Exercise 2C Hybrid MPI + OpenMP: the Mandelbrot Set

Fu Ziyang SM3800011 UniTS
MSc in Data Science & Artificial Intelligence
Foundations of Artificial Intelligence and Machine Learning

Introduction

The purpose of this assignment is to compute and visualize the Mandelbrot set over a specified region of the complex plane. By discretizing the plane into a grid of pixels, it is possible to iterate the function for each pixel and to determine whether it belongs to the Mandelbrot set based on a predefined condition.
This process allows us to generate an image where each pixel represents a point in the complex plane, colored according to whether it belongs to the Mandelbrot set or not.

This computation can be efficiently made in parallel, since each point can be computed independently of each other, and the exercise requires to employ a combination of **MPI (Message Passing Interface)** and **OpenMP (Open Multi-Processing)**.
This combination enables to ultimately determine:

- the **OMP scaling**: running the computation with a single MPI task and increasing the number of OMP threads;
- the **MPI scaling**: running the conmputation with a single OMP thread per MPI task and increasing the number of MPI tasks.

First Come First Serve

In order to distribute the workload among the different computational units, I employed a **First come First Served - Row Based Partition Scheme**.

Although it does not solve for the imbalance problem (inner points are computationally more demanding than the outer points), it mitigates it by assigning dynamically un-computed rows to the first available units, resulting in faster execution times compared with distributing chunks of rows in static and in a predetermined fashion.

The chosen partition scheme shares some similarity with schedulers in operating systems, with a master node sending work based on the availability.

For the **MPI scaling** the designed partition scheme is characterized by a **single master MPI process** and **MPI worker processes**.

The first is responsible for:

- the dynamic assignment of the indexes of rows to computed to the available workers;
- receiving the indexes and the computed rows and map the latter to their position in the global Mandelbrot matrix;
- sending the termination signal to each worker process, once the global matrix has been computed in its entirety.

The workers processes receive indexes, compute the assigned rows and lastly send the processed rows.
At row level the computation of the columns can be made parallel leveraging OpenMP.

Since different types of data are comunicated between workers and the master, it was necessary to define four different communication tags:

```
# define TAG_TASK_REQUEST 1 // signal the worker's availability and its rank;
# define TAG_TASK_DATA 2 // communicate the indexes and the termination signal;
# define TAG_TASK_ROW 3 // communicate from the index of the computed row;
# define TAG_MATRIX_ROW 4 // communicate the the computed row.
```

In the following section the code implementation and its execution flow are presented. The partition scheme for the **OMP scaling** will be explained in its section .

Code implementation

Designing the logical execution flow between the **master function** and the **workers function** to adhere to the flow of a First Come First Serve Partition scheme was the most challenging part of the assignment.
Following I will cover the main steps of the execution process.

Availability and Rows assignment

In order to implemented the designated partition scheme the workers should be able to signal their availability, with the master receiving such signal and "rewarding" the author with the index of the row to be computed.
This communication should go on until no more rows are left uncomputed.

```
mb_t *master (int nx, int ny, int size) {
    int next_row = 0;
    while (next_row < ny) {
        int available_p;
        MPI_Recv(&available_p, 1, MPI_INT, MPI_ANY_SOURCE, TAG_TASK_REQUEST,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&next_row, 1, MPI_INT, available_p, TAG_TASK_DATA,
        MPI_COMM_WORLD);
        next_row++;
    }
```

The master function will not exit the while loop until the variable `next_row` (representing the next uncomputed row) is smaller than the `ny` (the total number of rows composing the global matrix).

```
    for (int i = 1; i < size; i++) {
        int termination_signal = -1;
        MPI_Send(&termination_signal, 1, MPI_INT, i, TAG_TASK_DATA,
        MPI_COMM_WORLD);
    }
```

Once assigned all the rows the master can signal termination to the workers.

