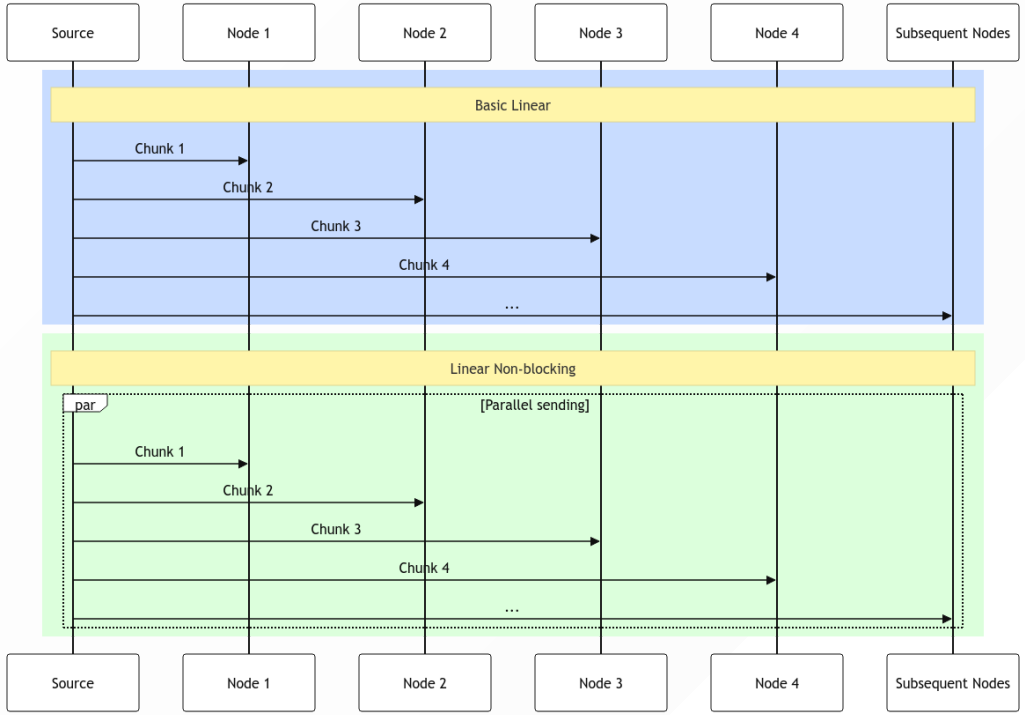

# Log-Scaled latency differences between Broadcast algorithms with default

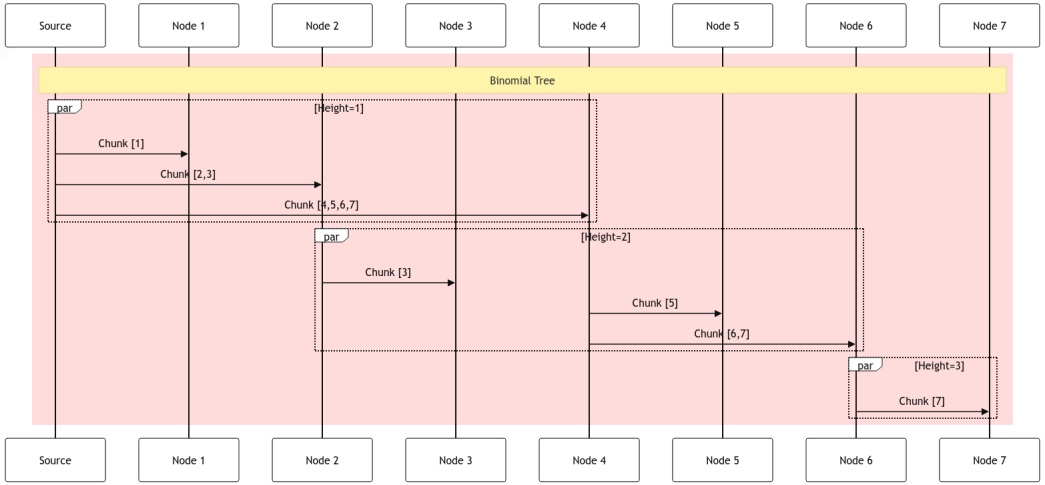


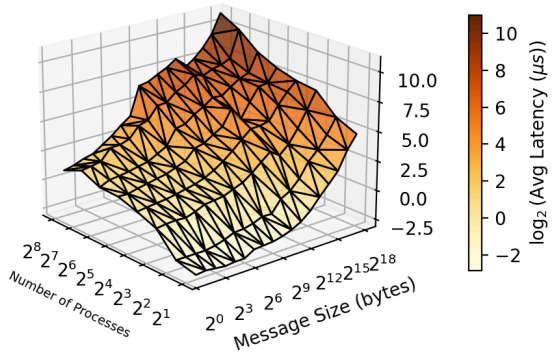
# `scatter`: basic linear, non-blocking, binomial



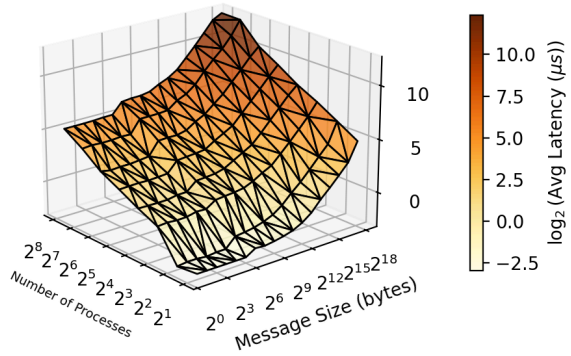



# Overall Analysis

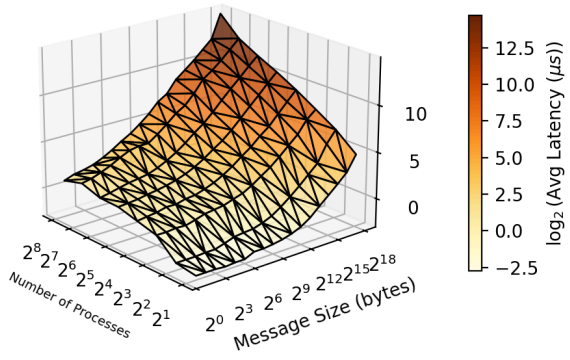
Scatter Latency
map-by core, default algorithm



