Final Project HPC - Exercise 2c

Mandelbrot Set Computation
Hybrid Implementation

Emanuele Ruoppolo
Università degli Studi Trieste
emanuele.ruoppolo@studenti.units.it

Abstract

The aim of this project is to implement a hybrid MPI+OpenMP version of the Mandelbrot set computation algorithm, leveraging MPI for distributed memory parallelism and OpenMP for shared memory parallelism, and verifying the scaling performances of the code on the ORFEO cluster. The code was implemented using the C programming language and the MPI and OpenMP libraries. The performances were evaluated by measuring the speedup and efficiency of the code for different numbers of processes and threads. Both strong and weak scaling tests were conducted, either by fixing the MPI tasks and increasing the OMP threads or running a single OMP thread per MPI task and increasing the number of MPI tasks.

1 Introduction

The Mandelbrot set 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. Starting from $z = 0$, this 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 $\mathcal{M}$, known as the Mandelbrot set, comprises all complex points c for which this sequence remains bounded. It can be demonstrated that if any term z_i in the sequence satisfies $|z_i| > 2$, the sequence will diverge to infinity. To determine whether a point c is part of $\mathcal{M}$, the following condition is used:

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

where $I_{\max}$ is the maximum number of iterations after which c is considered to belong to $\mathcal{M}$ if the sequence has not diverged. The Mandelbrot set can be then visualized by assigning a color to each point c based on the number of iterations required to determine its membership in $\mathcal{M}$.

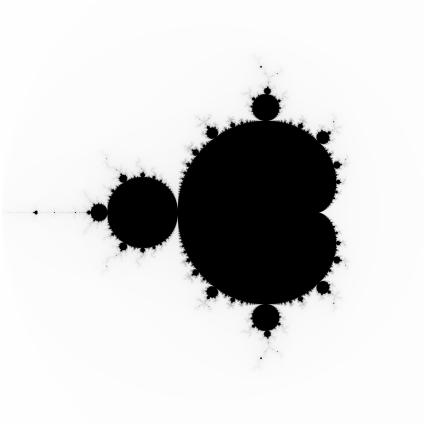

Figure 1: The Mandelbrot set, in the region $[-2, 2] \times [-1.5, 1.5]$, output from the hybrid program.

2 Hybrid strategy

2.1 Distributed Memory (MPI) Parallelization

To parallelize the computation of the Mandelbrot set across multiple processors, we utilized MPI to distribute the workload along the vertical axis of the image grid. The total number of rows (`ny`) was divided among the available MPI processes, ensuring that each process handled a contiguous block of rows. This division accounted for any remainder when `ny` was not perfectly divisible by the number of processes, resulting in some processes computing an extra row to maintain load balance. The starting row and the number of local rows for each process were calculated using the rank and size provided by `MPI_Comm_rank` and `MPI_Comm_size`.

For output, we implemented MPI I/O functions such as `MPI_File_open`, `MPI_File_write_at`, and `MPI_File_close` to allow each process to write its computed pixel data directly to the appropriate offset in the output file. The offset was calculated using `MPI_Offset`, considering the header size and the position of each process's data within the image. This approach eliminated the need for gathering data on a single root process and reduced communication overhead. Comprehensive error handling was included after each MPI I/O operation by checking the return codes and using `MPI_Error_string` to interpret any errors, ensuring robustness in file operations.

2.2 Shared Memory (OpenMP) Parallelization

Within each MPI process, we leveraged OpenMP to exploit shared-memory parallelism for the computationally intensive task of determining point membership in the Mandelbrot set. The number of threads was set using `omp_set_num_threads(num_threads)`, and dynamic adjustment of threads was disabled with `omp_set_dynamic(0)` to maintain consistent thread usage. The nested loops over the image grid were parallelized using the directive:

```
1     #pragma omp parallel for schedule(dynamic)
```

The `schedule(dynamic)` clause was crucial because the computational load per pixel varied significantly; some points required many iterations to determine divergence, while others did not. Dynamic scheduling allowed for more efficient load balancing among threads by redistributing iterations that took longer to compute. Each thread independently executed the `compute_mandelbrot` function for different pixels, writing the results into the shared `local_image` array. This method maximized CPU utilization within each node by effectively distributing the workload among the available cores.

