High-Performance Computing - Exercise 2

A Hybrid OpenMPI+OpenMP Parallelization of Mandelbrot Set:

:

Andela Cvetinovic [SM3800009]
Data Science and Artificial Intelligence [SM38-23]
Supervised professor: Luca Tornatore

Abstract

This project implements a program in C language that computes the Mandelbrot set in a parallel and distributed environment using OpenMP and MPI. Multiple tests are performed on the ORFEO cluster to evaluate the performance of the implementation, with variations in the number of resources used and the number of processes. Lastly, the results obtained are compared with the theoretical expectations.

1. Introduction

1.1. Mandelbrot Set

The Mandelbrot set is generated on the complex plane $\mathbb{C}$ by iterating the complex function $fc\,(z)$ whose form is $f_c\,(z) = z^2 + c$ for a complex point $c = x + i\,y$ and starting from the complex value $z = 0$ so to obtain the series $z0 = 0, z1 = f_c\,(0), z2 = f_c(z1), \ldots, f_c{}^n(z_{n-1})$. Denoted with M, this set is defined as the collection of complex numbers c for which the sequence defined by the iterative function remains bounded. It can be shown that if any element i of the sequence becomes greater than 2 in distance from the origin, the sequence will then diverge. Therefore, to determine if a point c is part of the Mandelbrot Set, the following condition is used: $z_n = f_c{}^n(0) < 2$ or $n > Imax$. Here, $Imax$ is a parameter that specifies the maximum number of iterations to perform before concluding that the point c belongs to the set M. Increasing $Imax$ improves the accuracy of the calculations but also increases the computational cost.

1.2. Hybrid approach

The Mandelbrot set is computed point by point and results are saved as a PGM image. Important to note is that the computation of each point is independent, which in turn allows parallelization and distribution among multiple processes to speed up the computation. MPI is used to distribute the computation across multiple nodes of the cluster, while OpenMP is used to parallelize the computation within each process.

Open Multi-Processing is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It provides a simple and flexible interface for developing parallel applications on multi-core processors. OpenMP is used to parallelize code by specifying the parallel regions, work-sharing constructs, and synchronization mechanisms, allowing efficient utilization of multiple cores within a single node to improve performance and reduce execution time.

Message Passing Interface is a standardized and portable message-passing system designed to function on a variety of parallel computing architectures. It is used primarily in distributed memory systems, where multiple processors, often on different nodes, need to communicate and coordinate their tasks. MPI provides a robust set of

functions for sending and receiving messages, synchronizing processes, and managing data distribution, making it essential for developing high-performance applications that run on large-scale clusters and supercomputers. In this project, MPI is used to distribute the computation of the Mandelbrot set across multiple processes.

2. Method

2.1. Node selection

All the tests were conducted on the ORFEO cluster, specifically on the THIN partition, which was chosen due to its simple architecture. This partition consists of 10 nodes, each equipped with 2 Intel Xeon Gold 6126 processors and 768 GB of RAM. The nodes are interconnected via a 100 Gbit HDR InfiniBand network, enabling high-speed communication with low latency. The benchmarks performed will occupy the most 4 THIN nodes (12x2 processors).

2.2. Hybrid implementation

In this hybrid implementation, the computation of the Mandelbrot set is distributed among multiple processes using MPI, and within each process, the computations are further parallelized using OpenMP.

2.2.1. MPI - for distributed Mandelbrot calculation

The program starts by setting up MPI to distribute the work across multiple processes. The total number of pixels to be computed is divided among the available processes (work distribution) where each process is responsible for the part of the image, i.e., calculates the number of pixels it is responsible for by *pixels_per_process = (num_pixels + np - 1) / np*. Then, an array of iterations is allocated to store the results for each pixel this process will compute.

2.2.2. OpenMP - for parallel processing

Within each process, OpenMP is used to further divide the work among multiple cores for faster computation. This is done through loop *#pragma omp parallel for schedule(dynamic, 512)* used to parallelize the computation of each pixel's value within the process. Each process calculates the Mandelbrot value for its assigned pixels using the compute_pixel function. The master process (rank 0) gathers the computed results from all processes, including itself, and combines them into the final image. The worker processes compute their assigned pixels and send their results to the master process. Once all data is gathered, the master process saves the combined results as an image file.

3. Scalability

The scalability aspects are tested by systematically varying the computational resources and analyzing the corresponding execution times, ensuring comprehensive scalability analysis for both strong and weak scaling scenarios. Then the experimental results are compared to theoretical expectations set by Amdahl's law.

