Annalisa Paladino
SM3800021

High Performance Computing 2023 – Exercise 2C

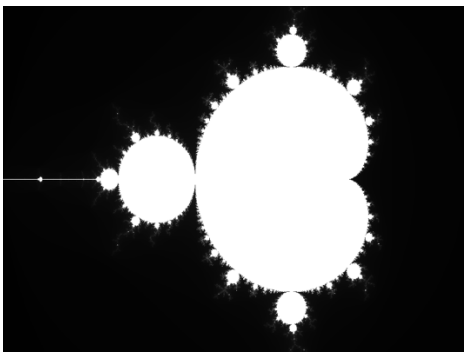
Introduction

The purpose of this project [1] is to implement a hybrid MPI + OpenMP code to compute and visualize the Mandelbrot set and to determine the strong and weak scaling of the code.
The Mandelbrot set $\mathcal{M}$ is constructed by iterating the complex function $f_c(z) = z^2 + c$ within the complex plane $\mathbb{C}$. The Mandelbrot set consists of those complex numbers c for which the iterations of $f_c(z)$, starting from $z = 0$, do not diverge. This means that the sequence $z_0 = 0, z_1 = f_c(z_0), z_2 = f_c(z_1), \ldots$ remains bounded in absolute value. If any term z_i exceeds the distance of 2 from the origin, the sequence will diverge. Thus, to generate $\mathcal{M}$, for any given section of the complex plane, $f_c(z)$ is iterated for each point c until the sequence either diverges or reaches a maximum iteration limit, I_{max}. The criteria to decide if a complex point c is within the Mandelbrot set is determined by the condition:

$$|z_n = f_c^n(0)| < 2 \quad \text{or} \quad n > I_{max}$$

Efficiently computing the Mandelbrot set in parallel is quite feasible since each point c is calculated independently, making $\mathcal{M}$ ideal for parallel computation. This report discusses a distributed memory parallel implementation using a hybrid approach that incorporates both Message Passing Interface (MPI) and OpenMP (OMP) for robust parallel computation. The analysis within will delve into the scaling behaviors, both strong and weak, of the computational approach, providing insights into its performance efficiency.

Below it's possible to see the Mandelbrot set in the $[-2, 1] \times [-1, 1]$ region of the complex plane computed in parallel with MPI and OpenMP.



[1] https://github.com/Foundations-of-HPC/High-Performance-Computing-2023

Set up

The implemented program has been tested to determine and assess the strong and weak scaling of the code; all the measurements have been conducted on the THIN nodes of the ORFEO [2] cluster, for a total of 24 cores (every THIN node has 2x12 cores) and 48 processes used.

Implementation

In this section the implementation of the algorithm will be discussed, for further details regarding the scripts, consult the repository [3].
This problem is ideally suited for parallel computation. To calculate the points of the set, the implementation effectively utilizes both distributed memory parallelization through the Message Passing Interface (MPI) and shared memory parallelization using Open Multi-Processing (OpenMP). The proposed code generates the Mandelbrot set using a hybrid parallel programming approach with both MPI (Message Passing Interface) and OpenMP (Open Multi-Processing).
First thing first, it's important to include the necessary headers for input-output operations, standard library functionalities, MPI and OpenMP libraries. The `mandelbrot` function calculates whether a point (x_0, y_0) is in the Mandelbrot set, given a maximum number of iterations. It returns the number of iterations it took to determine the point is outside the Mandelbrot set or the maximum iterations if it didn't escape.
The `main` function initiates with `MPI_Init_thread` by setting up the MPI environment with thread support specified to `MPI_THREAD_FUNNELED`, indicating that while the program may be multithreaded, only the main thread will make MPI calls. The current time is captured using `MPI_Wtime( )` to measure the total execution time. The program can accept command-line arguments to adjust various parameters like the dimensions, coordinate limits, maximum iterations, and the number of threads to be used for computation. Subsequently, the program retrieves the total count of processes and the respective identifier (ID) for each process within the `MPI COMM_WORLD` communicator. Based on this information, each process determines the specific segments of the Mandelbrot image it is responsible for generating, utilizing its unique ID and the total number of image rows. To achieve an equitable distribution of workload, the total number of rows in the Mandelbrot image is uniformly allocated among the processes. The final process is assigned any residual rows, ensuring complete coverage of the image generation task.
Utilizing OpenMP, each process concurrently computes its designated segments of the Mandelbrot image. The number of OpenMP threads engaged for this task is defined by invoking the `omp_set_num_threads` function, which establishes the quantity of threads that will operate in parallel within each MPI process. The `#pragma omp parallel for` directive is employed to automatically allocate the workload across the available threads within a process, facilitating parallel execution. This method enhances the efficiency of resource utilization on each node and expedites the generation of the image.