Now I will address how the worker function matches the workflow of the master:

```
void worker (int nx, int ny, double xL, double yL, double xR, double yR, int
Imax, int rank) {
    mb_t **big_array = NULL;
    int *big_index_arrays = (int *)malloc(sizeof(int))

    while (1) {
        MPI_Send(&rank, 1, MPI_INT, 0, TAG_TASK_REQUEST, MPI_COMM_WORLD);
        MPI_Recv(&row_index, 1, MPI_INT, 0, TAG_TASK_DATA, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);

        if (row_index == -1) { break;}
        mb_t *row = (mb_t *)malloc(nx * sizeof(mb_t));
        double dy = (yR - yL) / (double)(ny - 1);
        double y = yL + row_index * dy;
        compute_row(row, nx, xL, xR, y, Imax);
        size++;
```

1. the functions starts with signaling the worker's availability and then proceeds to receive the row index;
2. once received the index it checks if it matches the end signal and performs the Mandelbrot set computation.

```
        big_array = (mb_t **)realloc(big_array, size * sizeof(mb_t *));
        if (big_array == NULL) { exit(1);}

        big_index_arrays = (int *)realloc(big_index_arrays, size * sizeof(int));
        big_index_arrays[size -1] = row_index;
        big_array[size - 1] = row;
        received_rows++;
        }
```

So far the worker does not send back any of the computed rows, but it stores them in an array.
The reason is that up to this point of the workflow, the master has not opened any `MPI_Recv` for the rows data, so any `MPI_Send` by the workers will lead to a **deadlock situation**.
The workers will break from the while and start sending back the processed rows only once received the termination signal.

"Harvesting" the computed results

Once the master has successfully sent out the termination signal to all the workers, it means that the Mandelbrot matrix has been computed in its entirety. Now the master only needs to receive back the results:

```
mb_t *master (int nx, int ny, int size) {
    // ... code from before
    MPI_Status status;
    mb_t *Mandelbrot_1D = (mb_t *)malloc(nx * ny * sizeof(mb_t));

    for (int i = 0; i < ny; i++) {
        int row_index;
        MPI_Recv(&row_index, 1, MPI_INT, MPI_ANY_SOURCE, TAG_TASK_ROW,
        MPI_COMM_WORLD, &status);
        mb_t *row = (mb_t *)malloc(nx * sizeof(mb_t));
        MPI_Recv(row, nx, MPI_UNSIGNED_SHORT, status.MPI_SOURCE, TAG_MATRIX_ROW,
        MPI_COMM_WORLD, &status);
        memcpy(&Mandelbrot_1D[row_index * nx], row, nx * sizeof(mb_t));
    }
    return Mandelbrot_1D;
}
```

The function also takes care of mapping the received row back to its correct index. A **potential mismatch between the received index and the processed row** is avoided by specifying `status.MPI_SOURCE` as the accepted source in the second `MPI_Recv` , so that master is forced to receive the computed row from the same worker from which it has received the index.

```
void worker (int nx, int ny, double xL, double yL, double xR, double yR, int
Imax, int rank) {
    //... code from before
    for (int i = 0; i < received_rows; i++) {
        MPI_Send(&big_index_arrays[i], 1, MPI_INT, 0, TAG_TASK_ROW,
        MPI_COMM_WORLD);
        MPI_Send(big_array[i], nx, MPI_UNSIGNED_SHORT, 0, TAG_MATRIX_ROW,
        MPI_COMM_WORLD);
    }
    // freeing memory...
    return;
}
```

The worker function is simply responsible for sending the previously received indexes and the processed rows.

Each **MPI_Send** has its corresponding **MPI_Recv**, with corresponding tasks tag helping to identify the type of data involved in the communication.

Experiment setup

In this section I will briefly detail the inputs and the architecture with which MPI and OMP scaling benchmarks were run.

Inputs

n_x and **n_y** are set to 3096 and **I_max** is set to 65535 since short int was employed as size of integers.