3.1. Theoretical expectations

The theoretical speedup according to Amdahl's law states that "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used". In simple terms, it states that it does not matter how many processors there are or how much faster each processor may be; the maximum improvement in speed will always be limited by the most significant bottleneck in a system.

$$S_p = \frac{T_1}{T_p} = \frac{1}{(1-P) + \frac{P}{N}}$$

T1 = The time for completing the task using 1 processor
Tp = The time for completing the task using N processors
N = The number of processors being used
P = The fraction of the task that can be parallelized

Parallelizable fraction (P) is the most important part. It represents the portion of the task that can actually be split up and run in parallel. For example, if P=0.75 then 75% of the task can be done in parallel, and the remaining 25% must be done sequentially. Basically, if a large part of the task can be done in parallel (P is high), it can be expected significant speedup with more processors. Amdahl's Law is perfectly suited for strong scaling only (and not weak as well) because it considers a fixed workload and analyzes how much faster it can be completed with more processors. The law highlights the impact of parallelizable and non-parallelizable parts of the task when using increasing numbers of processors. The formula directly applies here as it compares the execution time of a fixed problem with one processor versus multiple processors.

3.2. Strong scaling

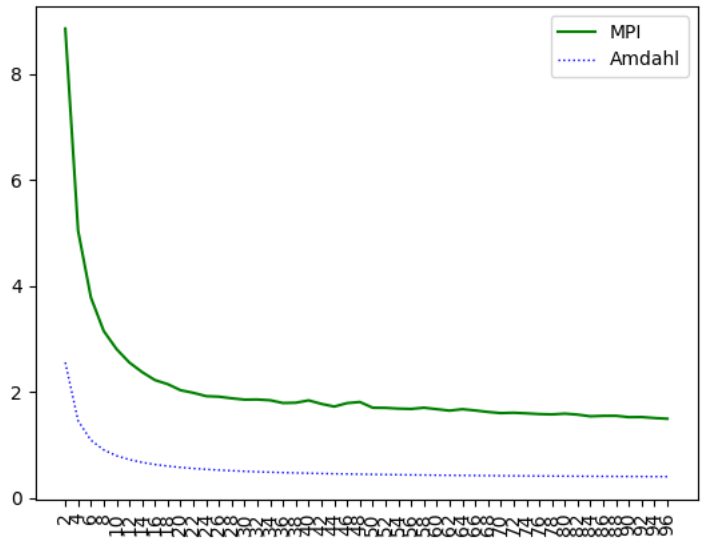
Strong scaling refers to the capability of a program to reduce the time required to solve a problem of fixed size by utilizing more computational resources. In this case, fixed parameters are image resolution, complex plane range, and a maximum number of iterations, while varied parameters are computational resources, i.e. a number of MPI processes. Image resolution, i.e., fixed problem size is set to 4096x4096, and max iterations to 1024. The number of MPI processes (np) is varied in a loop from 2 to 96 and for each value of np, the program is run with a fixed number of OpenMP threads set to 1 (OMP_NUM_THREADS=1). This ensures that the total number of computational resources (MPI processes) is increased while keeping the problem size (image resolution) constant.

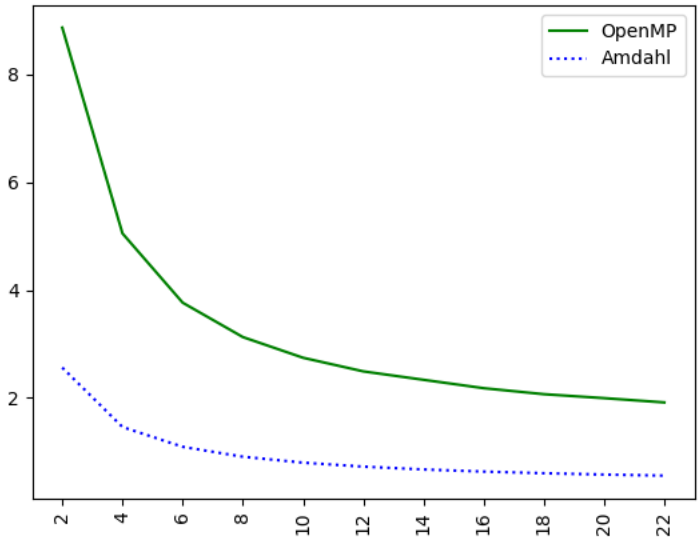
```
X_MIN=-2.0
X_MAX=2.0
Y_MIN=-2.0
Y_MAX=2.0
WIDTH=4096
HEIGHT=4096
MAX_ITERATIONS=1024
```

Fixed problem parameters

In parallel computing, the main interest lies in how performance improves as additional processors are added, so the benchmarking has been performed from the second process. So Amdahl formula is also adjusted to exclude the single-processor case and has an execution time of T2 using two processors. The parallelized fractions, P for both MPI and OpenMP are approximately 0.93. This means that 93% of the computation can be parallelized, while the remaining 7% must be executed sequentially. Here's what this indicates for the performance and scalability of the implementations:

The MPI implementation shows effective parallelization initially, but the benefits taper off as more processes are added. The discrepancy with Amdahl's Law highlights the impact of communication overhead and synchronization costs.