[2] https://orfeo-doc.areasciencepark.it
[3] https://github.com/annalisapaladino/High-Performance-Computing

After computation, the partial results from all processes are collected and assembled into a single image array on the root process using `MPI_Gatherv`. This root process then writes the complete image to a PGM file, a simple grayscale image format.
The program concludes by shutting down the MPI environment through `MPI_Finalize`. This approach effectively demonstrates the use of hybrid parallel programming by combining distributed memory parallelism through MPI and shared memory parallelism via OpenMP, which optimizes the program's efficiency and scalability in computing large-scale Mandelbrot images.

Scaling

Before analyzing the scaling for both MPI and OMP, let's see what strong and weak scaling are:

- **Strong scaling** is a concept in parallel computing that measures the efficiency of a system when solving a fixed-size problem with an increasing number of processors. Essentially, strong scaling assesses how the solution time decreases as more processing resources are applied to the same problem. The ideal goal in strong scaling is the speedup being directly proportional to the number of processors P, resulting in an efficiency equal to 1, where efficiency is the ratio of speedup to the number of processors. Speedup is defined as the ratio of the time required for one processor (approximately equivalent to the sequential time) to execute the application, compared to the time required for P processors

  $$S(P) = \frac{t(1)}{t(P)}$$

  If doubling the number of processors halves the computation time, the system exhibits good strong scaling. Strong scaling is particularly important in scenarios where the time to solution is critical and cannot be compromised by extending the problem size.

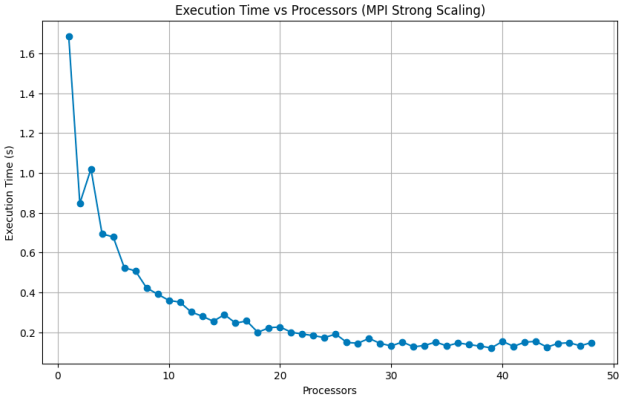
- **Weak scaling** is a concept in parallel computing that measures the efficiency of a system when increasing the size of the problem proportionally to the number of processors. Unlike strong scaling, where the total workload remains constant, weak scaling increases the workload in such a way that the amount of work handled by each processor remains constant as more processors are added. According to Gustafson's law, weak scaling evaluates how the execution time of a parallel application remains nearly constant as the number of processing elements increases in direct proportion to the problem size. Weak scaling is crucial in scenarios where the problem size naturally grows with the resources available, such as in simulations that need to maintain a certain resolution or detail as they scale.