Architecture

The tests were performed on ORFEO cluster using EPYC nodes, specifically two for the MPI scaling and one for OMP:

- **MPI scaling**: fixing the OMP threads constant to one using `export OMP_NUM_THREADS=1` , I mapped the processes to the processors' cores `--map-by core` and I measured the scaling by ranging the number of processes from 2 up to 256, increasing them by 2 at each iteration;
- **OMP scaling**: I set the number of MPI processes limited to one and mapped it to one of the two sockets, and lastly I set the **threads affinity** by placing them to the cores of the employed socket using `export OMP_PLACES=cores` .
  One socket disposes of 64 cores and since the Simultaneous MultiThreading is not active, the scaling has been conducted by ranging the hwthreads from 2 up to 64, increasing them by 2 at each iteration.

MPI scaling

In this section the results obtained with scaling the number of MPI processes are presented.

A few premises:

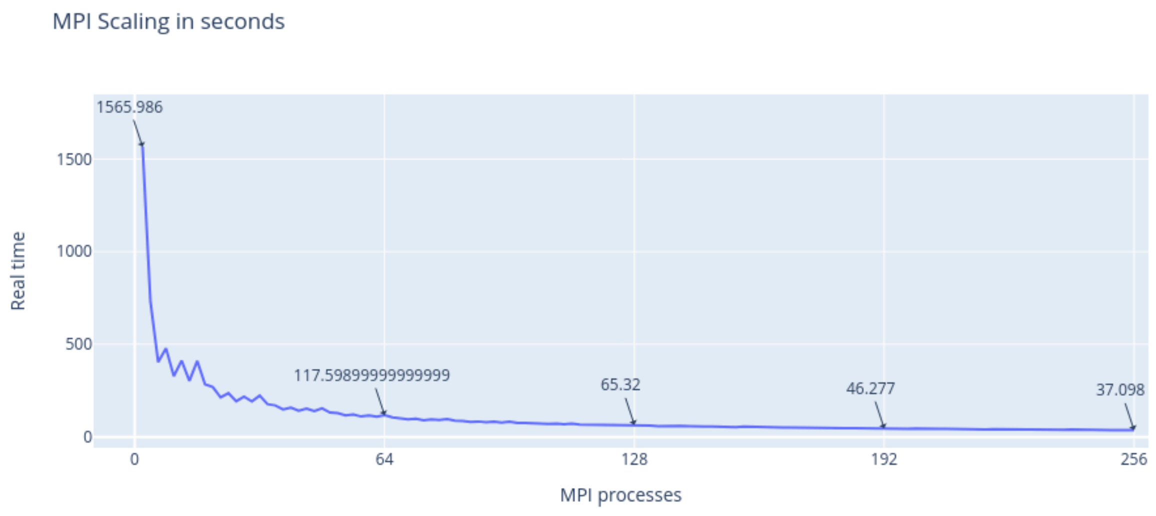
- keep in mind that with the designed approach the Master process does not take part in the pure computation of the Mandelbrot set, since it acts only as scheduler.
  So subtract one from the number of processes displayed in the following line chart as "active" processes;
- the time measurements were taken simply by means of:

```
export OMP_NUM_THREADS=1
# Loop through different inputs
for input in {2..256..2}
do
```

```
    time mpirun --map-by core -n $input ./executable 3096 3096 ...
done
```

so also the time spent reading the inputs and generating the image was accounted, but since the former are kept constant I assumed this chunk of the execution time to be constant over the iterations.