3 Scaling

In the MPI strong scaling tests, we varied the number of MPI processes from 1 up to 256 across two EPYC nodes. The tests have been performed specifying `--nodes=2` and `--ntasks-per-node=128` to fully utilize the cores on both nodes. Within each MPI process, we set the number of OpenMP threads to one (`THREADS=1`) to isolate the performance of MPI parallelization. For the OpenMP weak scaling tests, we fixed the number of MPI processes at one and varied the number of OpenMP threads from 1 to 128 on a single EPYC node. The SLURM script then included `--cpus-per-task=128` to allocate all available CPU cores to the single task. In the OMP scalings the option `--map-by socket` was selected so that all OpenMP threads are kept on the same physical processor socket. This improves memory access times and reduces the overhead of communication between sockets, enhancing performance as the number of threads is increased.

3.1 Strong Scaling

To assess the strong scaling performance of our hybrid MPI and OpenMP program for computing the Mandelbrot set, we conducted experiments while keeping the problem size constant and varying the number of processing units. We selected an image size of 10000×10000 pixels, corresponding to approximately 100MB, which provided a substantial computational workload to effectively utilize the available computational resources. This size was chosen to ensure that the computation time was significant enough to observe the effects of adding more processors or threads. The maximum number of iterations for determining point membership in the Mandelbrot set was set to 255, so each pixel has been stored as 1 byte, maintaining consistency across all test runs.

In the MPI case we employed the `mpirun` options `--map-by numa` and `--bind-to core` to map each MPI process to a NUMA domain and bind it to a specific core. This configuration optimized memory access patterns by ensuring that each process accessed memory local to its NUMA region, reducing latency and improving performance.

For the OpenMP strong scaling experiments we set the OpenMP environment variables

```
export OMP_PROC_BIND=close
export OMP_PLACES=cores
```

to control thread placement and binding, ensuring that threads were bound to individual hardware threads and placed close to each other to enhance cache utilization. Since the code utilized the `#pragma omp parallel for schedule(dynamic)` directive to parallelize the nested loops over the image grid, the OpenMP environment variable `OMP_WAIT_POLICY=active` was set, in order to minimize idle time. This policy ensures that threads remain ready to quickly handle new tasks, reducing delays caused by waiting for additional work.

3.2 Weak Scaling

To evaluate the weak scaling performance of our hybrid MPI and OpenMP implementation for computing the Mandelbrot set, we proportionally increased both the problem size and the number of processing units. This approach maintains a constant workload per processor, allowing us to assess how efficiently the application handles larger problems when more resources are added. The problem size was determined by the total number of pixels in the generated image, calculated as $n \times n$, where n is the number of pixels along one dimension. We set the per-processor workload to approximately one million pixels, ensuring that each processing unit had a consistent amount of work regardless of the total number of processors or threads. The parameter n was dynamically adjusted based on the number of workers W, being them processors P in MPI scaling or threads T for OMP scaling, using the relation $n = \sqrt{W \times C}$, where C is a constant representing the workload per processor.

In the MPI case we employed `mpirun` options `--map-by core` and `--bind-to core` to assign each MPI process to a specific core, optimizing CPU utilization and minimizing context switching.

This configuration enabled us to effectively assess the scalability of our application under weak scaling conditions, demonstrating its ability to handle larger problem sizes efficiently as more computational resources are added.

4 Results

First, the time breakdown for strong and weak scaling can be observed in figures 2 and 3 respectively. The computing time refers to the duration spent within the `compute_mandelbrot` function, whereas the I/O time denotes the time spent within the MPI I/O operations. As evidenced by the plots, the I/O strategy is highly efficient. As each process writes its own data to the output file, the I/O almost occurs instantaneously after the computing time. It is evident that the MPI scalings exhibit a certain degree of overhead, particularly when the second node is utilized. Nevertheless, it can be posited that this strategy is more advantageous than a gather operation, wherein a single process is tasked with writing the entirety of the image. The remainder of the analysis will concentrate on the computing time, as the I/O time is not the primary objective of the project.

In order to evaluate the scaling performance of the code, two aspects are examined: **speedup** and **efficiency**. In the **strong scaling** case, the speedup is calculated as the ratio between the serial time and the parallel time. The serial time, denoted by $T(1)$, is defined as the time taken to run the program using a single process or thread. In contrast, the parallel time, denoted by $T(N)$, is defined as the time to execute the same program using N workers. The efficiency is calculated as the ratio between the speedup and the number of processes or threads.