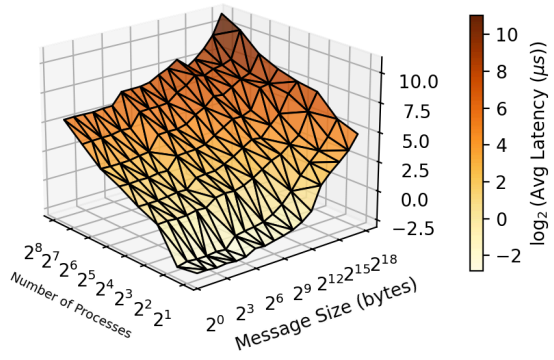

Scatter Latency
map-by core, basic_linear algorithm

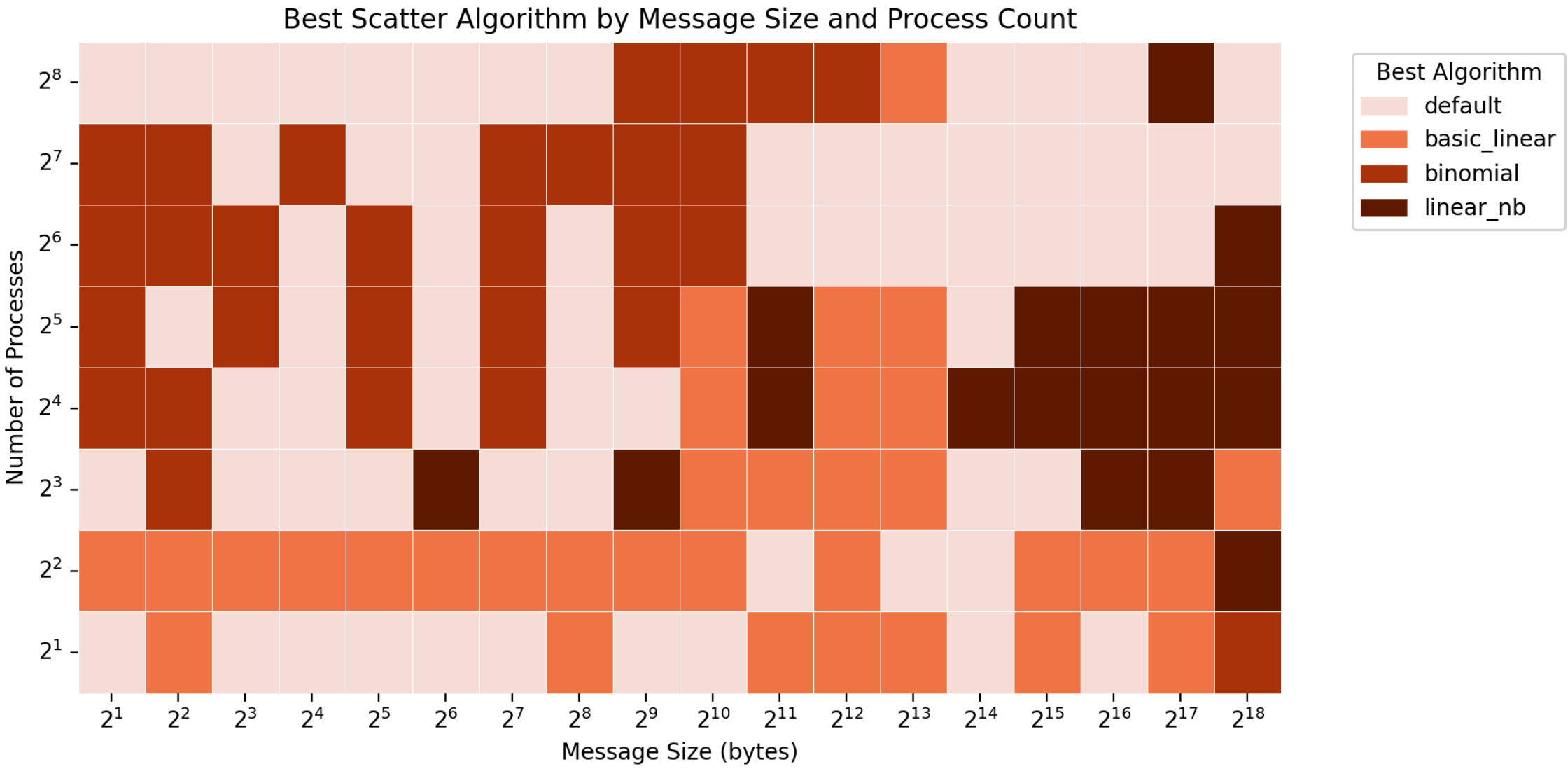



Scatter Latency
map-by core, binomial algorithm

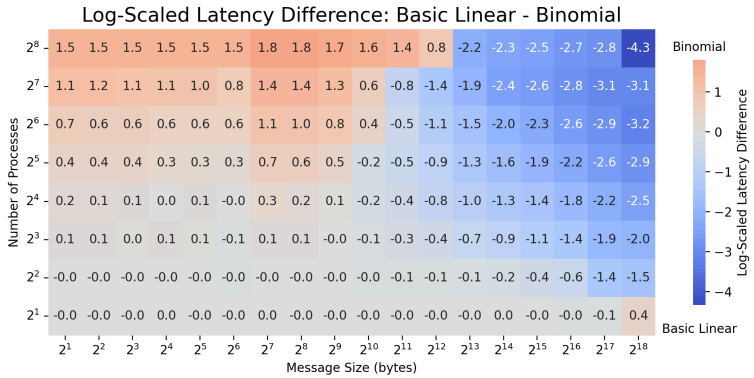


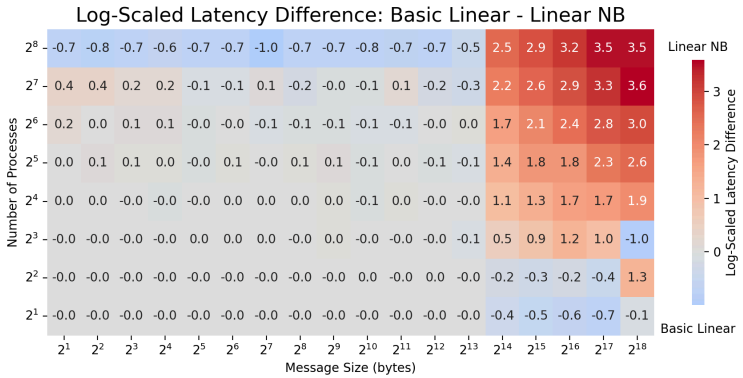

Scatter Latency
map-by core, linear_nb algorithm

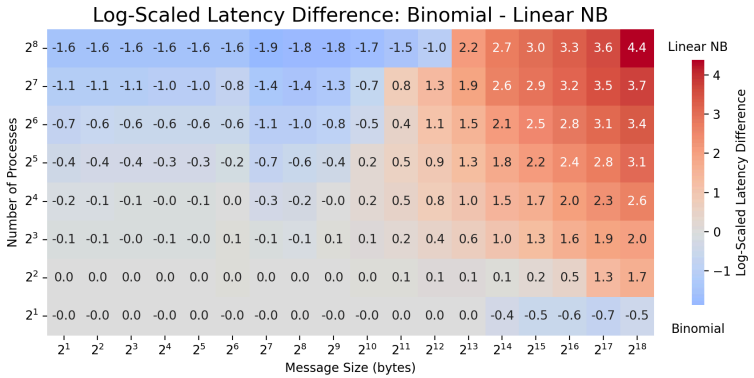



Best Scatter Algorithm by Message Size and Process Count
Number of Processes
$2^8$
$2^7$
$2^6$
$2^5$
$2^4$
$2^3$
$2^2$
$2^1$
$2^1$ $2^2$ $2^3$ $2^4$ $2^5$ $2^6$ $2^7$ $2^8$ $2^9$ $2^{10}$ $2^{11}$ $2^{12}$ $2^{13}$ $2^{14}$ $2^{15}$ $2^{16}$ $2^{17}$ $2^{18}$
Message Size (bytes)
Best Algorithm
default
basic_linear
binomial
linear_nb

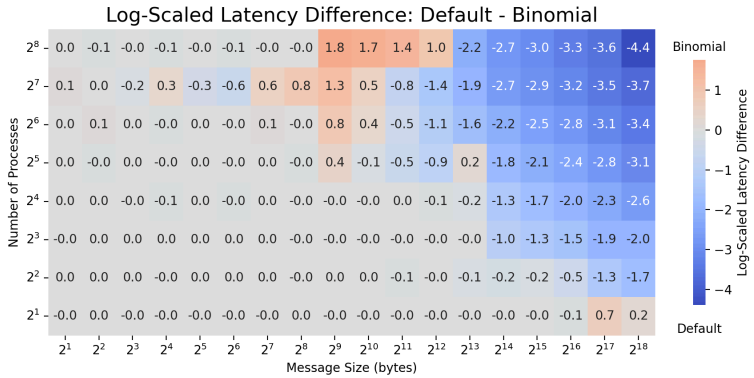
# Log-Scaled latency differences between Scatter algorithms

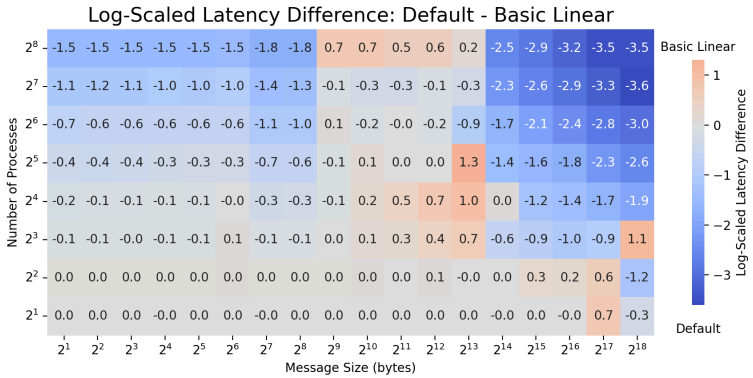


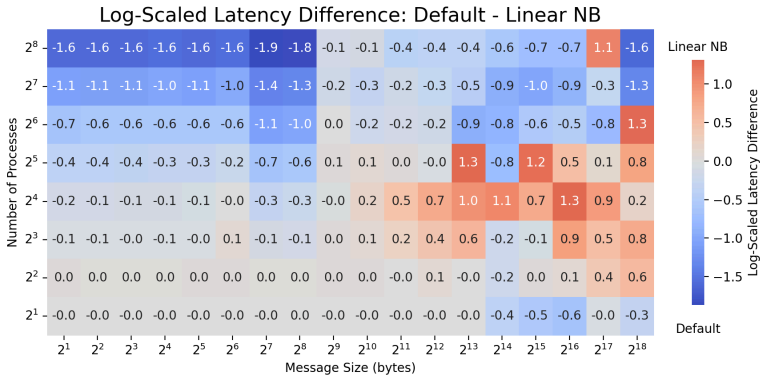




# Log-Scaled latency differences between Scatter algorithms with default



# Performance models

- `--report-bindings` flag to understand resources allocation
- fixed message size (4 bytes): instataneous data transfer **assumption**
- point-to-point latency communication between the different node binding:

| Region | Latency $(\mu s)$ |
|---|---|
| Same CCX | 0.15 |
| Same CCD, Diff. CCX | 0.31 |
| Same NUMA | 0.34 |
| Same SOCKET | 0.36 |
| Diff. SOCKET | 0.65 |
| Diff. NODE | 1.82 |

