Assignment 2c

Final Report submitted by Nooshin Behrooz

ORFEO username nbehrooz

Problem: parallel implementation of the Mandelbrot Set Algorithm

▪ Introduction of Quick-Sort Algorithm

The Mandelbrot set is a complex mathematical set defined by iterating a function $f_c(z) = z^2 + c$, where z is a complex number and c is a constant. The set consists of all complex numbers c for which the iteration does not diverge when starting from $z = 0$. Visually, it forms an infinitely complex fractal pattern, exhibiting self-similarity at different scales. The boundary of the set, known as the "Mandelbrot set boundary," is an area where infinitely detailed patterns emerge.

Here are the steps for Mandelbrot Algorithm:

Starting Point: We begin with a simple number, usually 0, as our starting point.

Iteration Rule: We have a rule, often written as $f_c(z) = z^2 + c$, where z is the current number, we're looking at, and c is a complex number that represents the point on the graph we're investigating.

Iteration Process: We repeatedly apply this rule to our starting number. Each time we apply the rule, we get a new number. We then take this new number and plug it back into the rule to get another number, and so on. We keep doing this repeatedly, creating a sequence of numbers.

Checking Boundedness: At each step in the sequence, we check if the number is staying close to 0 or if it's getting big. If it stays close to 00, we continue to the next step. If it starts getting big, we stop and say that the point we're investigating isn't part of the Mandelbrot set.

By following these rules and checking lots of points on our graph paper, we can systematically explore the complex plane and identify which points belong to the Mandelbrot set and which ones do not.

▪ Implementation of OMP Mandelbrot Set Algorithm

We have used these definitions in our code:

n_x, n_y: represent the number of pixels along the horizontal and vertical axes, respectively, when generating the Mandelbrot set image. We considered image with 256×256 pixels.

The circular region (**c_L, c_R**) is centered on **(- 0.75,0)** with radius 2, which is a commonly used center point due to the richness of detail it reveals within a manageable computational space.

x_L, y_L, x_R, y_R: represent the left, bottom, right and top boundaries, respectively. In our model according to circular center, we set **x_L=-2.75, y_L=-2, x_R=1.25 and y_R=2**.

(x_L+iy_L), (x_R+iy_L), (x_L+iy_R), (x_R+iy_R): state the bottom_left, bottom right, top left and top right corners.

I_max: is a parameter that sets the maximum number of iterations used to determine whether a point belongs to the Mandelbrot set. We have considered **I_max = 255**.

We will receive all these arguments **(n_x, n_y, x_L, y_L, x_R, y_R, I_max)** through the command-line.

Here we will discuss very briefly different parts of the code's structure and parallelization strategy. The result for Strong and Weak scaling performance will be investigated in the next part.

1. Header Files and Definitions:

The code includes standard header files stdio.h, stdlib.h, math.h, and omp.h and it defines MAX_ITERATIONS_INT as 255.