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

The experimental results can be fitted with Amdahl's law in order to assess if it's worth to increase the number of workers. Also it is useful to identify eventual bottlenecks caused by serial code and address inefficiencies in the resources utilization. Amdahl's law is defined as:

$$S_A(N) = \frac{1}{f + (1-f)/N} \tag{2}$$

Where f is the fraction of the code that cannot be parallelized and is forced to be serial and $1 - f$ the fraction that can be parallelized.

In the **weak scaling** case, since the problem size increases proportionally with the number of workers, we can define a **scaled speedup** and the efficiency as:

$$S(N) = \frac{T(1)}{T(N)} \cdot N \quad \text{and} \quad E(N) = \frac{S(N)}{N} \tag{3}$$

Then the experimental results can be fitted with Gustafson's law. Performance analysis through Gustafson's law provides insights into how problems can expand to utilize additional computing resources effectively. By examining scaling behavior as both workload and parallel processors increase proportionally, we can better justify the using of a larger computing system. Gustafson's law is defined as:

$$S_G(N) = f + (1 - f) \cdot N \tag{4}$$

Where f is the fraction for sequential program and $1 - f$ the fraction for parallel program.

Prior to an analysis of scaling performances, the following table presents statistical data for both strong and weak scaling tests. It should be noted that the "average I/O overhead" represents the mean ratio between I/O time and total time. This metric provides insight into the extent to which I/O time impacts overall execution time. Additionally, the "peak performance" denotes the maximum ratio between problem size and total execution time, measured in millions of pixels per second. This can be regarded as a gauge of the program's "speed," reflecting the effectiveness of load balancing.

	Strong Scaling		Weak Scaling	
	MPI	OMP	MPI	OMP
Avg. Efficiency	31.7%	82.9%	31.7%	83.1%
Compute Time Range (s)	0.17 - 13.45	0.21 - 13.47	0.14 - 0.47	0.14 - 0.28
I/O Time Range (s)	0.04 - 1.38	0.03 - 0.76	0.01 - 1.6	0.01 - 0.07
Average I/O Overhead	51%	12%	47%	12%
Peak Performance (M pixels/s)	0.02	0.04	0.02	0.05

Table 1: Example table with MPI and OMP statistics.

We already see a clear difference between the two scalings, with the OMP strategy showing a better efficiency and a better peak performance. Later will be discussed some possible reasons for this difference.

In figures 4, 5, 6 and 7 we can see the speedup and efficiency comparison for strong and weak scaling. We notice again that both different scaling strategies, for MPI and OMP cases, show a clear advantage in choosing the OMP, which clearly emergeges to be better in both speedup and efficiency.

Finally in figure 8 we can see the speedup and efficiency analysis for the four different performed scalings. We fitted the experimental results with Amdahl's law for the strong scaling and with Gustafson's law for the weak scaling. The fitting parameters are reported in the following table and in the plots.

	Strong Scaling		Weak Scaling	
	MPI	OMP	MPI	OMP
f	0.01	0.01	0.34	0.70
$1 - f$	0.99	0.99	0.66	0.30

Table 2: Parallelization factors.

In the strong scaling case, it is evident that the proportion of the program that cannot be parallelized is minimal. Consequently, a program that has been optimized to a high degree will be highly efficient in exploiting the parallel resources. It is evident that the optimization of this problem hinges on the varying degrees of effort required for the computation of the Mandelbrot set at each point. It is evident that the frontier and inner points of $\mathcal{M}$ are more computationally demanding than the outer points. The OMP strategy provides a straightforward means of addressing this issue by dynamically scheduling threads to the points requiring more iterations through the use of the `schedule(dynamic)` clause. To enhance the MPI strategy, a potential avenue for improvement could be to subdivide the image into a more intricate structure, with a heightened focus on the critical regions. This approach would facilitate a more balanced distribution of the points that necessitate additional iterations among the processes.

Similarly, in the weak scaling case, the OMP strategy demonstrates enhanced parallelization, enabling the program to scale more effectively with an increasing number of workers. In any case, even within the OMP strategy, we observe a flattening of the speedup and a reduction in efficiency when the number of threads exceeds a single socket capacity (64). This could be due to the fact that the memory access time increases when the threads are distributed among different sockets, resulting in a greater overhead of communication between sockets. This is particularly relevant in the case of a dynamically redistributed load. Despite this issue, different mapping strategies, including those that deviate from the one chosen, demonstrate worse scaling results.

