High Performance Computing - Part 2

Mandelbrot set implementation with a hybrid MPI and OpenMP code

Munini Matteo

October 2025

Abstract

The Mandelbrot set is computed using hybrid MPI+OpenMP parallelism: MPI distributes work across processes, while OpenMP threads compute pixels independently. Strong and weak scaling tests on Orfeo's EPYC nodes show effective intranode parallelism and balanced inter-node distribution, demonstrating suitability for HPC environments.

1 Introduction

This project develops two hybrid MPI+OpenMP Mandelbrot solvers using row- and column-wise domain decompositions. Both implementations adopt a hybrid MPI+OpenMP approach, in which MPI manages inter-process workload distribution while OpenMP exploits shared-memory parallelism within each process. Performance is tested under strong and weak scaling, analyzing speedup and efficiency against Amdahl's and Gustafson's Laws. The study evaluates scalability and the impact of decomposition strategy on hybrid parallel performance.

2 Architecture and Software

All experiments were conducted on the EPYC partition of the ORFEO high-performance computing 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.

The EPYC Rome microarchitecture design: each processor integrates 8 Core Complex Dies (CCDs), and each CCD contains two Core Complexes (CCXs), with 4 cores and 16 MB of shared L3 cache per CCX. Communication between CCDs is managed through a central I/O die, while inter-socket communication relies on AMD's Infinity Fabric.

The software used: the benchmarking experiments were executed using SLURM for job submission. The software stack included OpenMPI v4.1.6, the programming language used has been C, and OpenMP exploiting shared-memory parallelism within each process.

3 The math of Mandelbrot set

The *Mandelbrot set* is a fractal defined in the complex plane by

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

A point C belongs to the set if $|Z_k| \leq 2$ for all k. Iteration stops when $|Z_k| > 2$ or the limit $I_{\max}$ is reached, with the iteration count defining the pixel intensity. The region $[X_{\min}, X_{\max}] \times [Y_{\min}, Y_{\max}]$ is mapped to an image grid of size $W \times H$ by

$$X = X_{\min} + \frac{i}{W-1}(X_{\max} - X_{\min}), \quad Y = Y_{\min} + \frac{j}{H-1}(Y_{\max} - Y_{\min}) \tag{2}$$

Each pixel corresponds to $C = X + iY$, iterated independently up to $I_{\max} = 255$, producing a grayscale `PGM` image. It is important to notice that each pixel computation is independent.

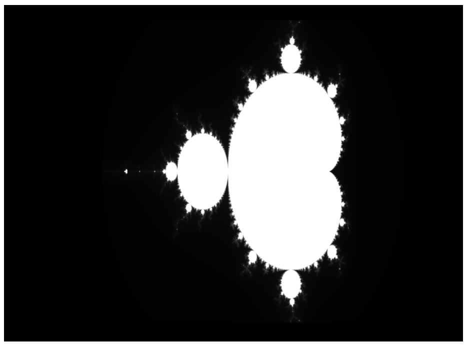


Figure 1: Mandelbrot set: Row strategy's output

4 Parallel Efficiency Theory

The model's performance is evaluated using two theoretical laws:

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:

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

5 Code design

These are two parallel implementations of a Mandelbrot fractal generator using hybrid MPI/OpenMP programming. They differ fundamentally in their domain decomposition strategy: one divides work by columns (vertical strips), the other by rows (horizontal strips). Those two implementations will be refereed as *column strategy* and *row strategy* respectively.

5.1 Common choice for row and column strategy code

These programs employ a hybrid two-level parallelization strategy, which will differ only on the chosen domain. Indeed, 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. This centralized approach simplifies debugging, guarantees correct PMG output formatting, and 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.

For both strategies it has been used the same core algorithm *compute_mandelbrot()*:

```
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;
}
```