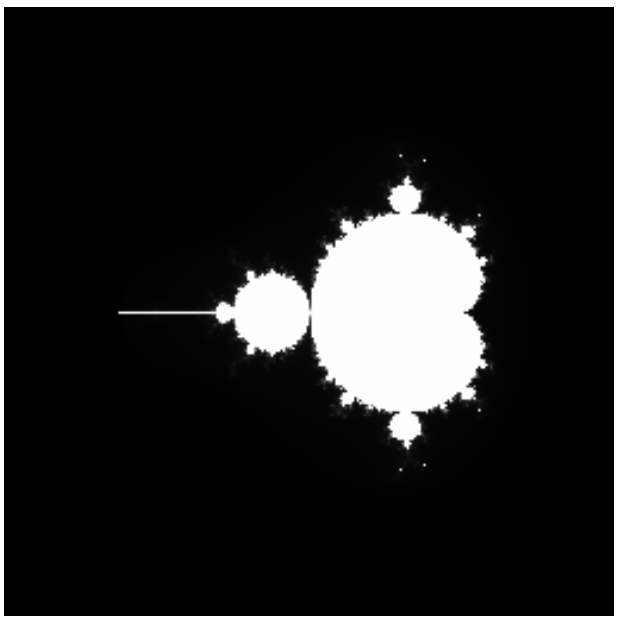
2. Definition of Function

- **Mandelbrot Function:** determines whether a point in the complex plane belongs to the Mandelbrot set. It takes the real and imaginary parts of the point and the maximum number of iterations as arguments and iterates the Mandelbrot formula $f_c(z) = z^2 + c$ until either the maximum number of iterations is reached or the magnitude of $|f_c z|$ exceeds 2.
- **Write_PGM_Image Function**: writes pixel data to a PGM image file. It takes the filename, pixel data array, image width and height, and the maximum pixel value as arguments and writes the PGM header and pixel data in binary format to the file.

3. Main Function:

- It **checks** if the correct number of **command-line arguments** (n_x, n_y, x_L, y_L, x_R, y_R, I_max) are provided. If not, it prints the correct usage and exits, parses the command-line arguments into **variables** and calculates the delta values for the complex plane coordinates based on the image dimensions and boundaries.
- **omp_get_wtime():** measures execution time.
- **OpenMP directive:** is a **Parallelization Strategy** for this code. The program utilizes OpenMP directives to parallelize the generation of the Mandelbrot set. Specifically, it parallelizes the nested loops responsible for iterating over each pixel of the image. The **#pragma omp parallel for** directive distributes iterations of the outer loop (n_y) among multiple threads, while the collapse (2) clause collapses the loop nest into a single loop, optimizing thread management.
- **write_pgm_image function:** It writes the Mandelbrot image to a PGM file.

The output Mandelbrot Image:



OMP Mandelbrot Set Algorithm: snippet

```
// Definition of Function
int Mandelbrot (double real, double imag, int max_iter) {...}
void write_pgm_image (const char *filename, int *pixel_data, int width, int height, int max_value) {...}
// implementation of functions
int main () {
    int n_x = atoi(argv[1]);
    int n_y = atoi(argv[2]);
    double x_L = atof(argv[3]);
    double y_L = atof(argv[4]);
    double x_R = atof(argv[5]);
    double y_R = atof(argv[6]);
    int I_max = atoi(argv[7]);

    double delta_x = (x_R - x_L) / n_x;
    double delta_y = (y_R - y_L) / n_y;

    double start_time = omp_get_wtime();

    // Parallelization Strategy: Generate Mandelbrot set with OMP directive
    #pragma omp parallel for schedule (static, 1) collapse (2)
    for (int j = 0; j < n_y; j++) {
        for (int i = 0; i < n_x; i++) {
            double real = x_L + i * delta_x;
            double imag = y_L + j * delta_y;
            int index = j * n_x + i;
            iteration_counts[index] = mandelbrot(real, imag, I_max);}}
    write_pgm_image("mandelbrotints_image.pgm", iteration_counts, n_x, n_y,
MAX_ITERATIONS_INT);
```

- # Performance Results (static ompi.mandelbrotints.c)

a. Strong scalability

Definition: According to **Amdahl's law** states that, for a fixed problem, the upper limit of speedup is determined by the serial fraction of the code (s).

$$S(p) = \frac{T_{(serial)}}{T_{(parallel)}} = \frac{T_1}{s+\frac{p}{N}}$$

$$\lim_{N\to\infty} S(p) = \frac{1}{s}$$

Therefore, **Strong scalability** refers to how the **execution time** of a parallel program changes as the **number of nodes** (processors or threads) is **increased** for a **fixed total problem size**. In an ideal strong scaling scenario, doubling the number of processors should ideally halve the execution time, resulting in a speedup that is close to doubling.

