Andrea Ivkovic (SM3800047)

High Performance Computing

For simplicity, all computations were carried out on the THIN partition of the Orfeo cluster. This partition is comprised of 10 nodes, each equipped with 2 Intel Xeon Gold 6126 CPUs and 768 GB of RAM. The nodes are interconnected via a 100 Gbps Infiniband network.

Exercise 1: Collective Operations

The initial exercise involved assessing the latency of two collective MPI operations: MPI_Bcast and MPI_Reduce. Factors varied in the analysis included the number of processes, message size, and distribution algorithm. The OSU Micro-Benchmarks, a suite of MPI benchmarks developed by Ohio State University, were utilized for these benchmarks. The maximum configuration used for this exercise involved 2 nodes (24*2 cores).

Broadcast operation

The broadcast operation is a type of collective communication where a message is transmitted from one process, known as the root process, to all other processes within the communicator. The duration of the broadcast operation is determined by the time it takes for the root process to dispatch the message and for all recipient processes to receive it. In our experiments, we utilized the default broadcasting algorithm, as well as alternative methods including basic linear, chain, and binary tree algorithms.

Basic linear algorithm

The basic linear algorithm is the most straightforward method for broadcasting a message. In this approach, the root process sequentially sends the message to each process in turn: first to the next process, then to the one after that, and so on, until all processes have received the message. The time complexity of this algorithm is O(n), where n represents the number of processes.

Chain algorithm

In the chain algorithm, the root process sends the message to the first process, which then forwards it to the second process, and this process continues until all processes have received the message. While the time complexity remains O(n), where n is the number of processes, this algorithm allows the root process to begin other tasks while the message is being propagated. Additionally, the chain algorithm improves upon the basic linear algorithm by reducing latency, as messages are transferred between neighboring processes, which are often located on the same node. This proximity typically results in lower communication overhead.

Binary tree algorithm

The binary tree algorithm provides a more efficient method for broadcasting a message. In this approach, the root process initially sends the message to two processes. Each of these processes then forwards the message to two additional processes, and this pattern continues until all processes have received the message. The time complexity of the binary tree algorithm is (O(\log(n))), where (n) is the number of

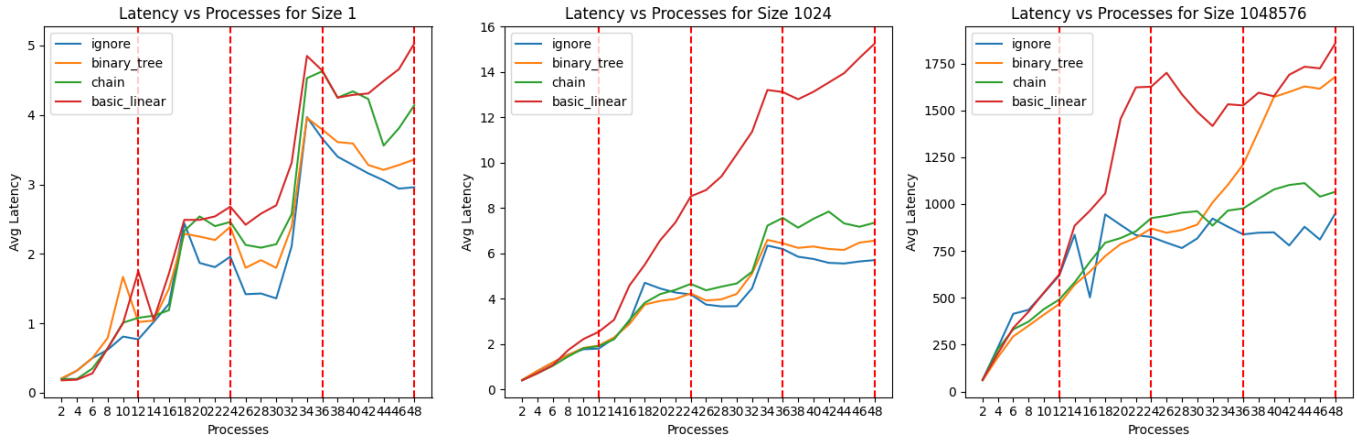
processes. This logarithmic complexity makes it significantly faster for large numbers of processes compared to linear-based algorithms.