This function computes the number of iterations for a point in the Mandelbrot set, and it returns how quickly the point diverges. It is used to generate Mandelbrot images. The `static inline` allows fast, file-local execution of the function, while the `unsigned char` efficiently stores values from 0 to 255 for coloring later the generated image (Figure 1). Firstly, there is the MPI initialization with:

```
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_Init` function initializes the MPI environment and must be called before any other MPI operations, taking command-line arguments to allow MPI-specific configuration. The `MPI_Comm_rank` function assigns each process a unique identifier (rank) ranging from 0 to n-1, stored in `process_id`, which allows each process to determine its role in the parallel computation. 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. Together, these three functions establish the foundation for parallel programming by giving each process both its unique identity and awareness of the collective computing environment. Finally, to evaluate the performances of these models, it is important to store the time needed to complete the task assigned. This is how it is done in the code:

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

The `double` allows to store the seconds, with microsecond precision. The goal of this section in the code is to measure wall-clock time (real elapsed time), where all MPI processes use same time base (synchronized during `MPI_Init`).Measured time includes parallel computation (OpenMP loops), MPI communication, synchronization overhead (barriers), and load imbalance (waiting for the slowest process); it does not include file I/O operations (after timing), memory allocation (before the start time), or MPI initialization.

5.2 Row strategy code details

Let's see how the code handles automatically the problem of not evenly division of the images among processes.

```
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);
```

This code implements a load balancing algorithm that evenly distributes image rows among MPI processes. `base_strips` defines the base number of rows per process, while `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.

To allocate necessary memory on the local buffer:

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

Each MPI process allocates a private buffer of size `local_strips * img_width` to store pixel values as `unsigned char` (1 byte per grayscale pixel). Memory is allocated on the heap with `malloc` and must be freed to avoid leaks. The buffer uses row-major order, storing all pixels of each row contiguously, which enables efficient sequential access and simplifies the subsequent MPI gather operation (data locality principle).

To effectively enable the hybrid parallelization, this code chunk allows to parallelize for loops across multiple threads:

```
#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);
    }
}
```

This nested loop performs the Mandelbrot computation using OpenMP. The outer loop distributes rows among threads with `#pragma omp parallel for schedule(dynamic)` for load balancing, while inner iterations map coordinates to $[min\_x, max\_x]$ and $[min\_y, max\_y]$. Results are stored in the local buffer using row-major indexing for efficient memory access.

At this point, it will be presented how the metadata preparation is executed for `MPI_Gatherv` collection.

```
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
}
```

Before `MPI_Gatherv`, rank 0 constructs two arrays: `receive_sizes` and `data_offsets`, which define how data from each process fits into `final_image`. Using the same load-balancing formula, offsets are accumulated to ensure contiguous, gap-free data placement in the final buffer.

```
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);
```

`MPI_Gatherv` aggregates variable-sized data from all processes to the root (rank 0). Each process sends its `local_buffer` of size `local_strips * img_width`, while the root uses `final_image`, `receive_sizes`, and `data_offsets` to assemble the complete image. Data is transferred as `MPI_UNSIGNED_CHAR`. `Gatherv` handles uneven workloads, ensuring rows from all processes are collected correctly in a single communication phase. This is useful guarantee, not only for output quality but also for debugging issues.

5.3 Column strategy code details

The previous subsection discussed the code sections tailored to the row-based strategy. The same considerations apply to the column-based strategy; however, the implementation differs. Only the modified code is shown here to highlight these differences, as the underlying explanations remain unchanged. This is how the balancing problem of image partiton over ranks is handled:

```
int base_strips = img_width / total_processes;      // ← width
int extra_strips = img_width % total_processes;      // ← width
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);
```

Here, how the buffer is allocated:

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

This code enables the hybrid parallelization in column startegy:

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

These last two code chunks are used for metadata and data preparation for final aggregation of information and gather communication:

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

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

5.4 Code in the GitHub repository

The *Project 2/source_code* folder contains Mandelbrot solvers for row and column strategies (*parallel_on_rows_code.c*, *parallel_on_columns_code.c*) and four bash scripts per strategy to test MPI and OpenMP strong and weak scaling.

6 Experiment design

This project analyzes weak and strong scaling of the parallel program. Four experiments are defined for each strategy:

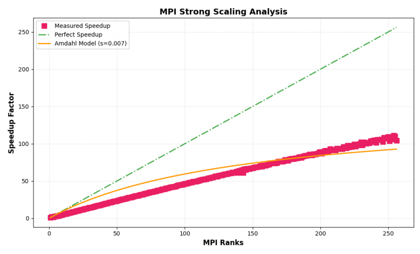
- **MPI Strong Scaling:** Up to 256 processes on 2 nodes compute subsets of a 10K×10K Mandelbrot set with `OMP_NUM_THREADS=1` to isolate MPI performance.
- **MPI Weak Scaling:** Keeps 10^6 pixels per core, scaling the grid as $\sqrt{\text{CORES} \times 10^6}$ (1K×1K to 16K×16K pixels). Processor affinity 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 10^6 pixels per thread, scaling the grid as $n = \sqrt{\text{THREADS} \times 10^6}$ (1K×1K to 11K×11K pixels).

7 Analysis of the results - Row strategy

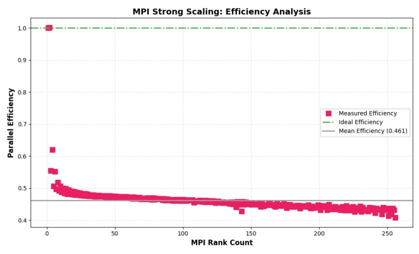
In this section the four kind of scaling tests will be reported for the row strategy.

7.1 Strong scaling

The first strong scaling analysis will be on the MPI. On Figure 2 it is reported the speed up and efficiency under the Amdahl's law:



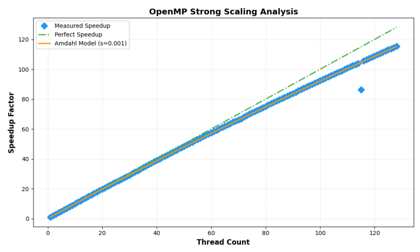
(a) Linear



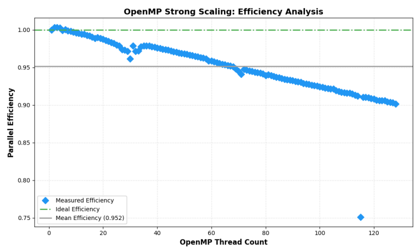
(b) efficiency

Figure 2: MPI: Speed up and Efficiency of strong scaling experiment on row strategy

On Figure 3 it is analyzed the strong scaling on the OpenMP:



(a) speed up



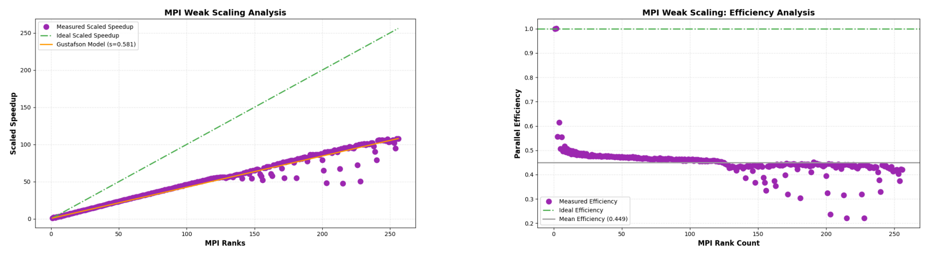
(b) efficiency

Figure 3: OpenMP: Speed up and Efficiency of strong scaling experiment on row strategy

Strong Scaling Summary: MPI strong scaling is limited, with mean efficiency around 45% and there is evidence of a serial fraction due to communication and synchronization overhead. OpenMP scales much better, with efficiency averaging 95%, and a linear behavior of the speedup (low serial fraction).

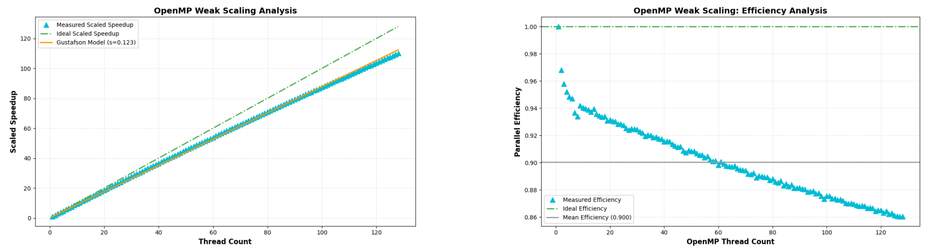
7.2 Weak scaling