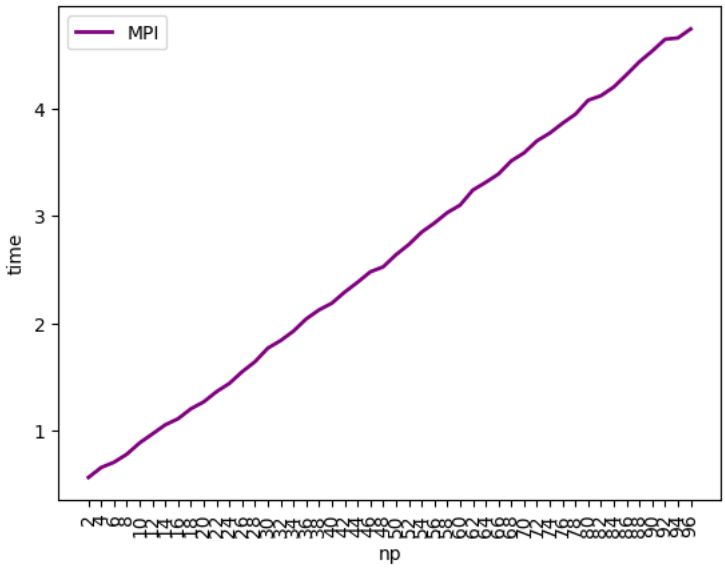


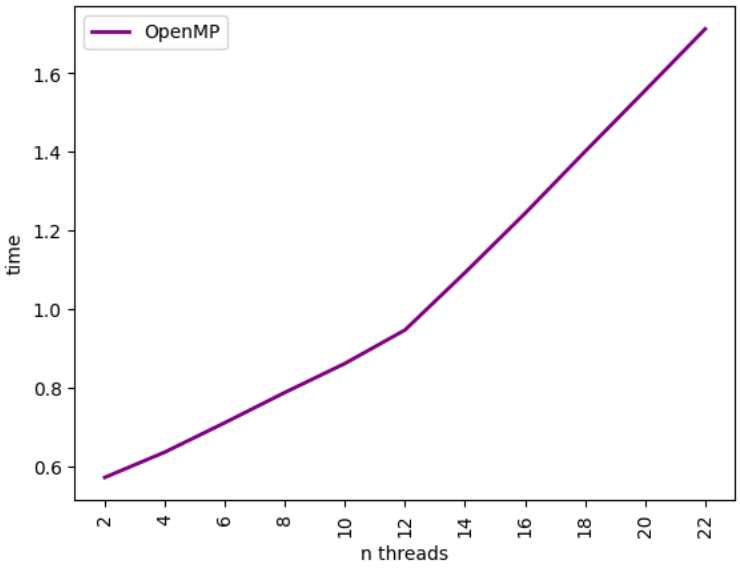
The OpenMP implementation also demonstrates good initial scaling but faces diminishing returns as thread count increases. The divergence from Amdahl's predictions indicates factors such as synchronization overhead and memory contention affecting performance.

3.3. Weak scaling

Weak scaling refers to the program's ability to maintain the same execution time while increasing the computational resources (Number of MPI processes and OpenMP threads) and proportionally enlarging the problem size. So, the same range of the complex plane and resolution parameters are used as in strong scaling. However, the problem size is effectively increased by dividing the work among more computational resources. The number of MPI processes (`np`) is varied in a loop from 2 to 96 but each process is responsible for a proportionate part of the problem. The number of OpenMP threads (`OMP_NUM_THREADS`) is varied from 2 to 24 to balance the workload within each process.

We have noticed that the execution time increases linearly instead of remaining constant. This suggests that the workload distribution among the workers needs to be adjusted to achieve a constant execution time (we should reduce the proportion of the data assigned to each process)





It is important to keep in mind the architecture (2 CPUs with 12 processors each) when analyzing this result. Up to 12 threads, the execution time increases gradually. Beyond 12 threads, threads are allocated to the second CPU (socket), which introduces additional overhead. This is likely due to inter-socket communication, increased memory access latency, and resource contention across the two CPUs.