5 Conclusions

The hybrid MPI+OpenMP implementation of the Mandelbrot set computation algorithm demonstrated the complexities inherent in parallel programming. The primary conclusion that can be drawn is the necessity for an effective load balancing strategy, which is essential for achieving optimal scaling performance and utilizing computational resources in an efficient manner. Further optimizations could be implemented to address the bottlenecks identified in the scaling analysis, such as improving the distribution of the computational workload among the processes. Furthermore, hybrid scaling strategies could be implemented to seek and eventually exploit the advantages of both MPI and OpenMP parallelization, combining the strengths of distributed and shared memory parallelism. It was observed that within the same socket, the OMP strategy fully parallelized the problem, achieving a linear speedup and an efficiency approaching the ideal case. To achieve the same results with the MPI strategy, a more complex division of the image could be implemented, focusing more attention on the critical regions.

A Appendix: Images

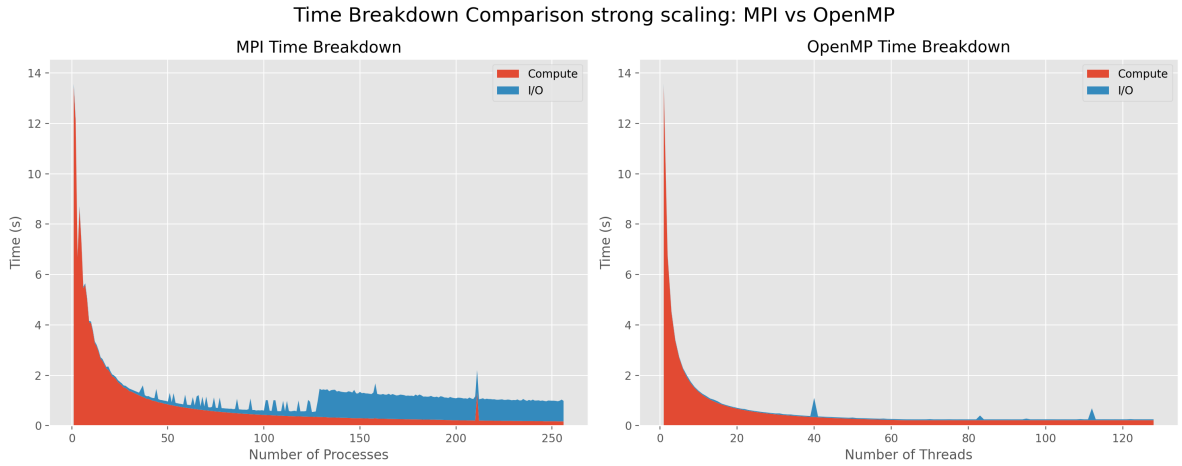

Figure 2: Comparison between computing time and I/O time for strong scaling.

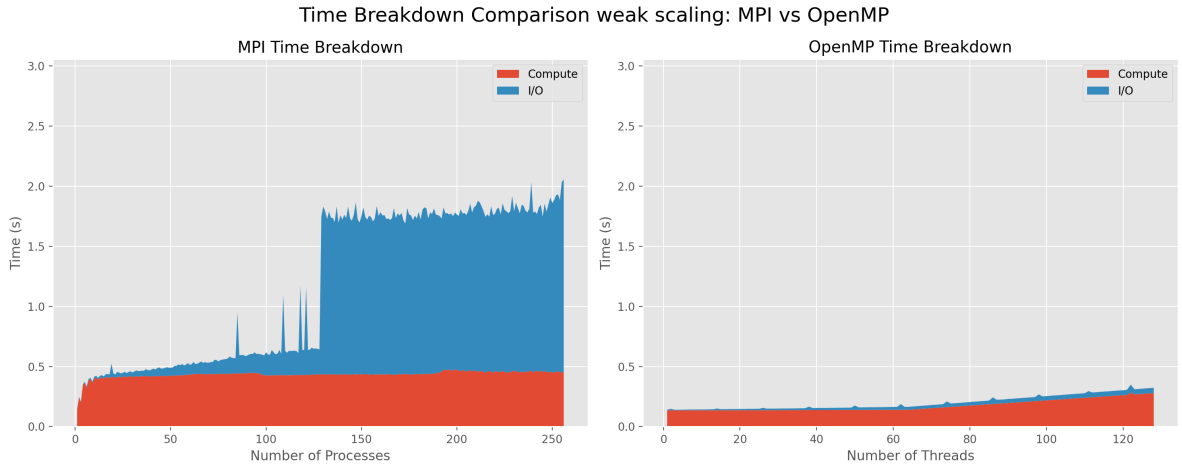

Figure 3: Comparison between computing time and I/O time for weak scaling.