On this section it will be present the analysis on the weak scaling experiment. On Figure 4 it is reported the weak scaling on MPI, while on Figure 5 on OpenMP.



(a) speed up (b) efficiency

Figure 4: MPI: Speed up and Efficiency of weak scaling experiment on row strategy



(a) speed up (b) efficiency

Figure 5: OpenMP: Speed up and Efficiency of weak scaling experiment on row strategy

Weak Scaling Summary:

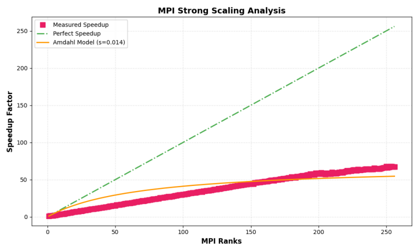
MPI weak scaling shows an increase in variability in efficiency with the number of ranks increased. There is evidence of a growing overhead with problem size and likely caused by network, memory, or load imbalance issues. OpenMP weak scaling remains robust, with efficiency up to a mean of 90%, and an higher serial fraction than strong scaling.

8 Analysis of the results - Column strategy

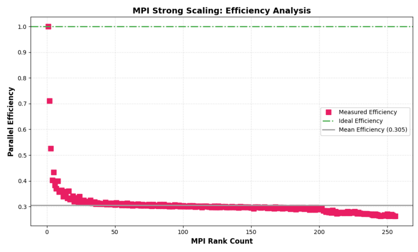
In this section the four kind of scaling tests will be reported for the column strategy.

8.1 Strong scaling

On Figures 6 and 7, the strong scaliong on MPI and on OpenMP respectivelly,

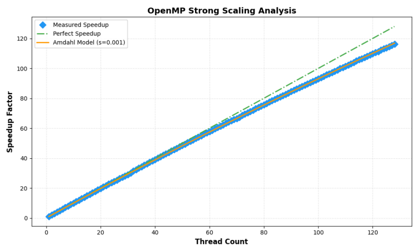


(a) speed up

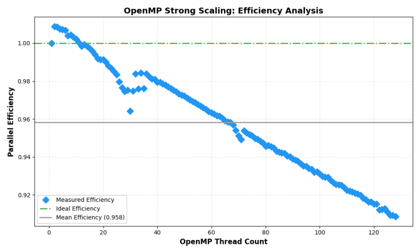


(b) efficiency

Figure 6: MPI: Speed up and Efficiency of strong scaling experiment on column strategy



(a) speed up



(b) efficiency

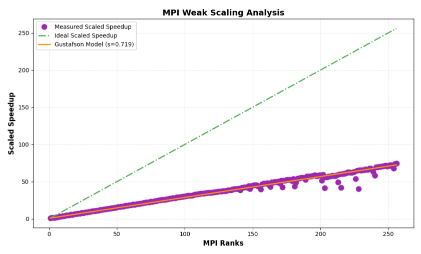
Figure 7: OpenMP: Speed up and Efficiency of strong scaling experiment on column strategy

Strong Scaling Summary:

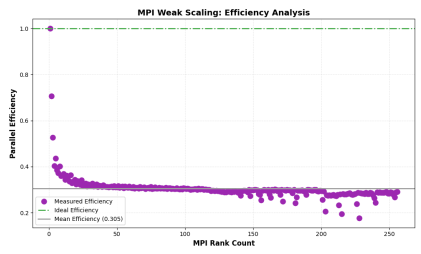
MPI strong scaling shows an average efficiency around 30.5%, showing that communication, synchronization, and load imbalance dominate performance. OpenMP performs even here much better, maintaining an average efficiency up to 95%, with speedup that exhibits a linear behavior. Overall, OpenMP achieves 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.

8.2 Weak scaling

In this subsection the results from weak scaling are reported:

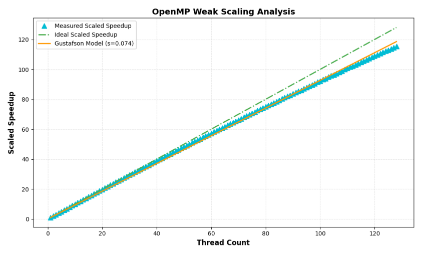


(a) speed up

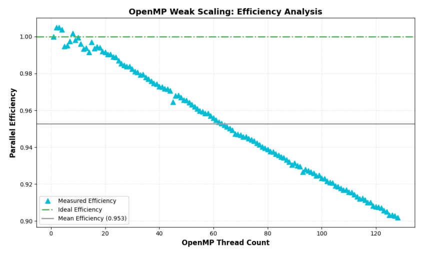


(b) efficiency

Figure 8: MPI: Speed up and Efficiency of weak scaling experiment on column strategy



(a) speed up



(b) efficiency

Figure 9: OpenMP: Speed up and Efficiency of weak scaling experiment on column strategy

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.

9 Conclusion

Hybrid MPI+OpenMP accelerates Mandelbrot computations. OpenMP scales efficiently on a single node, while MPI is limited at high ranks by communication and synchronization. Scaling analyses confirm the algorithm is parallelizable, though performance depends on strategy and hardware. Future work could explore larger problem sizes and alternative data aggregation methods.