Strong Scaling

In the implementation of strong scaling utilizing MPI processes, the dimensions of the image, specifically the number of rows and columns, were set at 1600 and 2400, respectively, while employing a single OMP thread. Conversely, for the implementation of strong scaling using

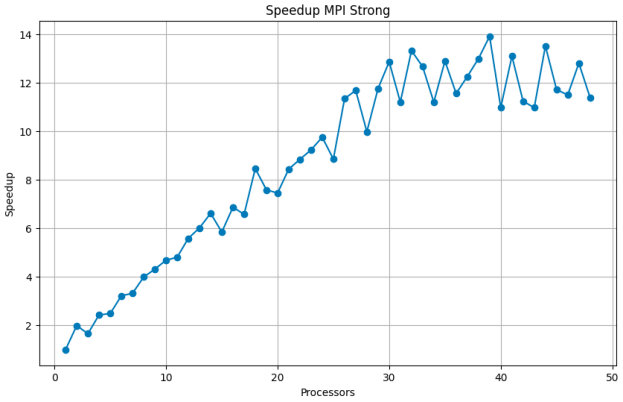
OMP threads, the image dimensions, specifically the number of rows and columns, were maintained at 600 and 800, respectively, with a single MPI process held constant.

MPI strong scaling

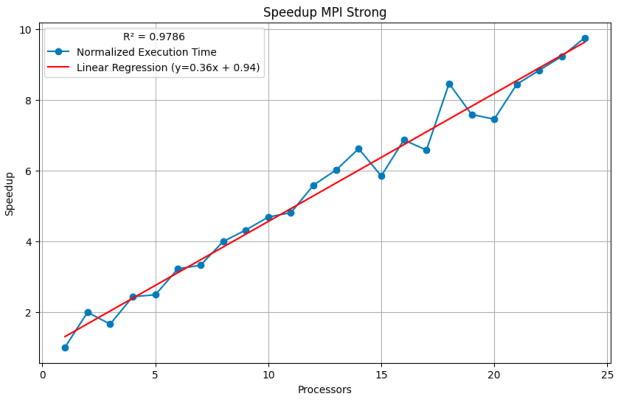


The plot shows a sharp decrease in execution time as the number of processors increases, particularly noticeable from 1 to around 10 processors. After this initial steep decline, the reduction in execution time begins to level off, with marginal decreases as the number of processors continues to increase past 10.
The execution time is highest when the number of processors is very low. As more processors are added, the execution time decreases significantly, indicating that the parallelization of the task is effective, especially at lower numbers of processors. The curve flattens out as the number of processors increases, suggesting a diminishing return on adding more processors. This is typical in strong scaling scenarios where the overhead of managing an increasing number of processors may begin to outweigh the benefits of parallel execution for a fixed-size problem.

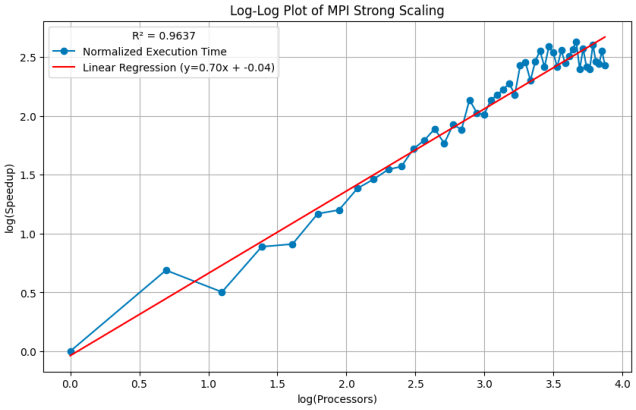
To further analyze the results, speedup and efficiency were calculated.