# Pipeline model

$$T_{\text{pipeline}}(n) = \sum_{i=0}^{n-1} T_{\text{pt2pt}}(i, i+1)$$

- $T_{\text{pipeline}}(n)$: latency for $n$-th process
- $T_{\text{pt2pt}}(i, i+1)$: communication latency within subsequent processes

# Linear model

## Accounting method for distant processes

$$T_{\text{linear}}(n) = \sum_{i=0}^{n-1} T_{\text{pt2pt}}(0, i) \cdot \text{discount}(i)$$

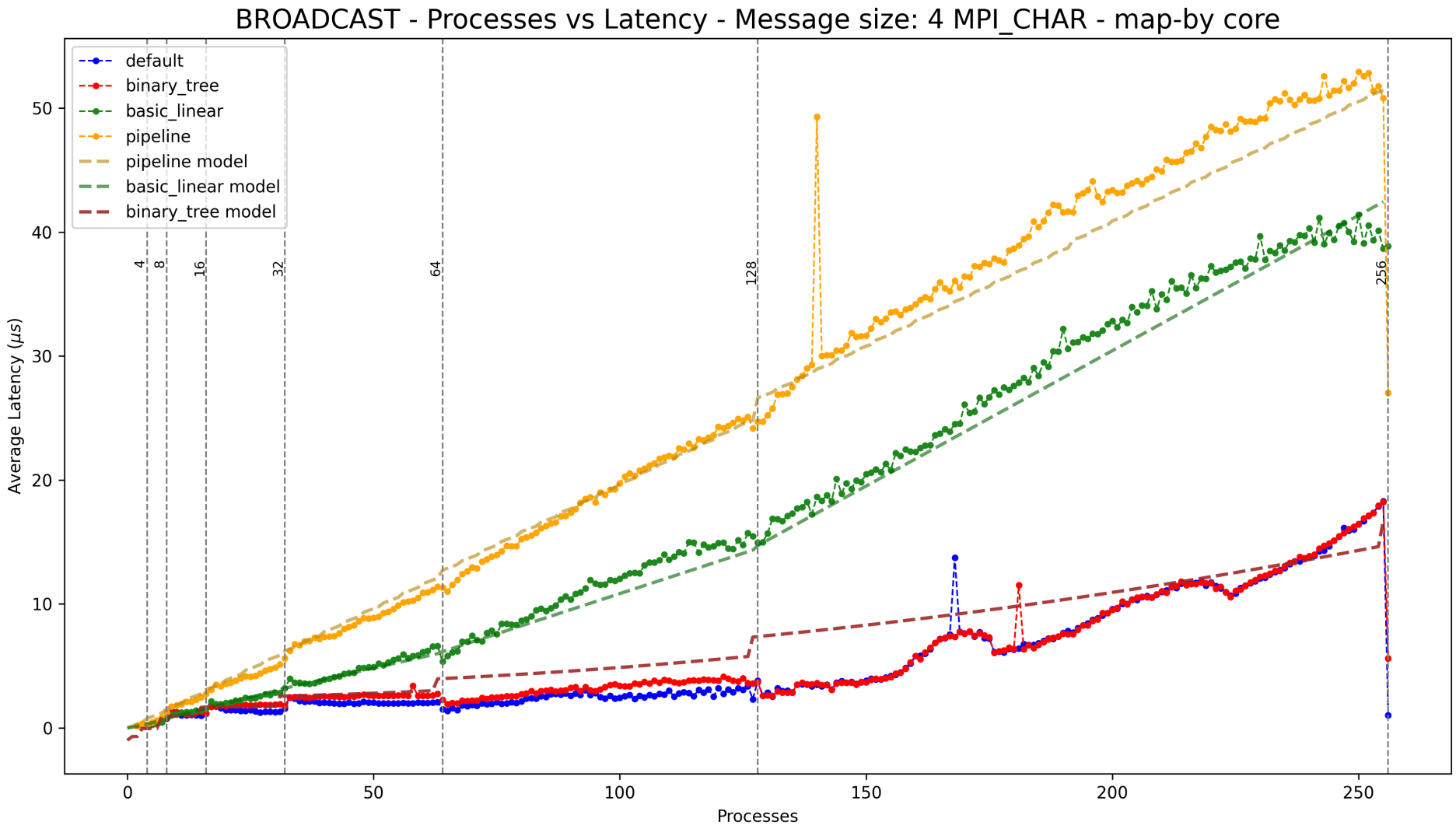
- $T_{\text{pt2pt}}(0, i)$: communication latency within first the $i-$th processes
- $\text{discount}(i)$: discount factor for the $i$-th process, empirically fixed for each region

# Binary tree model

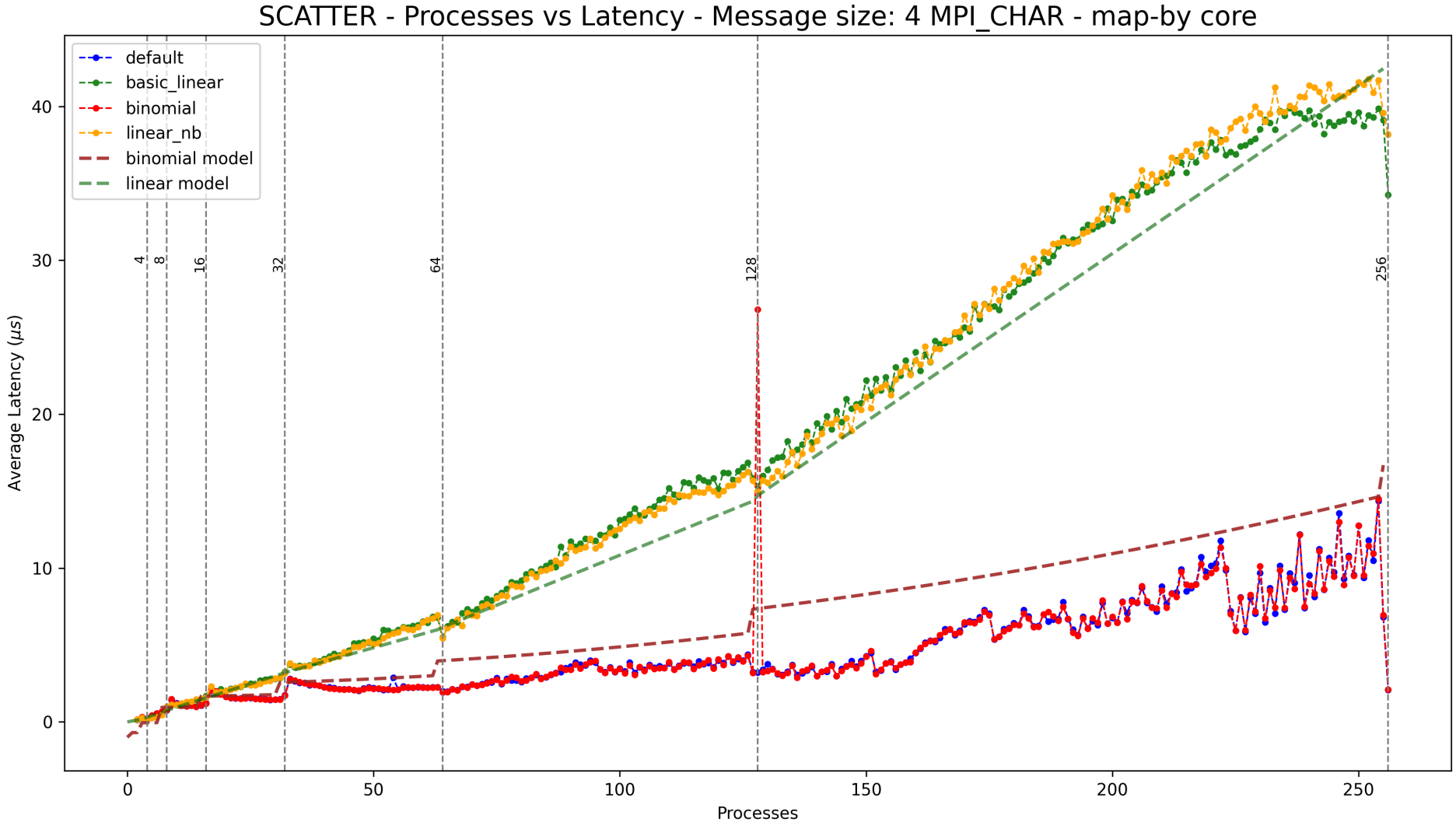
$$T_{\text{binary}}(n) = \text{n-proc-pen}(n) + \sum_{i=1}^{H(n)} T_{\text{pt2pt}}(i) \cdot \big(1 + \text{comm-pen}(i)\big)$$