Interpretation of results

The strong scaling is tested by running our code with different numbers of OMP Threads of epyc007 node of ORFEO HPC center, while keeping the dimension of the Mnadelbrot_image constant as 256×256. The Execution time for generating the Mandelbrot set is monitored for each calculation. The results are shown in Table 1,2 and Figure 1,2,3. In the Figures we also show the fitted curve based on Amdahl's law.

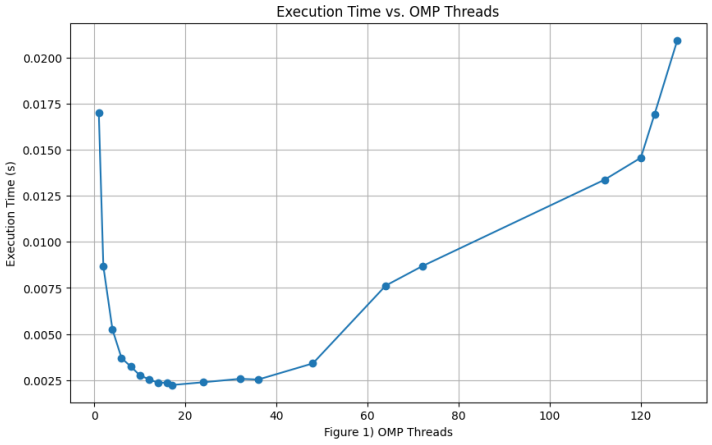


Figure 1) OMP Threads

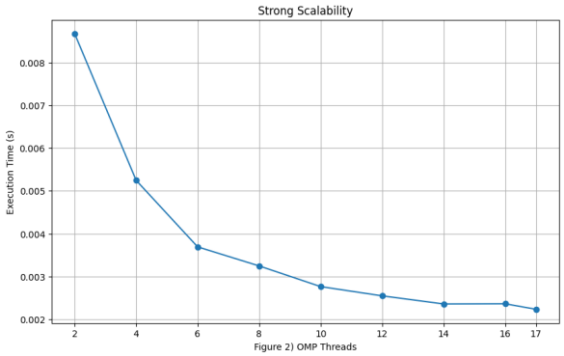


Figure 2) OMP Threads

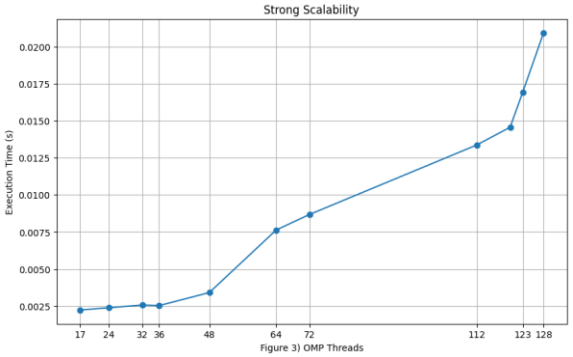


Figure 3) OMP Threads

To better analyze the results, we divided the data from figure 1 into two intervals: [2, 4, 6, 8, 10, 12, 14, 16, 17] in figure 2 and table 1, and [17, 24, 32, 36, 48, 64, 72, 112, 120, 128] in figure 3 and table 2. We observed that the execution time decreased consistently with strong scalability up to 17 nodes.

However, between 18 and 36 nodes, the execution time remained relatively constant.

Beyond 36 nodes, the execution time increased as the number of nodes increased. Interestingly, with 72 nodes, we achieved the same execution time as with only 2 nodes. Furthermore, with 123 nodes, we achieved the execution time equivalent to running with just 1 node. However, beyond this point, up to 128 available nodes, the performance got even worse than the serial code.