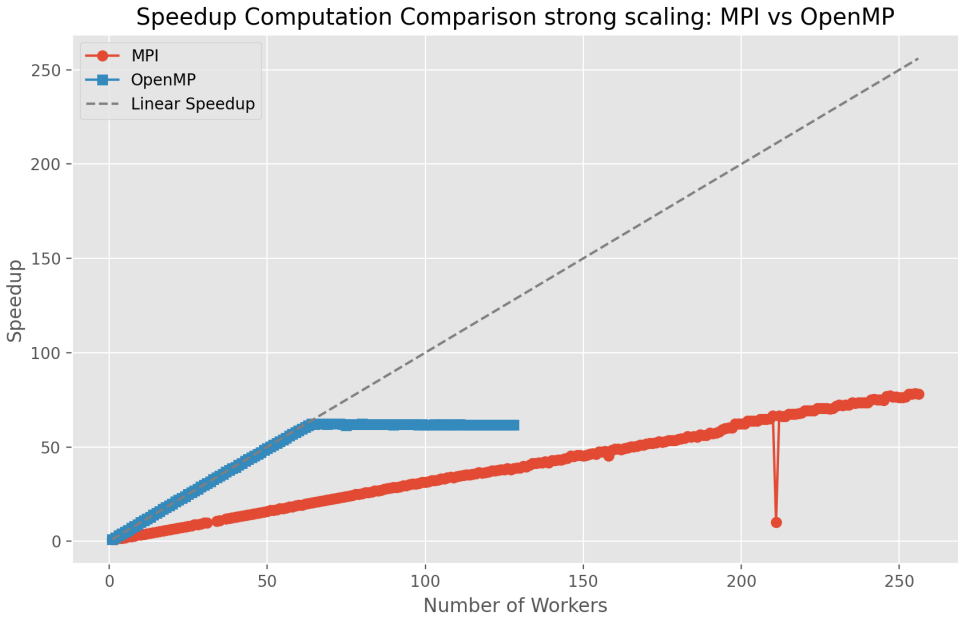

Figure 4: Speedup comparison for strong scaling.

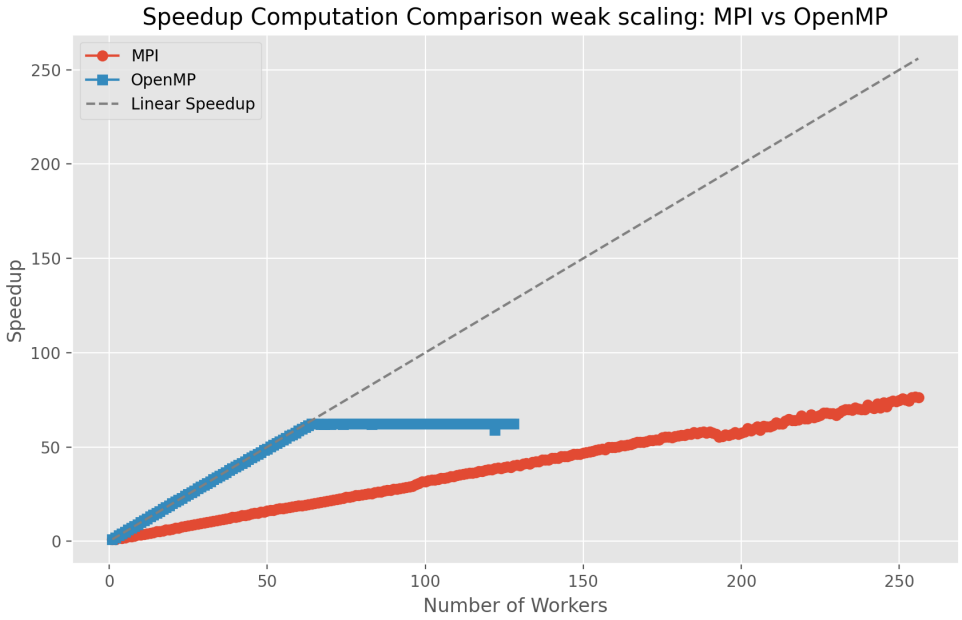

Figure 5: Speedup comparison for weak scaling.