The plot shows a consistent increase in speedup as the number of processors grows from 1 to around 24. This indicates that the parallelization is effective, with each additional processor contributing significantly to reducing the computation time. The increase in speedup suggests that the MPI implementation is effectively distributing the workload among multiple processors, significantly improving performance over using a single processor. From around 24 processors onwards, the speedup factor fluctuates between approximately 10 and 14 but does not show a clear upward or downward trend. The fluctuation indicates that while additional processors continue to contribute to faster processing, the gains vary and may be influenced by factors such as inter-processor communication, in fact this fluctuation starts upon utilizing all 24 cores of the initial node, which may occur when transitioning from one THIN node to another. Moreover, it's possible to observe that the speedup shows a pattern like the logarithmic curve.



When examining the speedup achieved by utilizing a single node, it becomes evident that the trend follows a linear pattern.

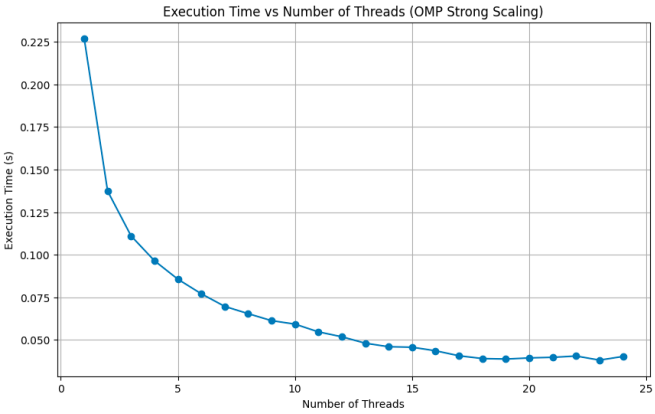


To analyze the complete dataset, a logarithmic model has been developed, as the linear model does not adequately capture the data's characteristics.
The equation of the line is given as

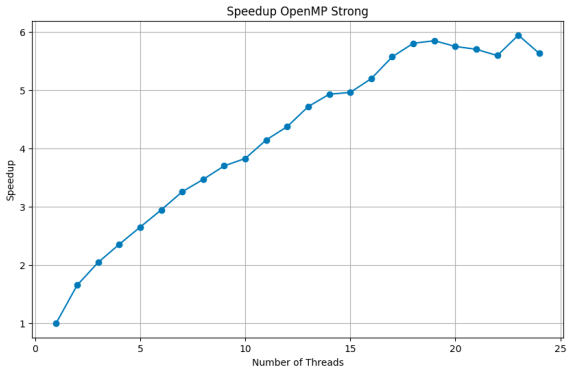
$$y = 0.70x + 0.04$$

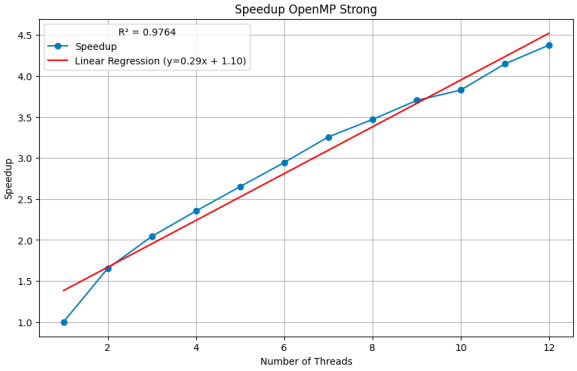
where y is the log of the speedup and x is the log of the number of processors. The value of $R^2 = 0.9637$ indicates a very high degree of correlation between the logarithms of the number of processors and speedup. This high value suggests that the model explains a substantial portion of the variability in speedup. The near-linear relationship on a log-log scale indicates strong scaling is effective up to the number of processors tested. A slope close to 1 would indicate perfect linear scaling, where doubling the number of processors halves the execution time. Here, the slope of approximately 0.70 suggests sub-linear scaling, but still significant improvement in speedup with increased processors.