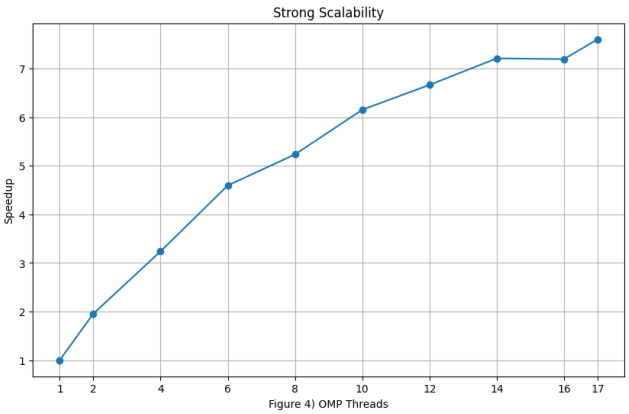
Table 1

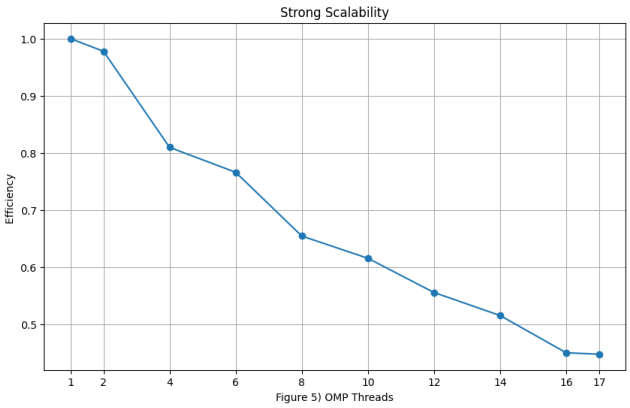
Dimension of image n_x × n_y	#node	Execution Time (s)	$S(p) = \frac{T_1}{T_p}$	$E(p) = \frac{S_p}{N}$
256 × 256	1	**0.016990**	1.000000	1.000000
256 × 256	**2**	**0.008672**	**1.959178**	0.977817
256 × 256	4	0.005245	**3.239275**	0.809474
256 × 256	6	0.003691	4.603088	0.765773
256 × 256	8	0.003246	**5.234134**	0.654101
256 × 256	10	0.002763	6.149113	0.615144
256 × 256	12	0.002547	6.670592	0.555152
256 × 256	14	0.002358	7.205258	0.514851
256 × 256	16	0.002362	**7.193056**	0.449561
256 × 256	**17**	**0.002234 Min**	7.605192	0.447079

Table 2

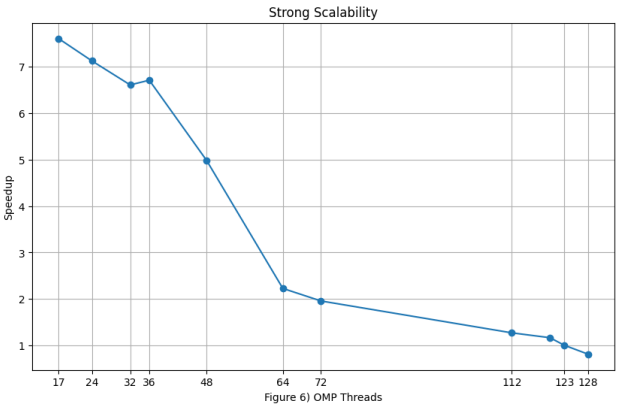
Dimension of image n_x × n_y	#node	Execution Time (s)	$S(p) = \frac{T_1}{T_p}$	$E(p) = \frac{S_p}{N}$
256 × 256	**17**	**0.002234 Min**	7.605192	0.447079
256 × 256	24	0.002385	7.123689	0.296820
256 × 256	32	0.002571	6.608323	0.206510
256 × 256	**36**	**0.002532**	6.710110	0.186391
256 × 256	48	0.003408	4.985328	0.103861
256 × 256	64	0.007629	2.227028	0.034797
256 × 256	**72**	**0.008681**	1.957147	0.027182
256 × 256	112	0.013367	1.271040	0.011348
256 × 256	120	0.014565	1.166495	0.009720
256 × 256	**123**	**0.016920**	1.004137	0.008163
256 × 256	**128**	**0.020937Max**	0.811482	0.006339