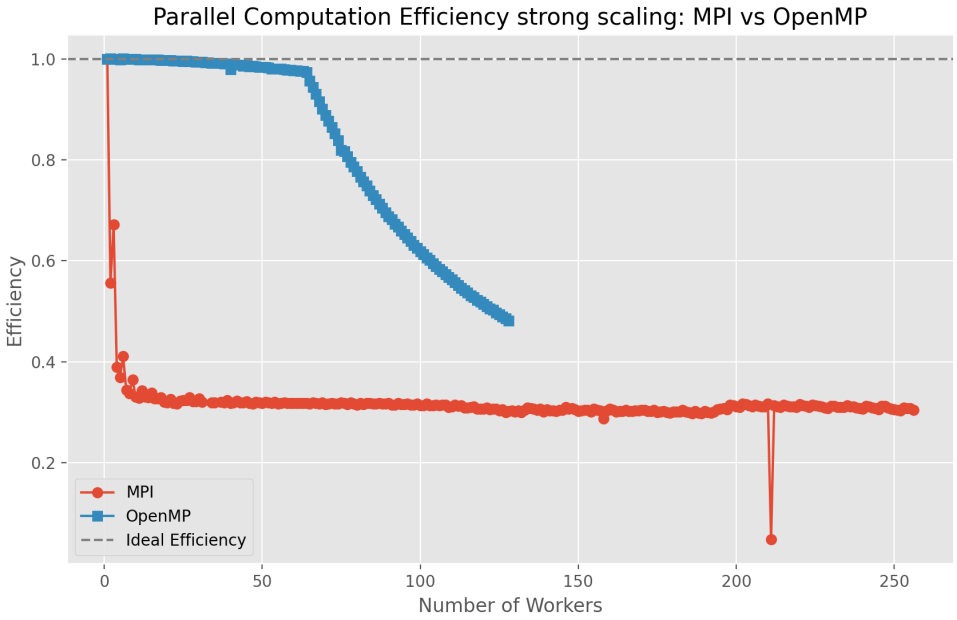

Figure 6: Efficiency comparison for strong scaling.

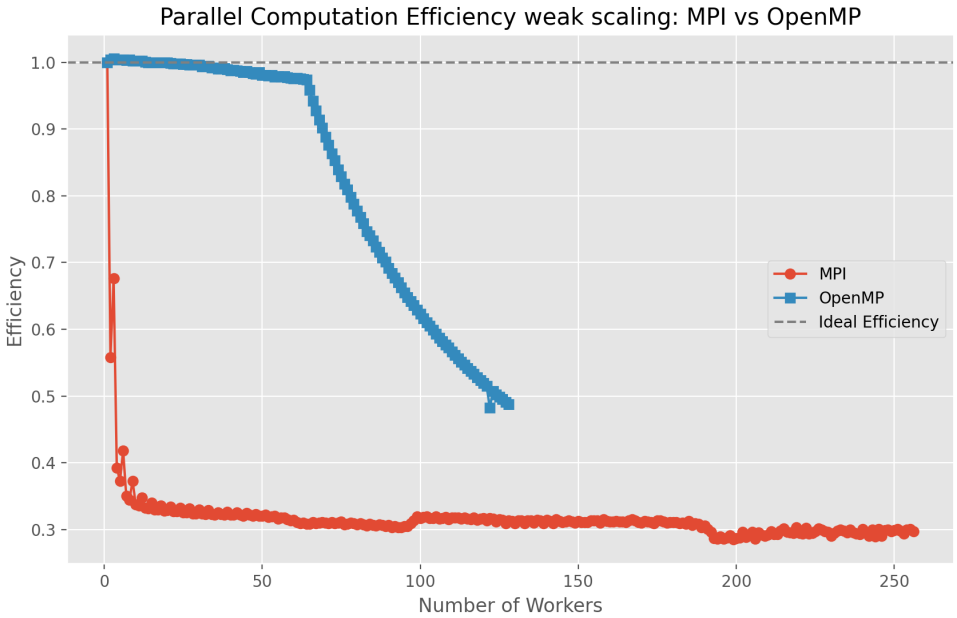

Figure 7: Efficiency comparison for weak scaling.