- $T_{\text{pt2pt}}(i)$: latency of the longest communication occurring on the $i$-th level
- $H(n)$: height of the tree at the $n-$th process
- $\text{n-proc-pen}(n)$: penalty factor for the total number of processes
- $\text{comm-pen}(i)$: penalty factor depending on the communications on the $i$-th layer

# `broadcast` results



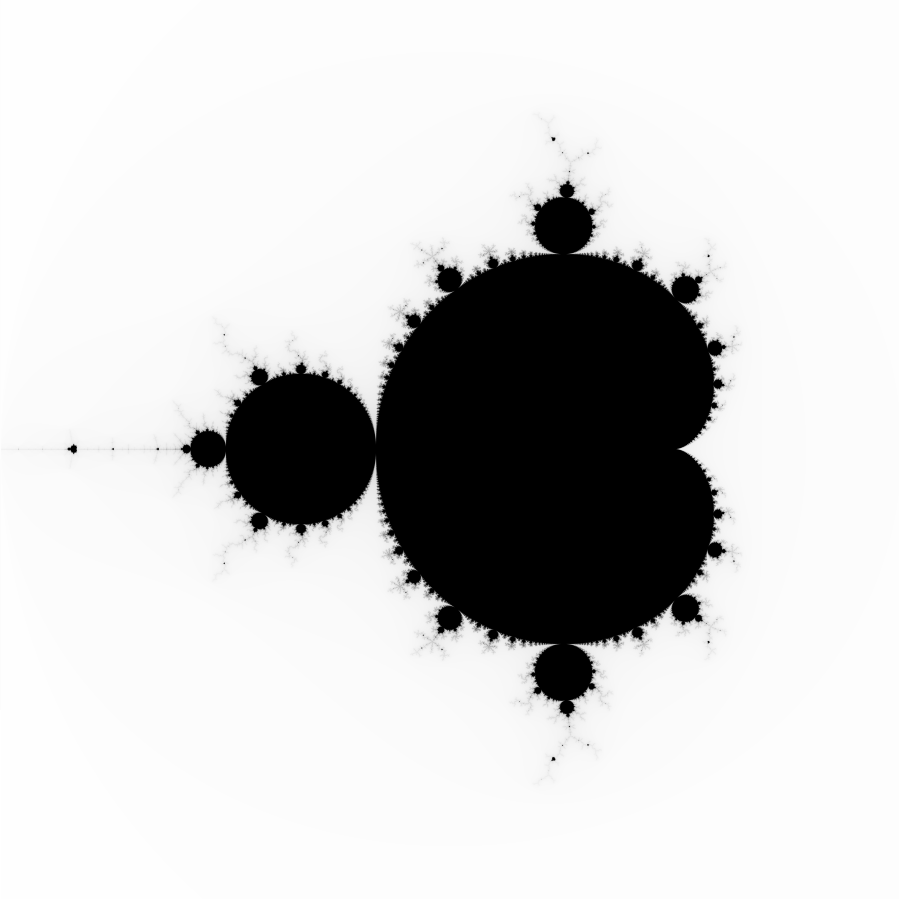
# `scatter` results



## Exercise 2c

# Mandelbrot Set Computation

## Hybrid Implementation



# Mandelbrot set

The Mandelbrot set $\mathcal{M}$ emerges on the complex plane $\mathbb{C}$ through the iteration of the function

$$f_c(z) = z^2 + c$$

where $c = x + iy$ is a complex number. The process generates the sequence

$$z_0 = 0, \quad z_1 = f_c(0), \quad z_2 = f_c(z_1), \quad \ldots, \quad z_n = f_c(z_{n-1}).$$

The set comprises all the points $c$ for which this sequence remains bounded. To determine whether a point $c$ is part of $\mathcal{M}$, the following condition is used

$$|z_n| < 2 \quad \text{for all } n \leq I_{\max},$$

# Problem statement

1. Implement a hybrid strategy OpenMP+MPI to compute (and then visualize in $\mathbb{R}^2$) the set $\mathcal{M}$
   - Using 1 byte pixels resolution ($I_{\max} = 255$)
   - Returning as output a `.pgm` image
2. Scale the problem and assess the performances

## Curse of the problem

- Workload balancement: the points distributions is not uniform, some processes may work way more than others

# Strategy

- Each point $c_{ij}$, $(i, j)$ element of a $n \times n$ matrix, can be computed independently from the others, within a `max_iter=255` range, and assigned to the set (or not)

```
int compute_mandelbrot(double cx, double cy, int max_iter) { // single point computation
    // ... variables definition
    while (x2 + y2 <= 4 && iter < max_iter) {
        y = 2 * x * y + cy;
        x = x2 - y2 + cx;
        x2 = x * x;
        y2 = y * y;
        iter++;
    }
    return iter;
}
```

- The matrix can be divided by rows assigned to MPI processes, each MPI process can divide its work between OMP threads
- Each process writes its own results in the final image using `MPI_File_write_at()`

# First attempt: static (MPI) workload

- Rows are evenly and subsequently divided between 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);
```

- Within a single process the computation is dynamically divided between threads using the OMP directive `schedule(dynamic)`