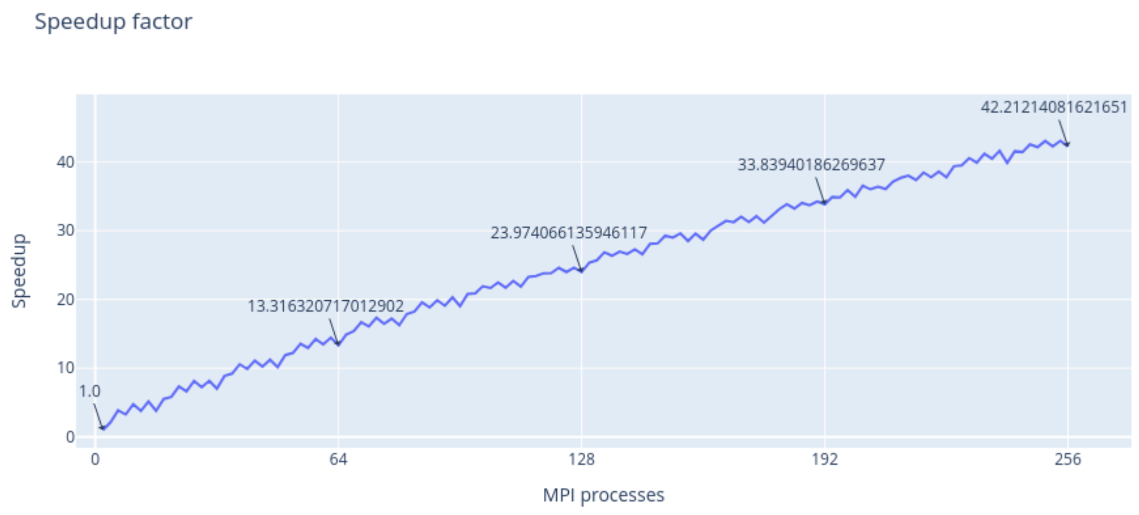
Following is presented the linechart of the measured MPI scaling:



The first highlighted number (from left to right) is the execution time spent by basically a **serial implementation** of the code: the master plus one worker process.
As one can expect the execution time decreases as the number of employed processes increases.

By simply observing the graph we can notice a first steep drop in the execution time passing from the serial implementation to one where 5 workers were employed; but as we increase the number of employed processes the line gets more and more flat, highlighting a progressive diminishing decrease in the execution time.

This pattern can be additionaly confirmed by the following linechart on the **speedup** given by increasing the number of workers:



I expect that such diminishing return is due to the **increasing communication overhead**: as more workers are added, more time is spent on communication between them and the master.

We can further stress this idea by computing a **theoretical speedup** using the Ahmdal's law:

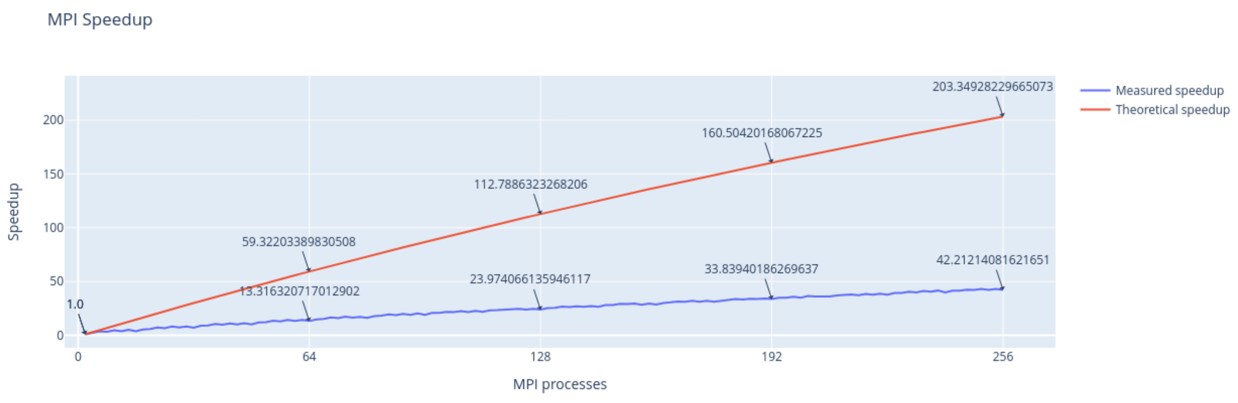
```
Speedup(s) = 1 / [(1-p) + p/s]
```

Where:

- *Speedup(s)* is the speedup achieved using *s* workers;
- *p* is the portion of the program that can be parallelized.

With various measurement *p* is estimated to be roughly 99%, that seems reasonable since other than computing the Mandelbrot matrix the program just needs to read the inputs and create write the image.