OMP strong scaling



The plot demonstrates a sharp decline in execution time as the number of threads increases from 1 to approximately 10. This suggests that the addition of threads significantly speeds up the computation by distributing the workload more efficiently among the available CPU cores. Beyond 10 threads, the decrease in execution time begins to level off, indicating a plateau. As more threads are added beyond this point, the reduction in execution time becomes less pronounced, suggesting diminishing returns on adding more threads. The steep drop in execution time with the initial increase in the number of threads indicates that the parallelization of the task is highly effective at lower thread counts. This is characteristic of strong scaling where the system efficiently utilizes additional computational resources to reduce execution time.



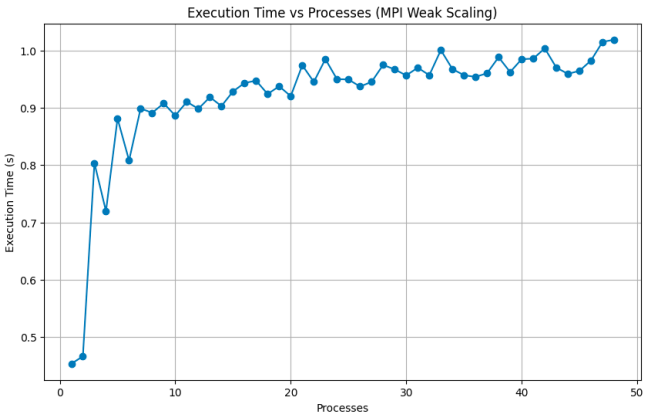


For the strong scaling analysis using OMP, a single THIN node was utilized. This analysis revealed a behavior like that observed with strong scaling using MPI, the plot shows a consistent increase in speedup as the number of threads increases from 1 to approximately 12. This suggests that adding threads improves performance significantly up to this point, likely due to the effective distribution of the computational load among multiple processors. The speedup observed with the initial 12 threads followed a linear trend, prompting the development of a corresponding linear model.

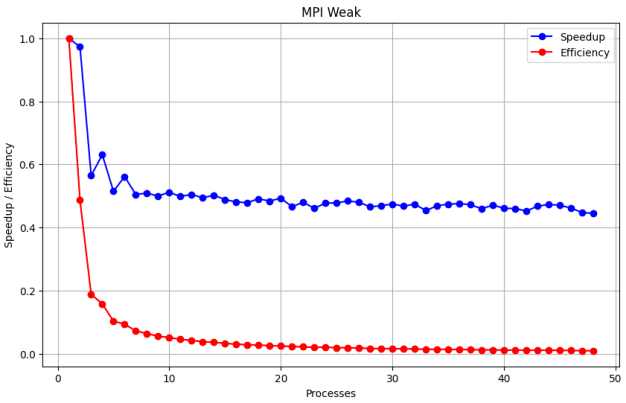
Weak Scaling

In this implementation of weak scaling using MPI processes, the dimensions of the image were systematically adjusted, with the number of rows and columns starting at 1000 by 1000 and incrementally expanding up to 48000 by 1000. Similarly, in the implementation of weak scaling utilizing OMP threads, the image size was also varied, with the number of rows and columns starting from 1000 by 1000 and extending up to 48000 by 1000.

MPI weak scaling



There is a sharp decrease in execution time as the number of processes increases from 1 to around 10. This suggests that initially, the additional processes significantly reduce the execution time, likely due to the efficient distribution of the workload. After the initial drop, the execution time stabilizes and fluctuates slightly around 0.8 to 0.9 seconds as more processes are added, from 10 up to 50. The overall trend from this point is relatively flat, with minor variations. The sharp decrease in execution time with the initial addition of processes indicates that the system efficiently handles parallel tasks by effectively utilizing more resources. The plot ideally illustrates weak scaling, where the goal is to maintain a constant execution time as the workload and the number of processes increase. The relatively flat trend beyond the initial drop suggests that the system manages to keep the execution time consistent despite the increased number of processes, demonstrating successful weak scaling.