```
#pragma omp parallel for schedule(dynamic)
for (int j = 0; j < local_rows; j++) {
    for (int i = 0; i < nx; i++) {
        double cx = x_left + i * dx;
        double cy = y_left + (start_row + j) * dy;
        int iter = 255 - compute_mandelbrot(cx, cy, max_iter);
        local_image[j * nx + i] = (unsigned char)(iter == max_iter ? 1 : iter);
    }
}
```

# Scaling

$$\text{Size of the image} = n \times n$$

- **Weak scaling**: The size is increased linearly with the workers (processes or threads), setting $n = \sqrt{W \times c}$ where $c$ determines a constant workload, $c = 1\text{MB}$
- **Strong scaling**: The size is fixed at $n = 10000$, the workers are linearly increased

| | OMP | | MPI | |
|---|---|---|---|---|
| | **Strong** | **Weak** | **Strong** | **Weak** |
| **Processes** | 1 | | [1-256] | |
| **Threads** | [1-128] | | 1 | |
| **Size (pixels)** | 10000 | n | 10000 | n |

# Analysis

Considering $N$ workers to analyze the scaling we consider:

## Strong scaling

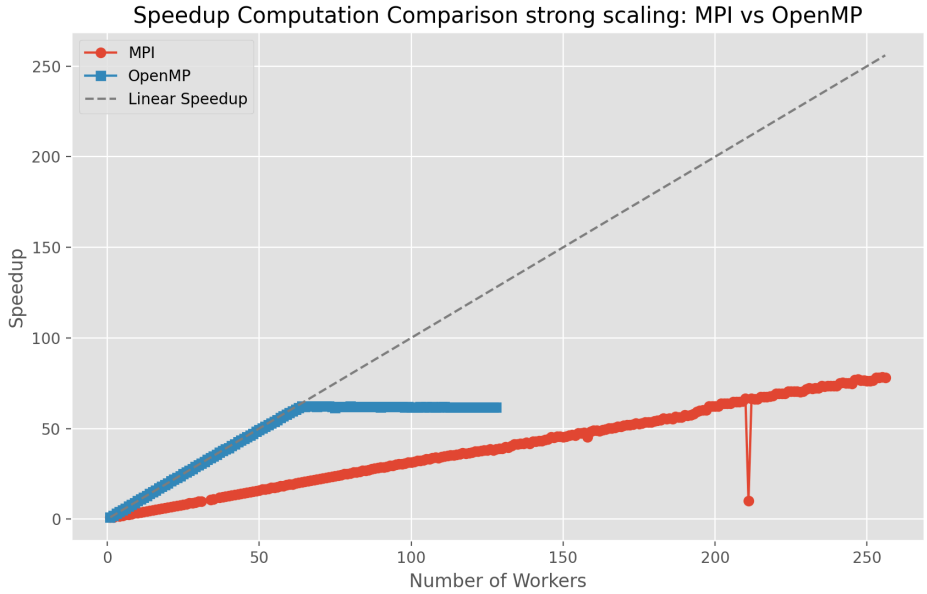
- Speedup and Efficiency

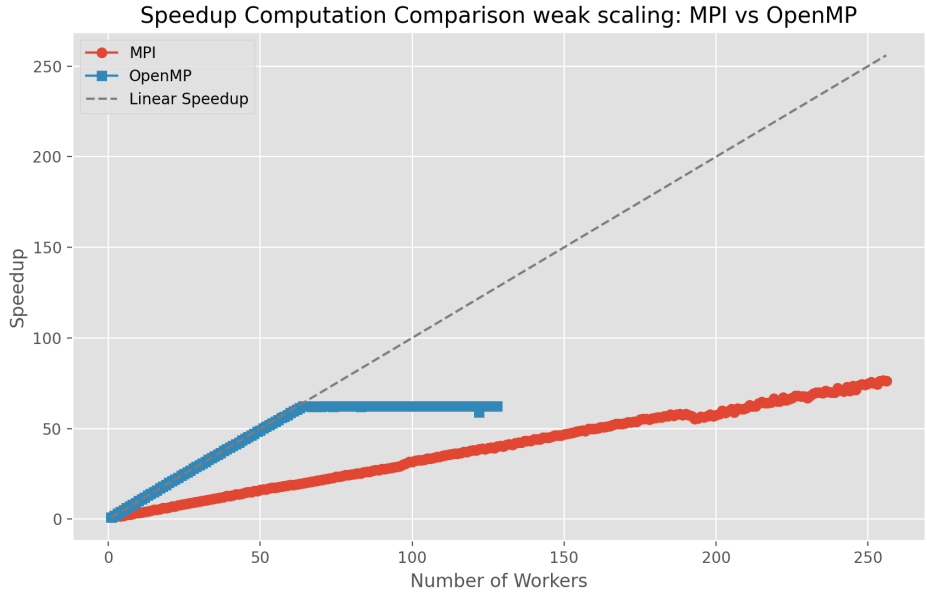
$$\mathrm{S}(N) = \frac{T(1)}{T(N)} \quad \text{and} \quad \mathrm{E}(N) = \frac{S(N)}{N}$$

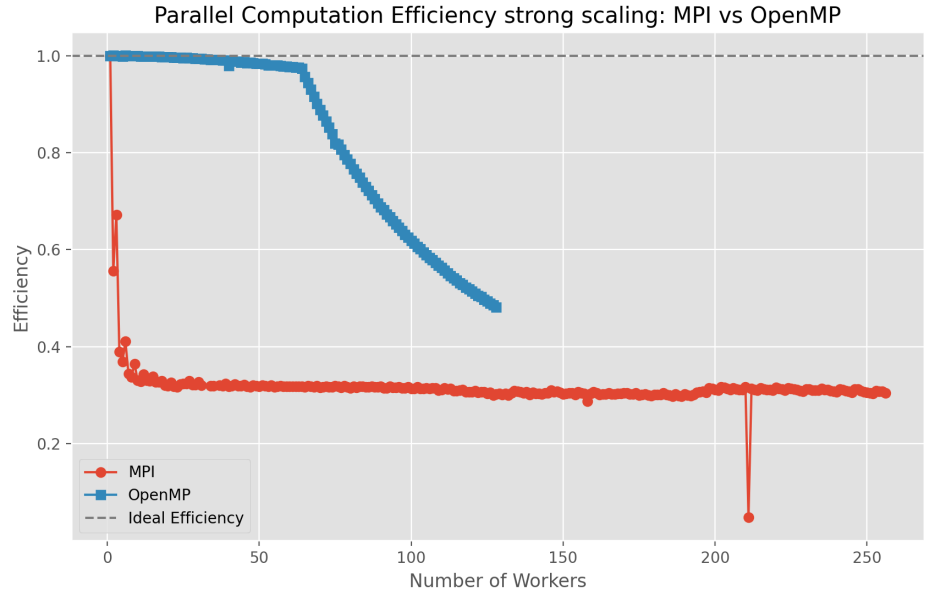
## Weak scaling

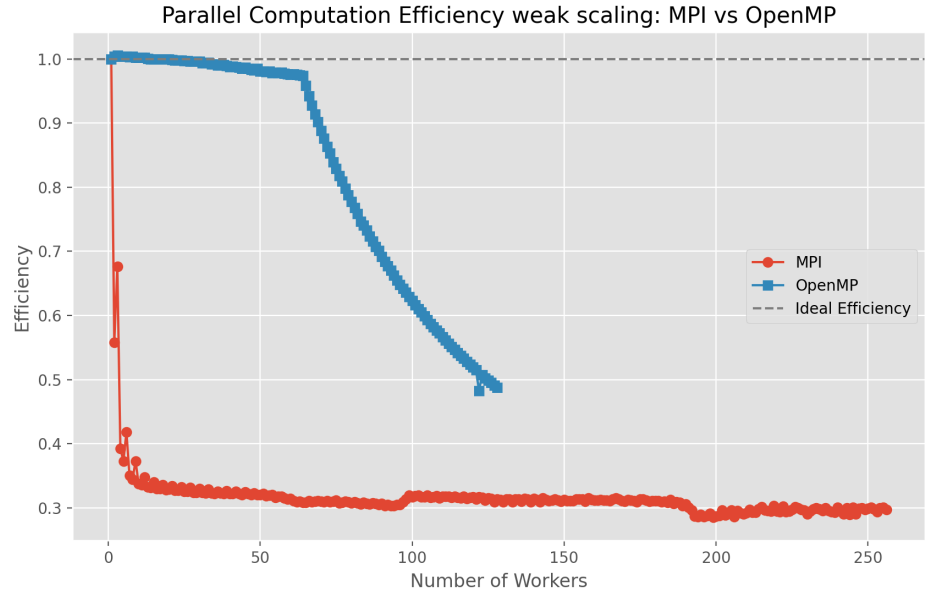
- Scaled speedup and Efficiency

$$\mathrm{S}(N) = \frac{T(1)}{T(N)} \cdot N \quad \text{and} \quad \mathrm{E}(N) = \frac{S(N)}{N}$$


Speedup Computation Comparison strong scaling: MPI vs OpenMP
MPI
OpenMP
Linear Speedup
Speedup
Number of Workers
0
50
100
150
200
250


Speedup Computation Comparison weak scaling: MPI vs OpenMP
MPI
OpenMP
Linear Speedup
Speedup
Number of Workers
0
50
100
150
200
250


Parallel Computation Efficiency strong scaling: MPI vs OpenMP
MPI
OpenMP
Ideal Efficiency
Efficiency
Number of Workers
0.2
0.4
0.6
0.8
1.0
0
50
100
150
200
250


Parallel Computation Efficiency weak scaling: MPI vs OpenMP
MPI
OpenMP
Ideal Efficiency
Efficiency
Number of Workers
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
50