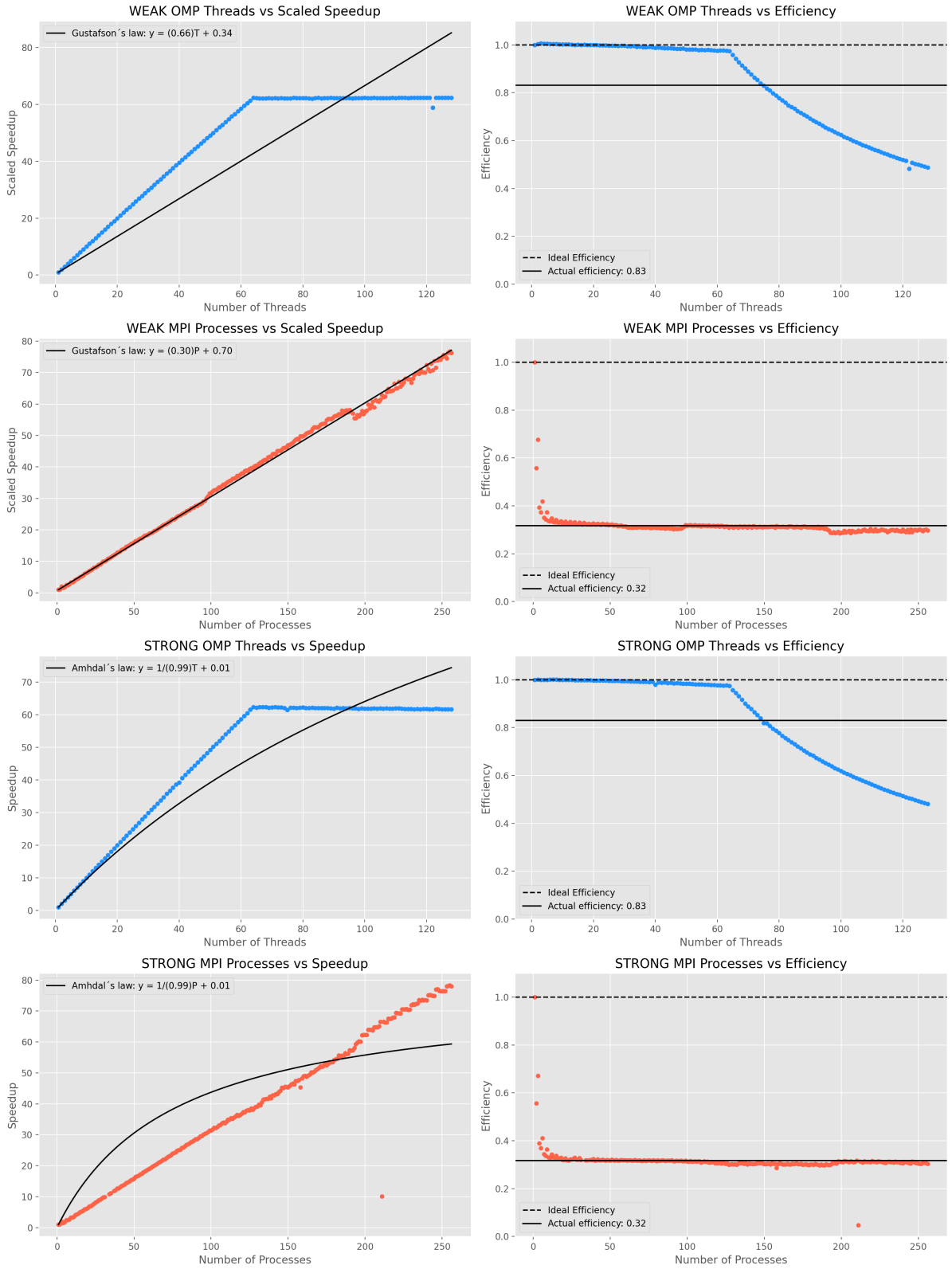

Figure 8: Speedup and efficiency analysis for the four different performed scalings.