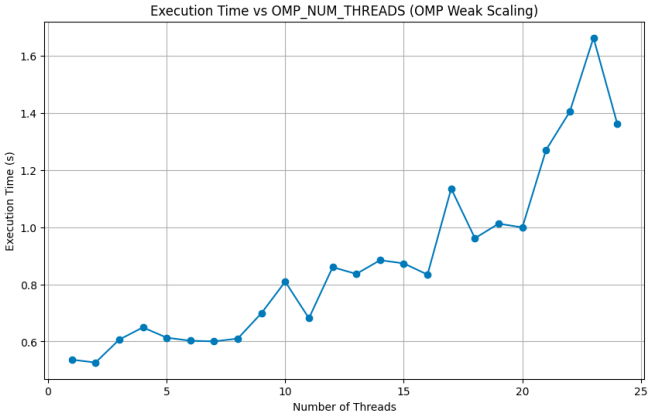


Speed up trend: there is a sharp initial decrease in speedup as the number of processes goes from 1 to around 5, indicating significant gains from parallelization. After the peak near 5 processes, the speedup gradually decreases and then stabilizes around 0.5. This indicates that while the speedup from adding more processes diminishes, it does reach a steady state where adding more processes does not significantly impact the speedup.

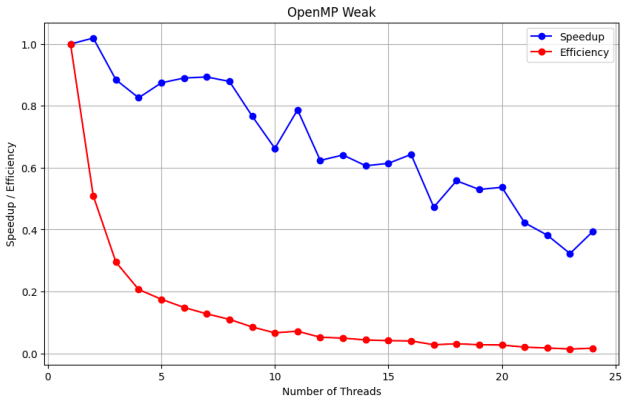
Efficiency trend: the efficiency shows a steep decline from the start as the number of processes increases. Starting close to 1 (optimal efficiency with a few processes), it drops significantly as more processes are added. The efficiency levels off at a low value around 0.1, suggesting that each additional processor contributes less effectively to the task as the number of processors increases.

According to Gustafson's law, the speedup should increase linearly with the number of processes and the problem size, nevertheless I can guess that the observed behavior is influenced by the fact that in weak scaling scenarios where the workload per processor remains constant, but the overhead associated with managing more processors (such as communication and synchronization costs) starts to outweigh the benefits of parallelization. The marked decline in efficiency suggests significant system overheads as the number of processors increases. These could include communication delays, load imbalance, and synchronization overhead.

OMP weak scaling



The execution time remains relatively low and stable as the number of threads increases from 1 to around 12 threads. This indicates good scalability and efficient utilization of resources within a single socket. A noticeable increase in execution time occurs at the point where the number of threads exceeds 12. This suggests a performance dip likely due to the transition from utilizing cores within a single socket to spreading the workload across multiple sockets. Inter-socket communication overhead may contribute to this increase in execution time. Post the transition, there is an oscillatory pattern with a general upward trend in execution time as the number of threads continues to increase. Despite some fluctuations, the overall execution time increases with the number of threads, which is typical in weak scaling scenarios due to increased communication and synchronization overheads as more threads are added.



Speed up trend: the speedup starts near 1 with one thread and quickly rises as more threads are added, reaching a peak around 5 threads. Beyond 5 threads, the speedup generally declines but with fluctuations. It shows a peak around 12 threads before gradually declining and then stabilizing slightly above 0.5 speedup as the thread count approaches 25.

Efficiency trend: efficiency decreases sharply from the start as more threads are added. The efficiency maintains a downward trend throughout, becoming more pronounced after 12 threads and approaching values near 0.1 by 25 threads, indicating that each additional thread contributes increasingly less to improving performance.