100
150
200
250

# Evaluation

Considering $N$ workers to assess the scaling (and the program itself) we consider $f$ the serial workload fraction of the program and $(1 - f)$ the parallel workload fraction:

## Strong scaling

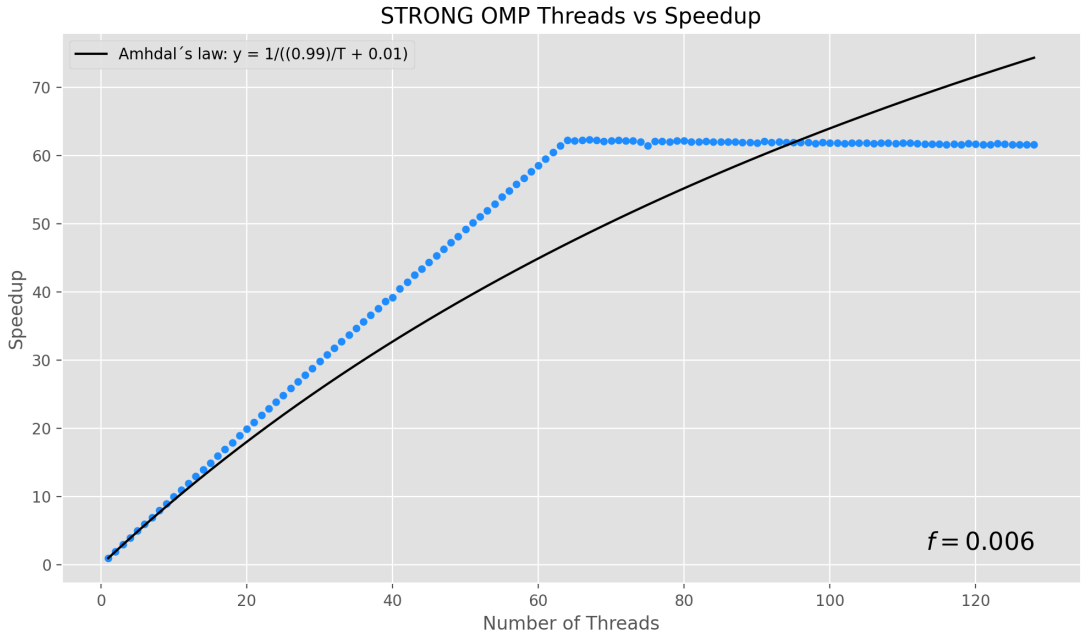
- Amdhal's law

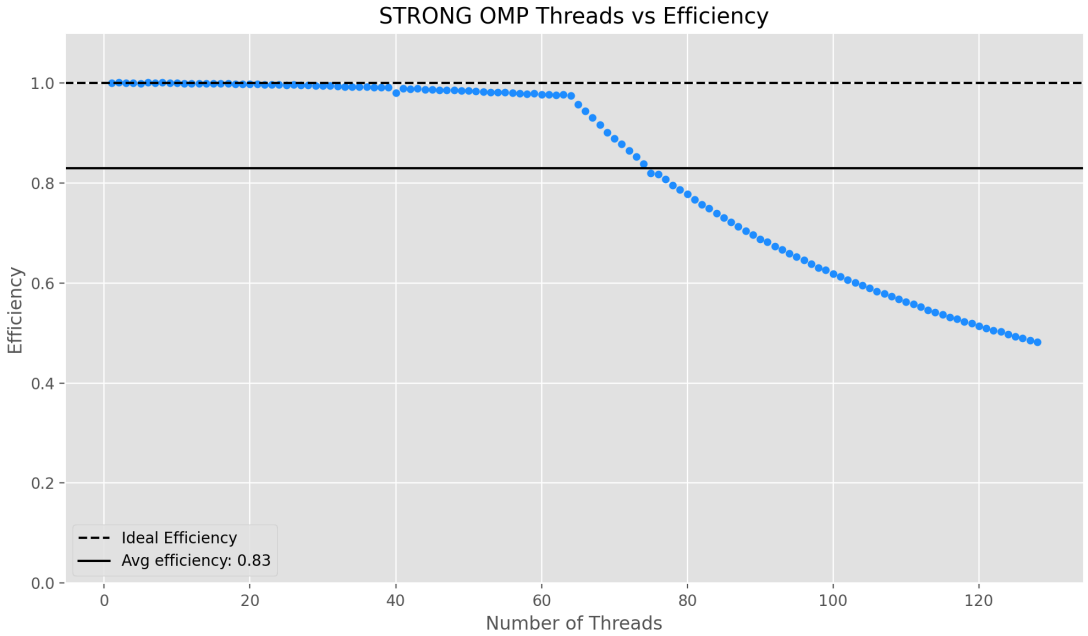
$$S_A(N) = \frac{1}{f + (1 - f)/N}$$

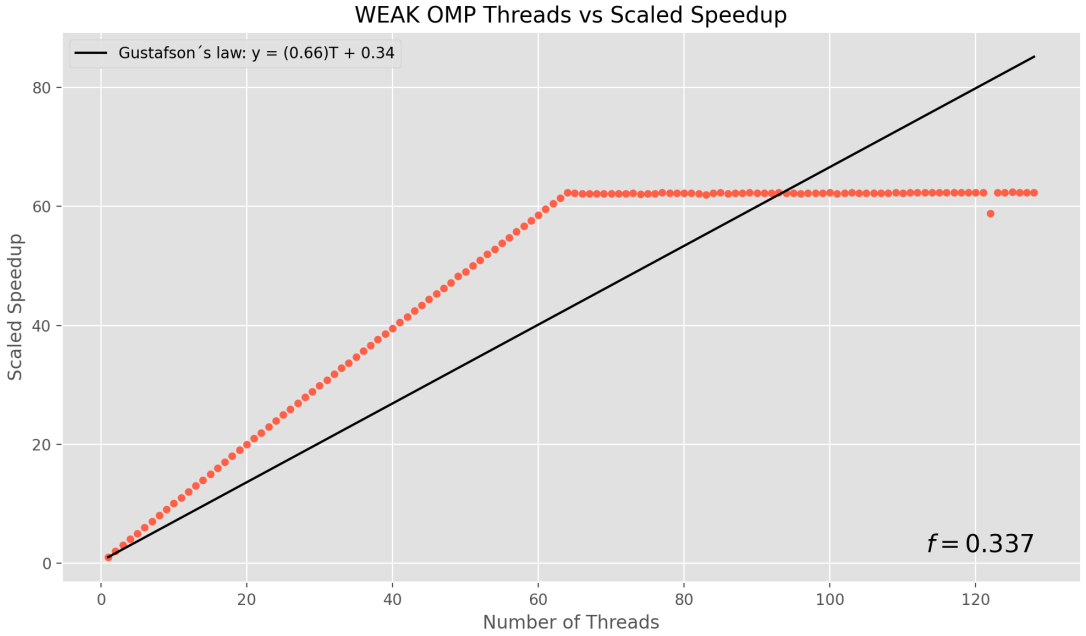
## Weak scaling

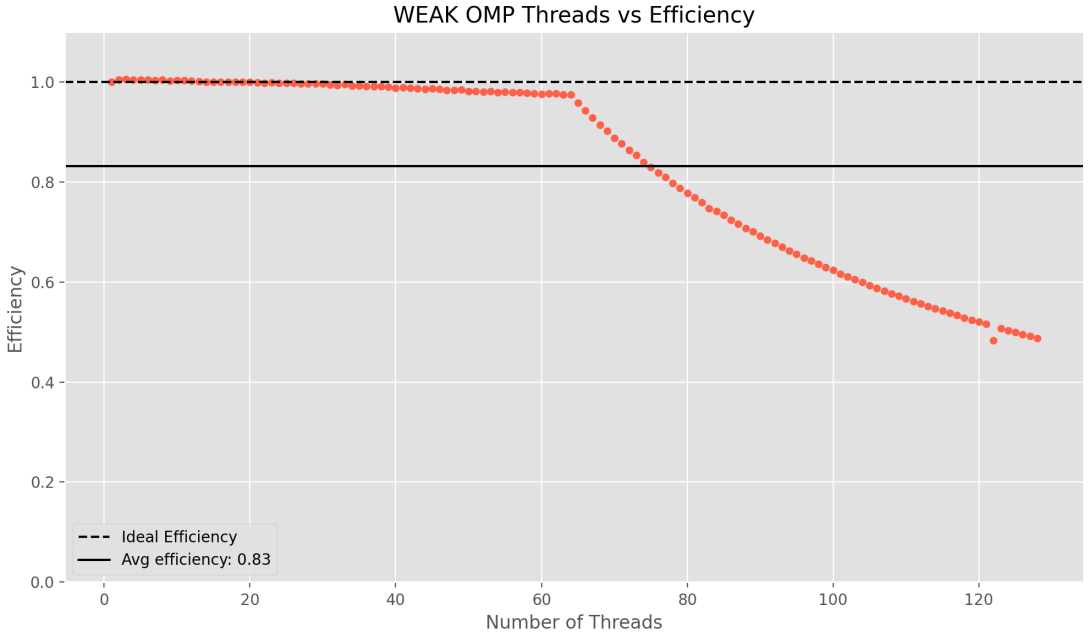
- Gustafson's law

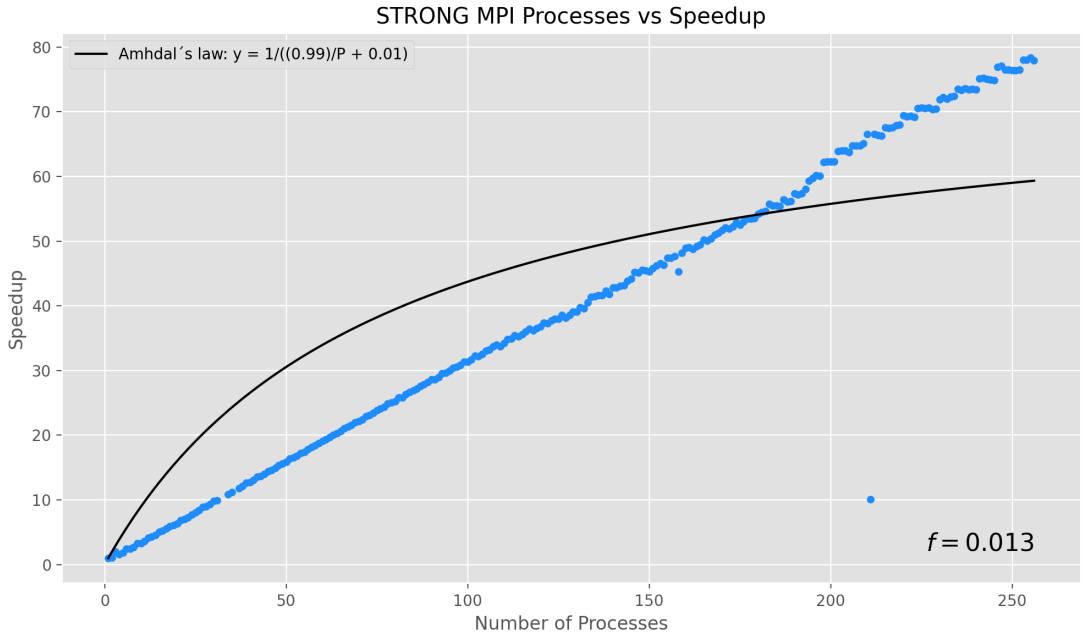
$$S_G(N) = f + (1 - f) \cdot N$$

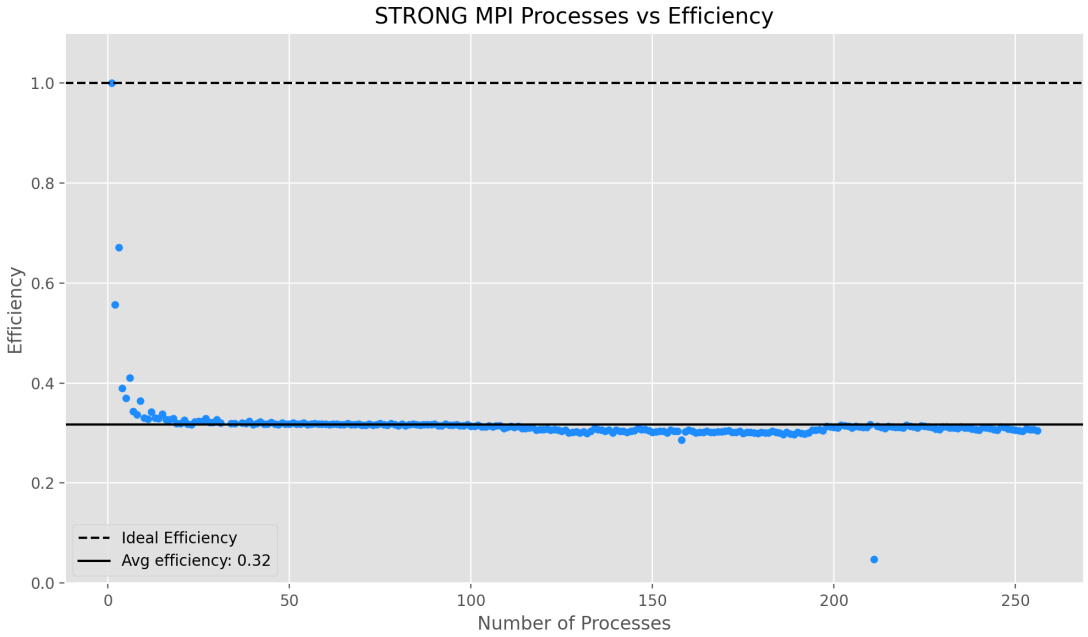

STRONG OMP Threads vs Speedup
Amhdal´s law: y = 1/((0.99)/T + 0.01)
$f = 0.006$
Speedup
Number of Threads

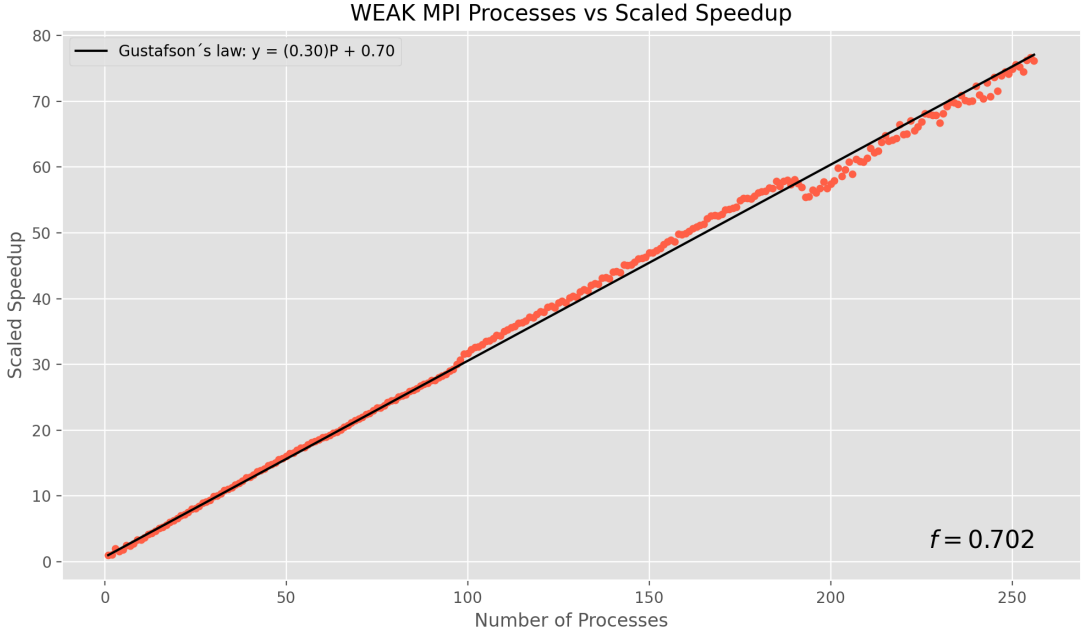

STRONG OMP Threads vs Efficiency
Ideal Efficiency
Avg efficiency: 0.83
Efficiency
Number of Threads

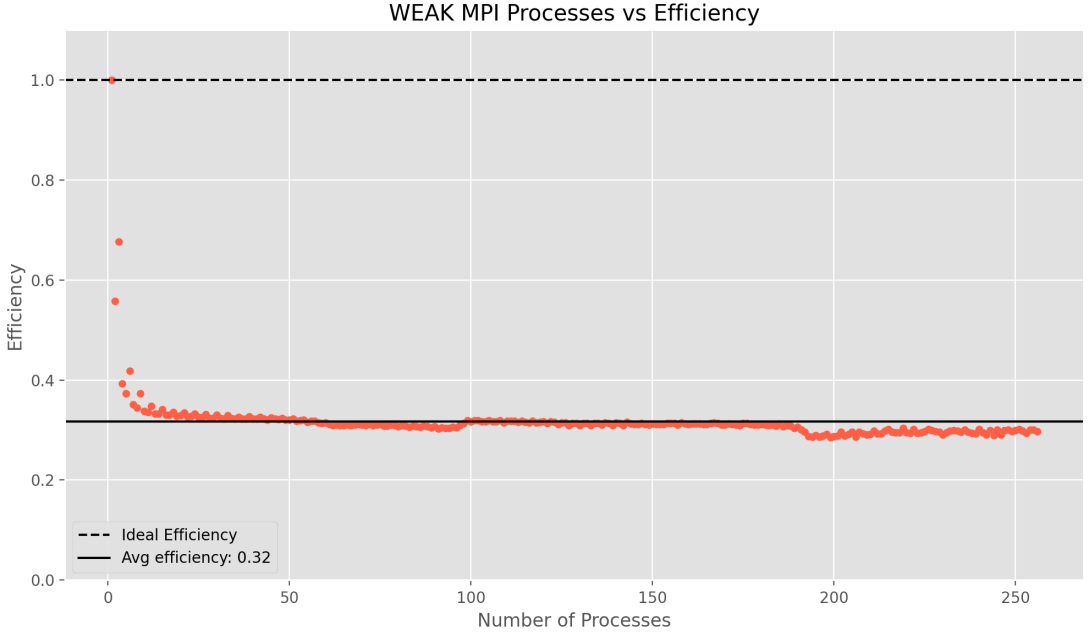

WEAK OMP Threads vs Scaled Speedup
Gustafson´s law: y = (0.66)T + 0.34
$f = 0.337$
Scaled Speedup
Number of Threads


WEAK OMP Threads vs Efficiency
Ideal Efficiency
Avg efficiency: 0.83
Efficiency
Number of Threads


STRONG MPI Processes vs Speedup
Amhdal´s law: y = 1/((0.99)/P + 0.01)
Speedup
Number of Processes
f = 0.013


STRONG MPI Processes vs Efficiency
Efficiency
Number of Processes
Ideal Efficiency
Avg efficiency: 0.32


WEAK MPI Processes vs Scaled Speedup
Gustafson´s law: y = (0.30)P + 0.70
Scaled Speedup
Number of Processes
f = 0.702


WEAK MPI Processes vs Efficiency
Efficiency
Number of Processes
Ideal Efficiency
Avg efficiency: 0.32

# Second attempt: dynamic (MPI) workload

- Despite the problem being embarassingly parallelizable the static approach doesn't totally exploit this property
- As the threads are **dynamically scheduled** we can do it with the processes too, using a **master/workers** solution:
  - A single process (e.g. `rank 0`) acts as a master, it distributes the work (a number of rows) to the processes whenever they are free
  - As a result there are processes which compute more rows (with lower complexity) and others which compute less rows (with higher complexity)