Analysis

The parameters of the following benchmarks are:

- Number of processes: 2 -> 48
- Message sizes: 1 Byte -> 1 KB -> 1 MB
- Algorithms: default, basic linear, chain, binary tree
- Number of iterations: 10000
- Warm-up iterations: 1000

We present three plots showing the broadcast operation performance for three different message sizes: 1 Byte, 1 KB, and 1 MB. The x-axis represents the number of processes (increased every time by 2), while the y-axis indicates the time (microseconds) taken for the broadcast operation. Each plot compares the default algorithm with three alternative algorithms.



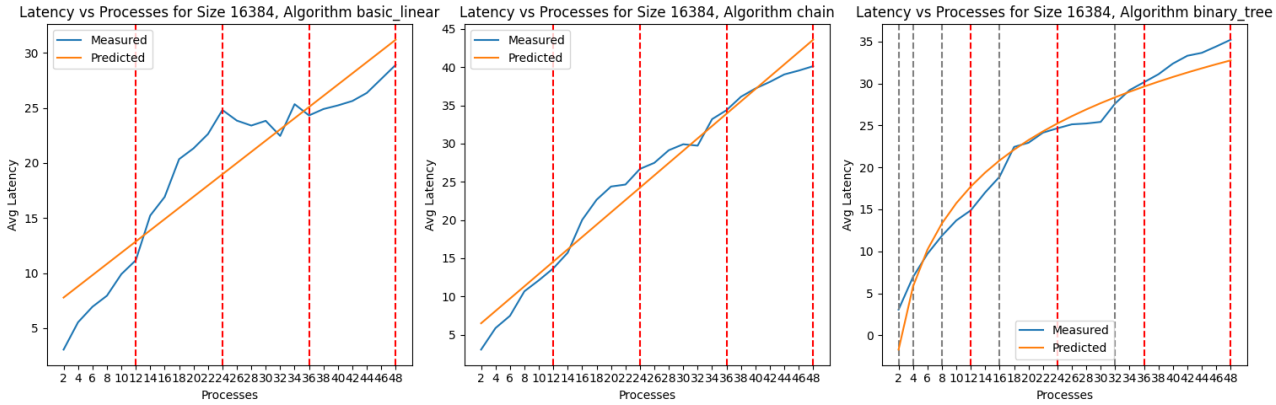
The basic linear algorithm consistently performs the worst across all message sizes.

In the first plot (1 Byte), we observe a rapid increase in latency when the number of processes exceeds 24. This increase is expected as it coincides with the processes no longer fitting on a single node, necessitating message transmission over the network. This network transmission introduces additional overhead, which is slower than intra-node communication. The overhead is more pronounced for smaller messages because a larger proportion of the total time is occupied by the overhead rather than by the actual message transmission. This effect is particularly evident in the first plot compared to the other two, where the smaller message size makes the overhead more significant.
Among the three algorithms chosen, the binary tree algorithm is generally the fastest. However, for the 1 MB message size, the chain algorithm becomes the fastest when there is a high number of processes (some processes are on different nodes). This is likely because the chain algorithm is more efficient for inter-node communication, reducing the number of hops needed to transmit the message. On the other hand, the binary tree algorithm may be slower in this scenario due to the increased number of transmissions between nodes, which can add additional latency.

We aimed to infer the latency of the broadcast operation by fitting models to the data (with a message size of 16 KB) using the following models:

- basic linear: (a + b * n)

- chain: (a + b * n)
- binary tree: (a + b * log(n))



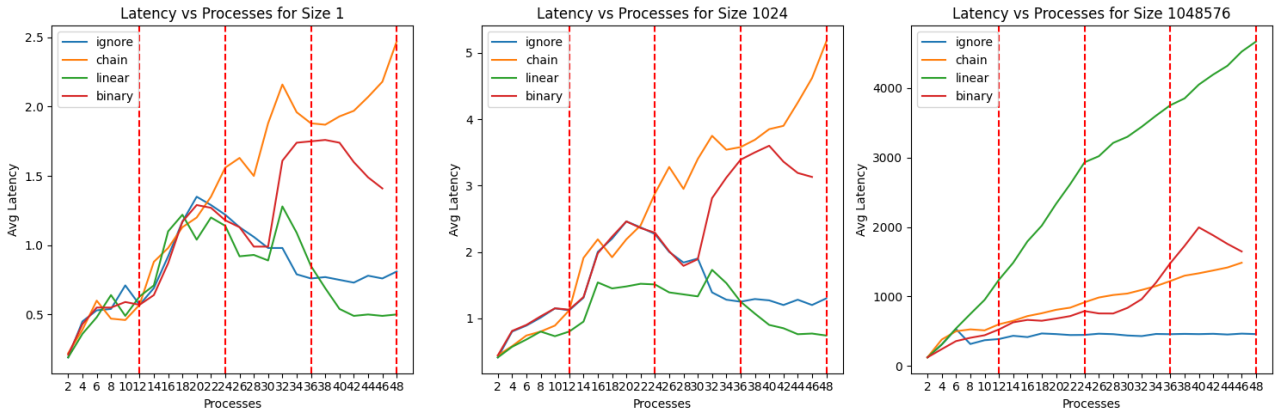
From the plots above, we observe the following:

- The basic linear algorithm shows a linear increase in latency, but only within four distinct subzones (corresponding to intrasocket, intersocket, and internode communication: 1, 12, 24, 36, 48). This indicates that a general linear model fitted to all the data is not suitable, and different models should be considered for these subzones.
- The chain algorithm displays a linear increase in latency across all data points. This can be explained by its efficiency in inter-node communication, reducing the number of hops needed to transmit the message, resulting in only one transmission between nodes.
- The binary tree algorithm appears to increase logarithmically

Reduce operation

The reduce operation is another type of collective communication in MPI (Message Passing Interface) where data from all processes is combined in a specified manner and the result is returned to a designated root process. The operation involves each process contributing data, which is then reduced (combined) using a predefined operation such as sum, max, min. Again, we utilized the default reduction algorithm, as well as alternative methods including linear, chain, and binary algorithms. The algorithms used for the reduce operation are similar to those used for the broadcast operation. The differences are:

- In the linear algorithm, the root process collects the data from each process in turn, performing the reduction operation as it goes.
- In the chain algorithm, for the reduce operation, each process receives data from its predecessor, performs the reduction operation, and passes the result to the next process in the chain.
- In the binary algorithm, for the reduce operation, data is combined in a binary tree pattern. Each process sends its data to its parent process, which performs the reduction operation and passes the result up the tree until the final reduced result reaches the root process.
  As before, we present three plots showing the reduce operation performance for three different message sizes: 1 Byte, 1 KB, and 1 MB.



Since we are applying a default aggregation operation, we can observe that for small messages, the linear algorithm performs the best. This is because the aggregation function is called only once, instead of multiple times with smaller chunks of data, as in the chain and binary tree algorithms. However, as the message size increases, the linear algorithm becomes the slowest. This shift is also due to the increased computation on the root process, which has to handle more data and perform more extensive aggregation operations compared to the other algorithms. The chain and binary tree algorithms distribute the computational load more evenly across processes, making them more efficient for larger message sizes.

Exercise 2: Mandelbrot set computation

The second exercise involves creating a program in C++ to compute the Mandelbrot set using hybrid approach of OpenMP and MPI for parallel and distributed computing. This program will be executed on at most 4 THIN nodes. The Mandelbrot set is a set of complex numbers that, when iterated through a specific mathematical function, remain bounded within a certain range. The set is defined by the equation (z_{n+1} = z_n^2 + c), where (z_n) is the complex number at iteration (n), and (c) is the complex number being tested. The Mandelbrot set is the set of all complex numbers (c) for which the sequence (z_n) remains bounded.