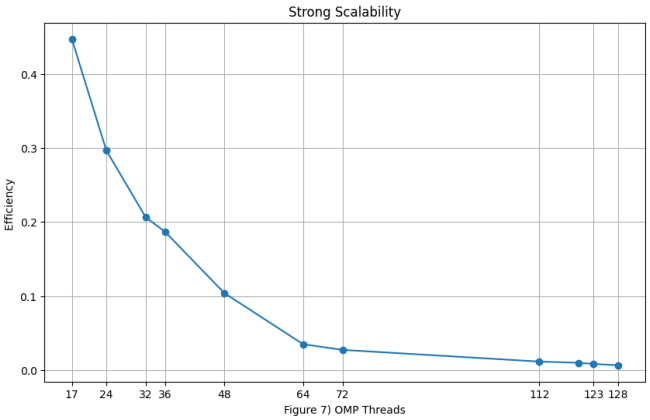
Moreover, calculation of execution time in table 1 and 2, we have measured the speedup and efficiency. As we can see in table 1, doubling the number of processors (from 2 to 4, 4 to 8, and 8 to 16) resulted in halving the execution time and approximately doubling the speedup.





According to Figures 4 and 5, the speedup trend increases from 1 to 17 nodes, with 17 threads leading to maximum speedup (7.605192). However, the efficiency decreases in that interval, with maximum efficiency in parallel computing belonging to 2 threads (0.977817).





It is illustrated in figures 6 and 7 that both speedup and efficiency decline as the number of threads rises, which is not good for strong scaling and results in waste of resources.

b. Weak scalability

Definition: According to **Gustafsson's law** in parallel problems, if the **total problem size increases** as the **number of nodes increase**, the **sequential proportion** of the computations will **decrease**. With Gustafson's law the scaled speedup increases linearly with respect to the number of processors (with a slope smaller than one), and there is no upper limit for the scaled speedup.

So, **for scaled speedup** we have the following formula:

$S(p) = s + Np$ (s: time of serial part, N: number of nodes, p: time of parallel part)

$T(p) = s + \frac{Np}{N} = S + P = \ CONSTANT$

Execution time remains constant when increasing parallel workload.

Interpretation of results

We measured execution time for weak scaling by running the code with different numbers of threads and with increasing the number of pixels along the horizontal axis of the image (n_x), which effectively increases the width of the image. The height of image (n_y) pixel was kept constant. The results of the weak scaling test are shown in Table 3, 4 and Figure 8.

Table 3

Dimension of image n_x × n_y	#node	Execution_Time (s)	$S(p) = s + Np$	$E(p) = \frac{S_p}{N}$
1 × 256 × 256	1	**0.016990**	1.00	1.000000
2 × 256 × 256	**2**	**0.017246**	1.25	0.625000
4 × 256 × 256	4	0.018877	1.75	0.437500
6 × 256 × 256	6	0.020559	2.25	0.375000
8 × 256 × 256	8	0.022960	2.75	0.343750
10 × 256 × 256	10	0.021401	3.25	0.325000
12 × 256 × 256	12	0.023201	3.75	0.312500
14 × 256 × 256	14	0.023294	4.25	0.303571
16 × 256 × 256	16	0.024469	4.75	0.296875
20 × 256 × 256	20	0.024002	5.75	0.287500
22 × 256 × 256	22	0.025158	6.25	0.284090
24 × 256 × 256	24	0.029063	6.75	0.281250
32 × 256 × 256	**32**	**0.027704**	8.75	0.273437

Table 4

Dimension of image n_x × n_y	#node	Execution Time (s)	$S(p) = \frac{T_1}{T_p}$	$E(p) = \frac{S_p}{N}$
48 × 256 × 256	48	0.289297	12.75	0.265625
64 × 256 × 256	64	0.368518	16.75	0.261718
72 × 256 × 256	72	0.561798	18.75	0.260416
86 × 256 × 256	86	0.561798	22.25	0.258720
100 × 256 × 256	100	0.760260	25.75	0.257500
112 × 256 × 256	112	0.953074	28.75	0.256696
120 × 256 × 256	120	1.084503	30.75	0.256250
128 × 256 × 256	128	1.234233	32.75	0.255859

According to the information provided in Table 3 and 4, the total execution time remains approximately constant for nodes in the range of 2 to 32. However, beyond 32 nodes, increasing the number of threads leads to a rise in execution time instead of remaining constant, which violates Gustafson's Law. Therefore, we will not consider them for further investigation."