  - Master and workers interact with non blocking communications to reduce idle time
    - `MPI_Isend`: workers while computing send new request to master
    - `MPI_Iprobe`: master constantly controls incoming messages
  - Master collects the computed data whenever ready and writes on the image

# Master

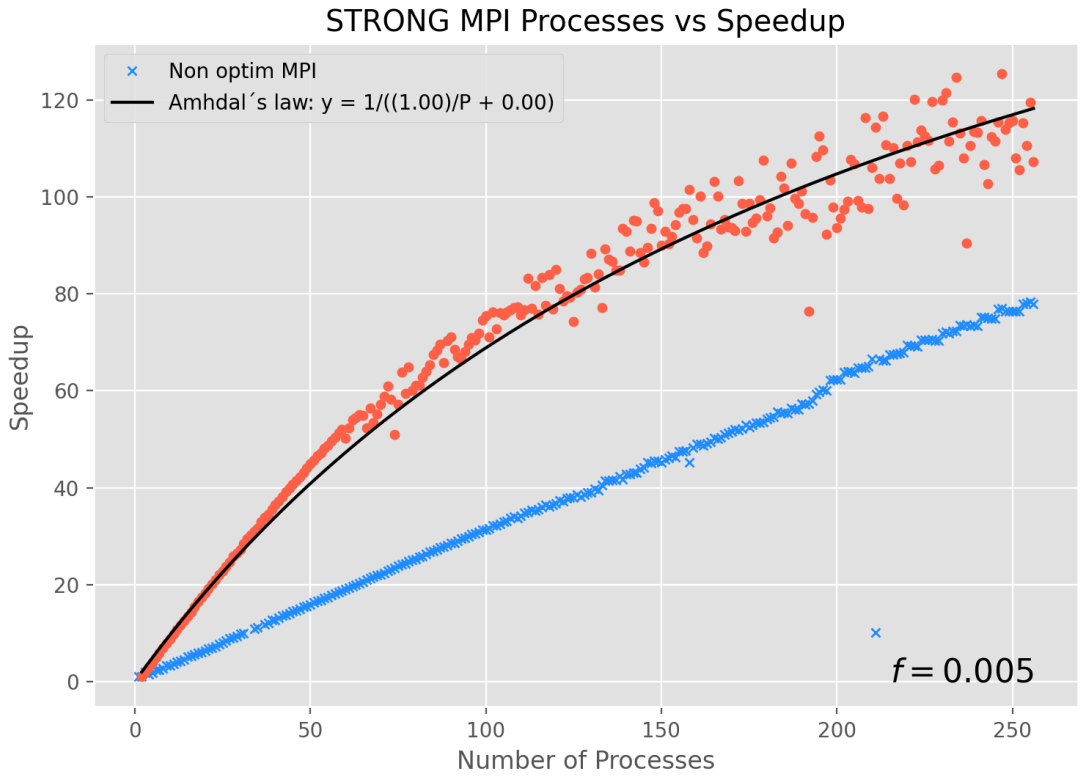
```
//...
#define ROWS_PER_REQUEST 10
/...
if (rank == 0) {
        // MASTER PROCESS
        // set rows to assign and count the active workers, allocate memory for the full image ...
        int pending_results = 0; // trace the number of pending results
        while (active_workers > 0 || pending_results > 0) { // ...
            MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &recv_status);
            if (flag) {
                if (recv_status.MPI_TAG == TAG_WORK_REQUEST) { // receive work request ...
                    MPI_Recv(&message, 1, MPI_INT, worker_rank, TAG_WORK_REQUEST, MPI_COMM_WORLD, &status);
                    if (next_row < ny) { // assign job
                      }
                        MPI_Send(&rows_to_assign, 1, MPI_INT, worker_rank, TAG_WORK_ASSIGNMENT, MPI_COMM_WORLD); // send job assignement
                        MPI_Send(&start_row, 1, MPI_INT, worker_rank, TAG_WORK_ASSIGNMENT, MPI_COMM_WORLD);
                        pending_results++;
                    } else { // If no work left send message of termination
                    }
                } else if (recv_status.MPI_TAG == TAG_WORK_RESULT) {
                    MPI_Recv(&rows_received, 1, MPI_INT, worker_rank, TAG_WORK_RESULT, MPI_COMM_WORLD, &status); // receive work results ...
                    // ...
                    pending_results--;
                }
        }
        // end the computation timer and start the I/O timer,
        //and the image writing, at the end stop the timer, print all the results and free space
    }
```

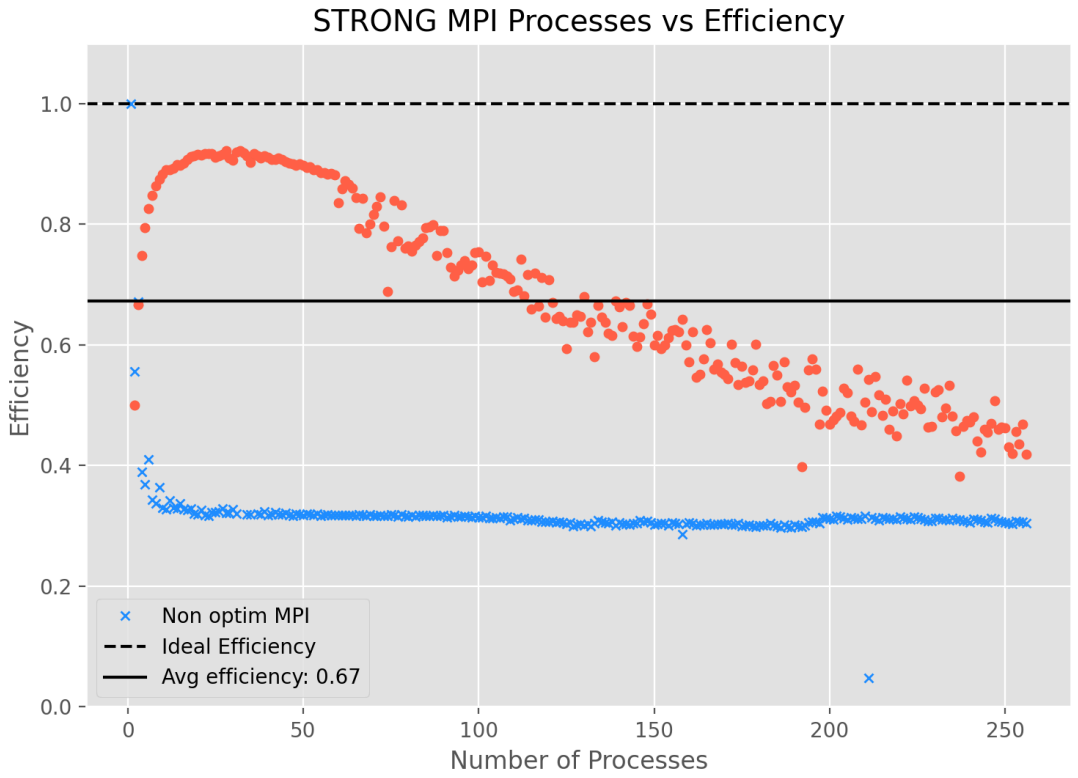
# Workers