Code implementation

The program works using at least two processes: the master process (rank 0) and the worker process. The master is responsible for distributing the work to the workers and collecting the results. The worker processes compute the Mandelbrot set for a subset of complex numbers and return the results to the master. The master then combines the results from all workers to generate the final image of the Mandelbrot set.
This dynamic allocation of the job gives a more balanced workload distribution among the worker processes, ensuring that each process has a similar amount of work to do.
The workers ask to the master for a new continuous chunk of pixels to compute, and the master sends the next chunk to the worker that requested it. The master keeps track of what chunk of pixels each worker is computing. The computation of the Mandelbrot set is done by the workers using OpenMP to parallelize the computation of each pixel. Each worker process is responsible for computing a chunk of the image. The computation of each pixel is independent of the others, making it suitable for parallelization. The OpenMP directives are used to parallelize the computation of pixels of a chunk, allowing multiple threads to work on different pixels simultaneously. We provide a part of the code for better understanding:

```
void master_process(ApproxConfig& config, int num_workers) {
    int num_pixels = config.width * config.height;
    int begin_idx = 0;
```

```
    std::vector<short int> image(num_pixels, 0);
    std::vector<int> worker_begin_idx(num_workers, 0);
    while (!all_workers_done(worker_begin_idx, num_pixels)) {
        MPI_Status status;
        MPI_Probe(MPI_ANY_SOURCE, TAG_ASK_FOR_WORK, MPI_COMM_WORLD,
&status);
        int worker_rank = status.MPI_SOURCE;
        int computed_pxs;
        MPI_Get_count(&status, MPI_SHORT, &computed_pxs);
        if (computed_pxs != 0) { // something to receive
            recv_image_part(image, worker_begin_idx[worker_rank - 1],
computed_pxs, worker_rank);
        } else { // at the beginning no data is received
            MPI_Recv(NULL, 0, MPI_SHORT, worker_rank, TAG_ASK_FOR_WORK,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        MPI_Send(&begin_idx, 1, MPI_INT, worker_rank, TAG_BEGIN_IDX,
MPI_COMM_WORLD);
        worker_begin_idx[worker_rank - 1] = begin_idx;
        begin_idx += MAX_NUM_PXS_PER_WORKER;
    }
    save_pgm(image, config.width, config.height, "output.pgm",
config.max_iterations);
}
```

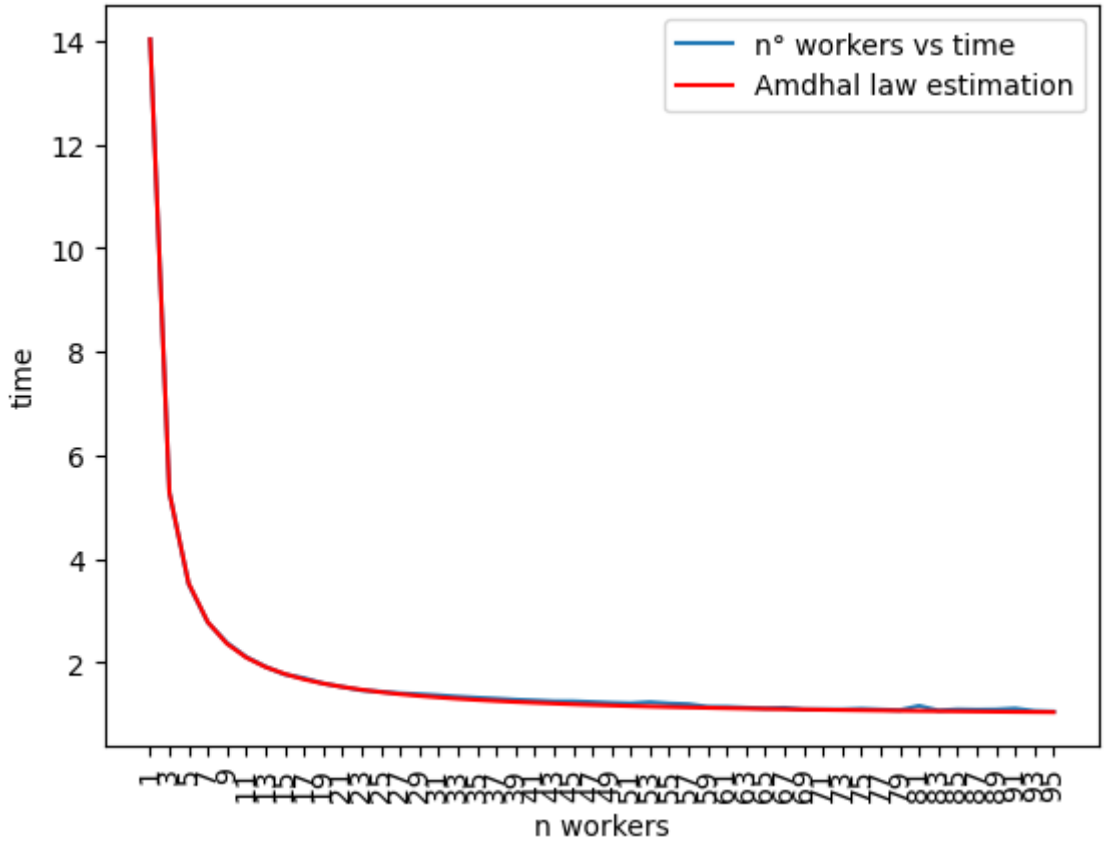
```
void worker_process(ApproxConfig& config) {
    int begin_idx;
    int num_pixels = config.width * config.height;
    MPI_Send(NULL, 0, MPI_SHORT, 0, TAG_ASK_FOR_WORK, MPI_COMM_WORLD); //
no data at the beginning

    while (true) {
        MPI_Recv(&begin_idx, 1, MPI_INT, 0, TAG_BEGIN_IDX, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int computed_pxs = num_pixels - begin_idx;
        if (computed_pxs <= 0) { // no more work
            break;
        }
        computed_pxs = std::min(computed_pxs, MAX_NUM_PXS_PER_WORKER);

        std::vector<short int> pixel_iterations(computed_pxs, 0);

        #pragma omp parallel for schedule(dynamic, 250)
        for (int i = 0; i < computed_pxs; i++) {
            int idx = i + begin_idx;
            Pixel p = {idx % config.width, idx / config.width};
            std::complex<double> pt = px_to_pt(p, config);
            pixel_iterations[i] = compute_point(pt, config.max_iterations);
        }
        MPI_Send(pixel_iterations.data(), computed_pxs, MPI_SHORT, 0,
```

```
    TAG_ASK_FOR_WORK, MPI_COMM_WORLD);
        }
    }
```