Having stated these assumptions we can take a look to the graph comparing the theoretical speedup with the measured one:



As the number of processes increases the gap between the two lines gets wider and wider, with the increasing communication overhead that progressively consumes bigger chunks of the benefit of adding workers.

OMP scaling

In this section I will first explain the approach with which I tackled the OMP scaling and lastly the measurement are presented.

Code

I implemented a function `parallel_mandelbrot` that gets called when the size of MPI communicator is one:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (size < 2) {
    if (size > 0) {
        Mandelbrot = parallel_mandelbrot(nx, ny, xL, yL, xR, yR, Imax);
        ...
    }
}
```

The function implements also a **First Come First Serve Row based partition scheme**:

```
#pragma omp parallel private (x, y, c, i) shared (next_row, Mandelbrot, nx, ny,
xL, dx, yL, dy, Imax)
{
    while (1) {
        #pragma omp atomic capture
        i = next_row++;
        if (i >= ny) {
            break;
        }
        y = yL + i * dy;
        for (int j = 0; j < nx; j++) {
            x = xL + j * dx;
            c = x + I * y;
            Mandelbrot[i * nx + j] = mandelbrot_func(0 * I, c, 0, Imax);
        }
    }
}
```

Inside the parallel region, each thread enters a while loop, that continues until all rows of the grid have been computed.
The flow of the function is the following:

1. the `#pragma omp atomic capture` ensures that the `i = next_row++` is performed atomically, so that race conditions and 2 simoultaneous increment of the `next_row` variable are prevented;
2. with the `capture` clause the thread reads atomically in `i` the original value of `next_row`;
3. once read `i` the function continues with computing the pixels of its designated row.

So once a thread enters its iteration of the while loop, it reads the `next_row` updated by the thread running before and in turn updates it, and ultimately computes the read index.

Scaling

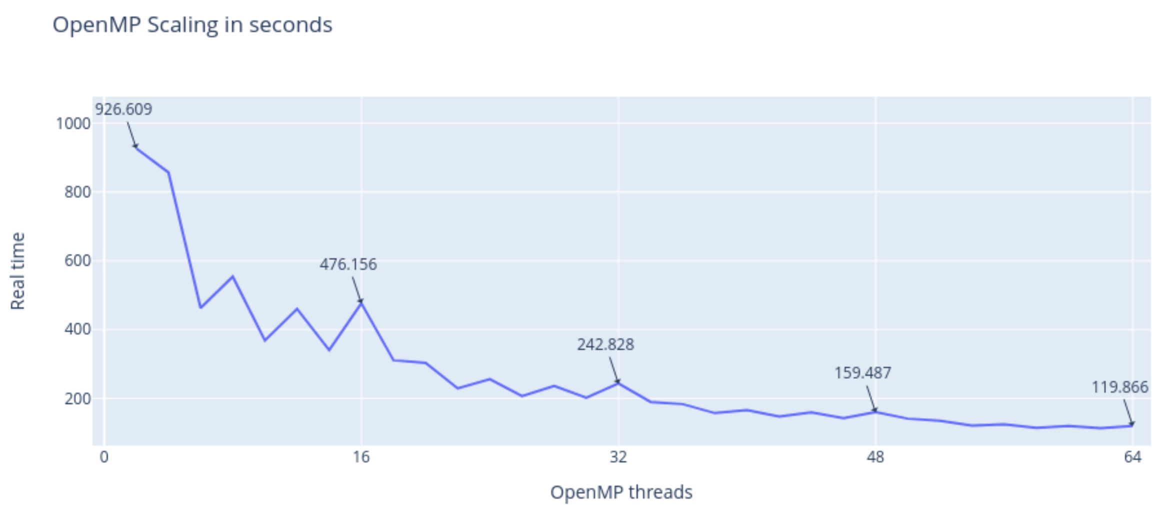
As highlighted before in the Experiment Setup section the only MPI process was mapped to a socket the EPYC node. Since the Simultaneous MultiThreading is not active the maximum number of **hwthread** we could make us of is 64.

The scaling was conducted by ranging the number of employed threads from 2 up to 64 in the following way:

```
for threads in {2..64..2}
do
```

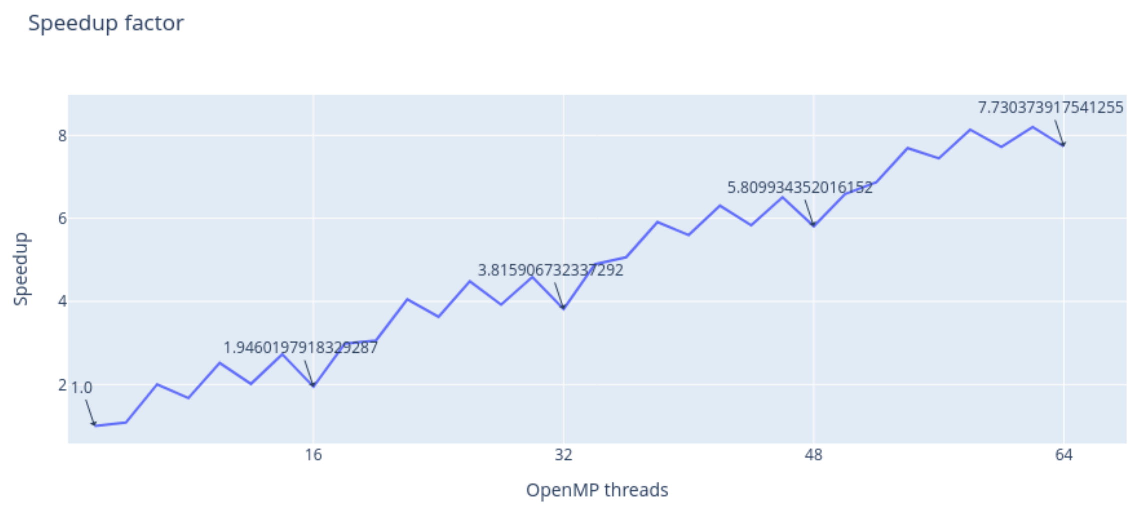
```
        # Set the number of OpenMP threads
        export OMP_NUM_THREADS=$threads
        time mpirun --map-by socket -n 1 ./executable 3096 3096 ...
    done
```

Following the results are presented:



Similarly to what has been reported before we can notice an initial steep decline passing from employing 2 threads to 6, and as the number of computational units gets increased we observe a progressive diminishing return on the additional threads.

The graph detailing the speedup can confirm so:



Conclusions

Within this assignment I explored a parallel approach for computing the Mandelbrot set, developing an hybrid MPI+OpenMP code.
The first challenge was to mitigate the intrisic inbalance probelms of the task: inner points are computationally more demanding than the outer points.

The designated solution has been the **First Come First Serve** row based partition scheme: each computational unit computes one row of the matrix at time and once it finishes, it will proceed with the next available one.
This technique was employed for both the **MPI and OMP scaling**, with the former having a master process workig as scheduler.

By measuring the results of both scalings, a common pattern between them has been observed: a **progressive diminishing return** in terms of faster execution times on the addition of new computational units.

I attributed this downside to the **increasing communication overhead** between master and workers in the MPI scenario, and possible ways to mitigate this problem could be:

1. using collective operations like **MPI_Bcast** instead of point-to-point communication;
2. using **non-blocking communication**, in the implementation I employed only blocking sends and blocking receives;
3. **reducing the communication**: instead of sending back the rows individually, I could simply make the workers to communicate the wrapper array used to store them during the computation step.

Regarding OMP scaling, the most straighforward conclusion I can draw is that this slowdown in performances can be attributed to **False Sharing**: it occurs when threads on different cores modify variables that reside on the same cache line.

In my case the global `Mandelbrot` array and the variable `next_row` are shared among all the threads: when one thread modifies them, other threads that have those cache lines in their caches need to invalidate their copies, leading to a cache miss the next time they need to access them.
Threads are forced to continously "load" the updated cache line from the main memory into their caches.
This process is slower compared than fetching data that is already in the cache.