```
else {
        // WORKERS
        //...

        MPI_Isend(&message, 1, MPI_INT, 0, TAG_WORK_REQUEST, MPI_COMM_WORLD, &send_request); // initial work request to the master
        while (1) {
            MPI_Recv(&rows_to_compute, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status); // receive work from  master
            if (status.MPI_TAG == TAG_WORK_ASSIGNMENT) {
                MPI_Recv(&start_row, 1, MPI_INT, 0, TAG_WORK_ASSIGNMENT, MPI_COMM_WORLD, &status);  // receive start_row
                unsigned char* rows_data = (unsigned char*)malloc(rows_to_compute * nx); // allocate memory for rows
                // ... error handling
                }
                MPI_Isend(&message, 1, MPI_INT, 0, TAG_WORK_REQUEST, MPI_COMM_WORLD, &send_request);
                #pragma omp parallel for schedule(dynamic)
                  // computation ...
                    }
                }
                // ... send computed data to master ...

        MPI_Wait(&send_request, MPI_STATUS_IGNORE); // barrier to wait end the of all computation before starting the new work
  }
```

# Results

## MPI Master/Slave Balancement



# Conclusions ...

- Balancing the workload is crucial to exploit the power of parallelization
- MPI requires more sophisticated strategies (e.g. master/Workers)
- OMP suffers out of a single socket

## ... and further improvements

- Testing more in deep binding and mapping policies
- Testing hybrid scaling strategies to assess their efficiency

# THANKS FOR THE ATTENTION