Analysis

MPI Strong scalability

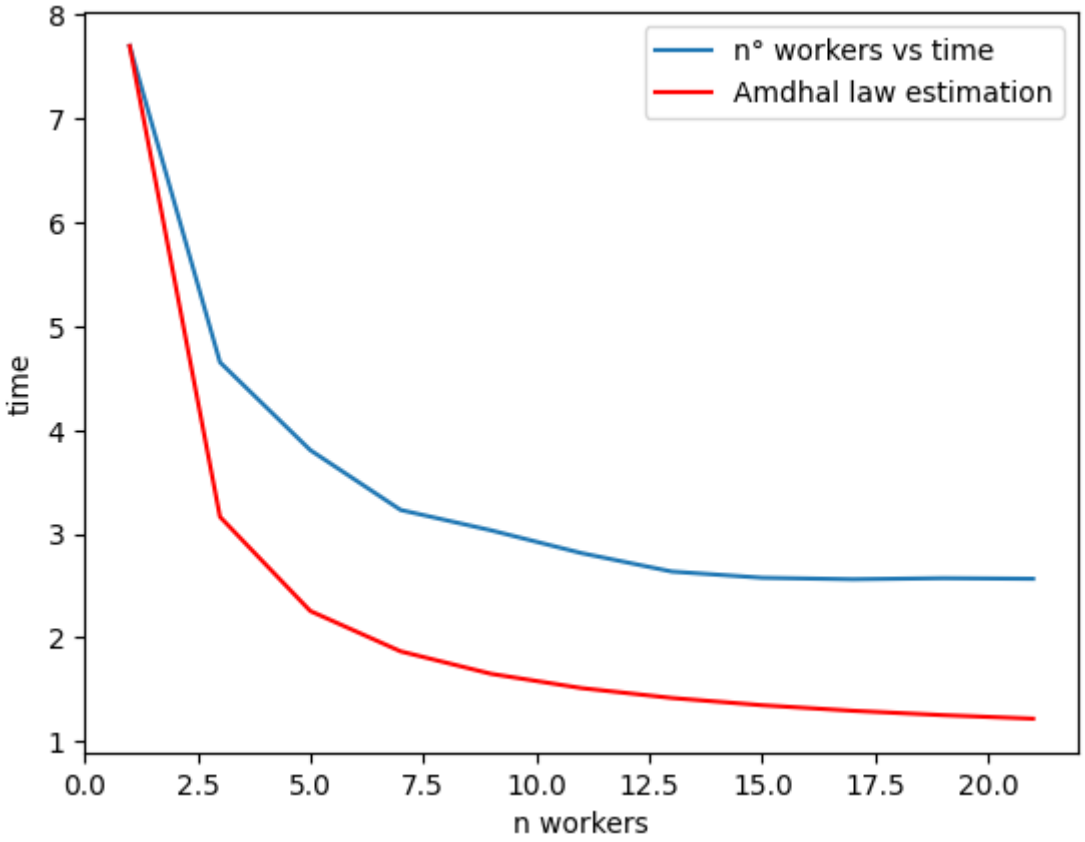
Strong scaling measures the reduction in execution time for a fixed problem size with increased computational resources. The test parameters included a 3840x3840 image resolution and 1000 iterations, with the number of MPI processes varied from 2 to 96 (1 to 95 workers). The number of OpenMP threads was fixed at 1.
We plot the strong scalability of the Mandelbrot set computation program using MPI. The x-axis represents the number of workers, while the y-axis indicates the execution time (seconds). An Amdahl's law curve is also included for comparison.



For plotting the Amhdal's law curve, we used the Amhdal's law formula: [S(N) = \frac{1}{(1 - P) + \frac{P}{N}}] where (S(N)) is the speedup, (P) is the parallel fraction of the program, and (N) is the number of workers. P=0.94 has been estimated using -3/2 * (1/S - 1), where S is the speedup with 3 workers.
From the plot we can see that the execution time follows the Amhdal's law curve.

OpenMP Strong scalability

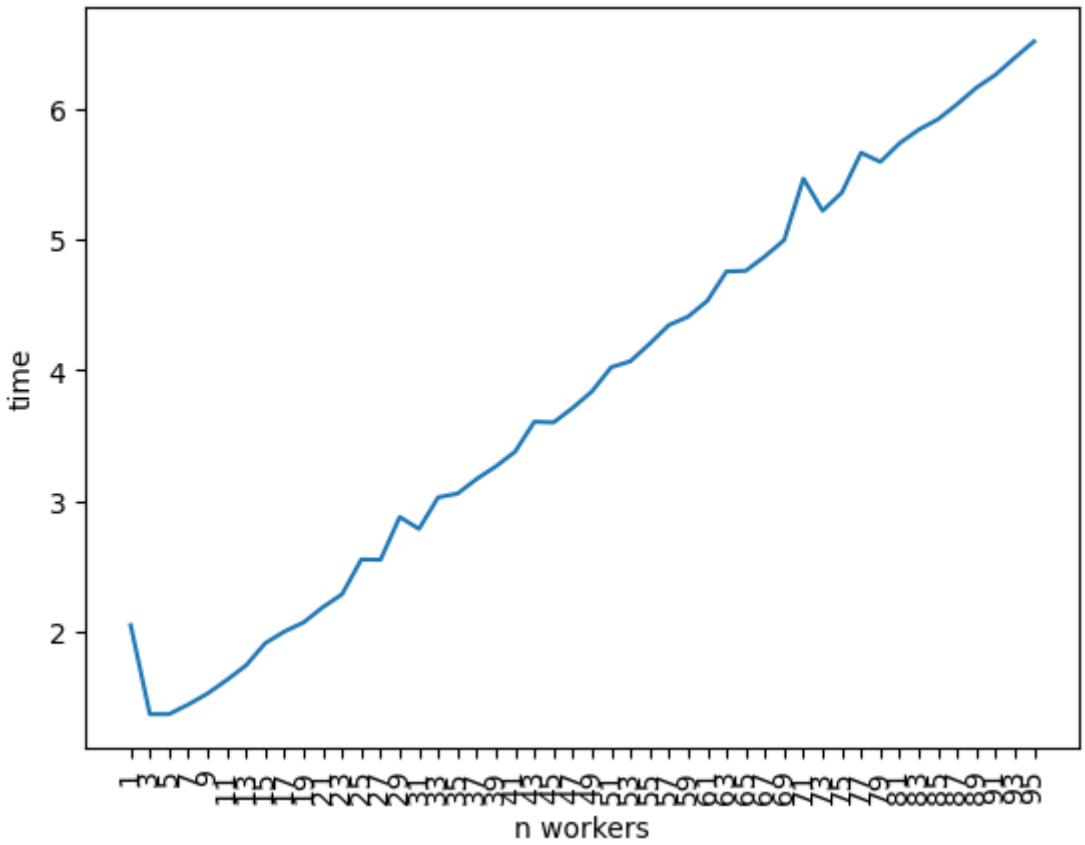
The test parameters included again a 3840x3840 image resolution and 1000 iterations, with the number of OpenMP threads varied from 1 to 24. The number of MPI processes was fixed at 2 (1 worker and 1 master). We plot again the execution time and the Amhdal's law curve (P estimated as 0.88).



In this case, we observe that the Amdahl's law curve is consistently below the actual execution time curve. This discrepancy suggests that there may be room for improving the OpenMP parallelization, potentially by experimenting with different scheduling strategies or chunk sizes.

MPI Weak scalability

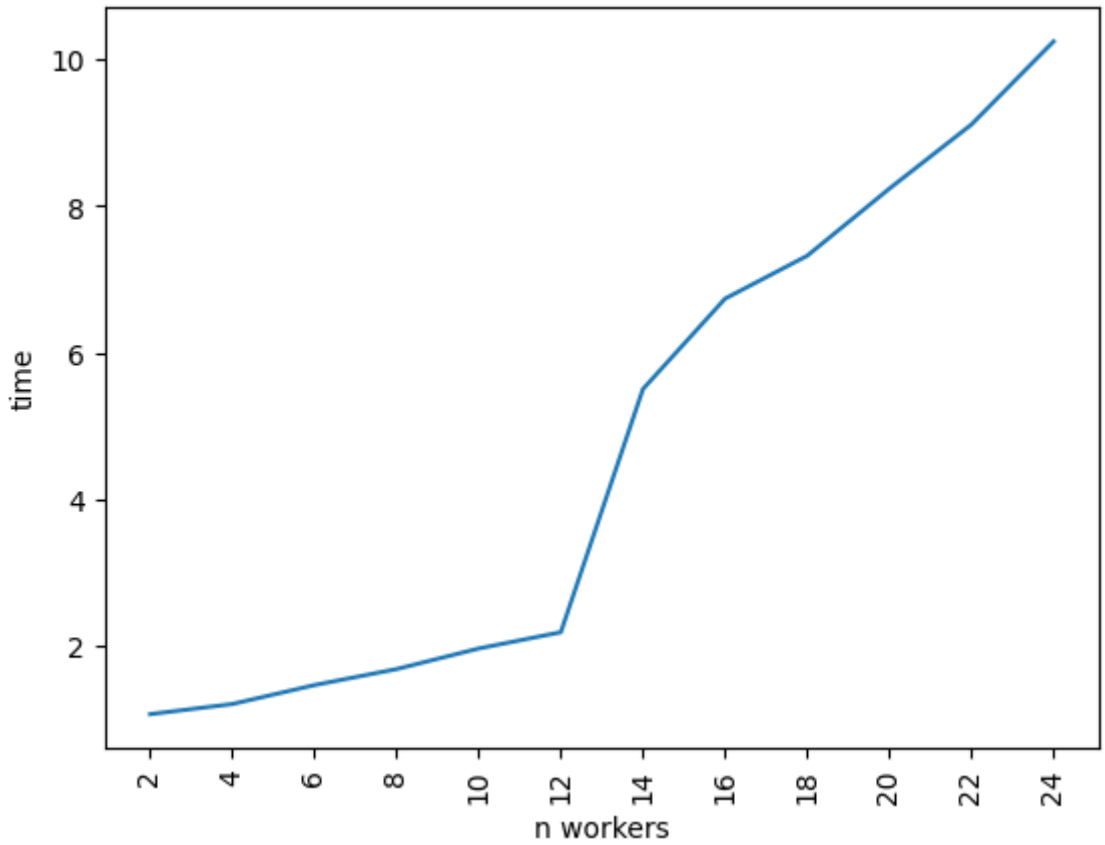
Weak scaling assesses the program's ability to maintain constant execution time while increasing both the problem size and computational resources. We proportionally increased the image height with the number of workers, keeping the image width constant at 3840 pixels. The number of iterations was set to 1000, and the number of OpenMP threads was fixed at 1. We plot the weak scalability of the Mandelbrot set computation program using MPI. The x-axis represents the number of workers, while the y-axis indicates the execution



time (seconds). We observe that the execution time increases linearly rather than remaining constant. This indicates that the workload distribution among the workers should be adjusted to achieve constant execution time.

OpenMP Weak scalability

Again we proportionally increase the size of the image with the number of OpenMP threads, keeping the number of MPI processes fixed at 2 (1 worker and 1 master). The image width was set to 3840 pixels, and the number of iterations was fixed at 1000. The number of OpenMP threads varied from 1 to 24. The number of MPI processes was fixed at 2 (1 worker and 1 master).



From the plot, we can observe that the execution time is linear up to 12 threads. Beyond this point, another linear trend emerges. This change in trend could be attributed to threads being distributed across both sockets.