Figure 8 is a graphical representation of data combination of table 3 and 4. To plot speedup in Figure 9, the serial fraction is assumed to remain constant, and the parallel part is assumed to be sped up in proportion to the number of threads.

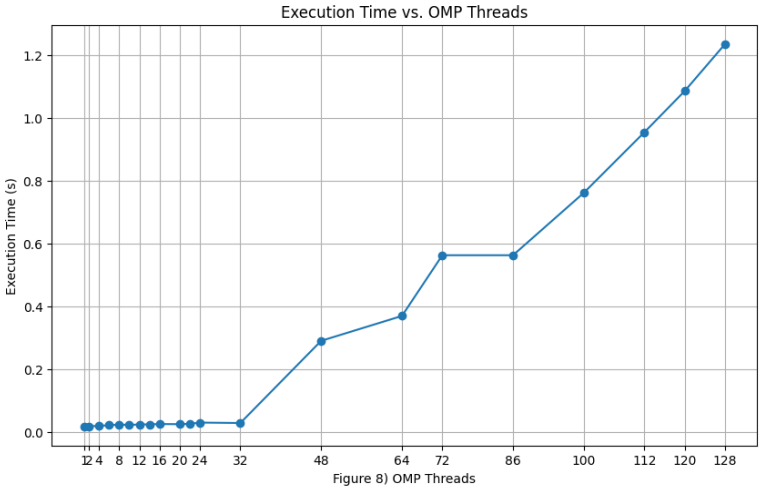


Figure 8) OMP Threads

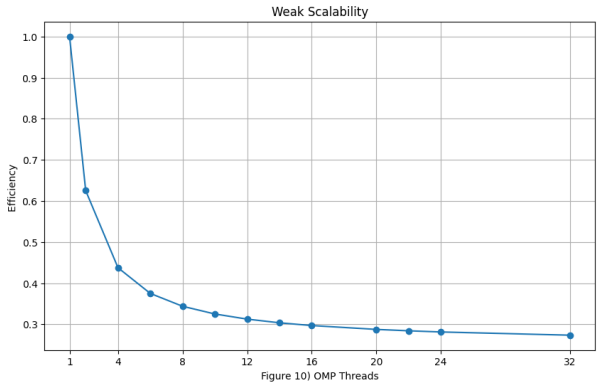


Figure 10) OMP Threads

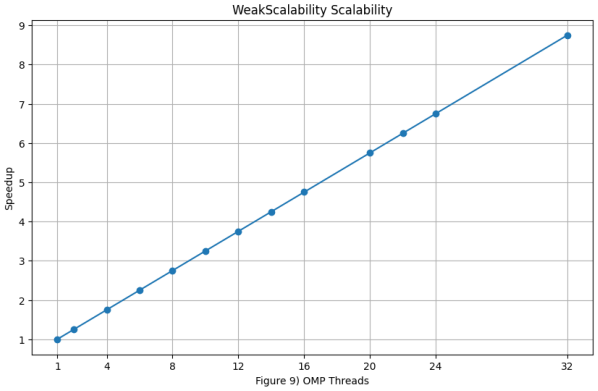


Figure 9) OMP Threads

▪ Conclusion

- Strong and weak scaling tests provided good indications for the best match between problem size and the number of nodes that should be requested for our model.

- The results of the **strong scaling** test for our parallel model to create the Mandelbrot image have shown that with 17 threads, we reached the minimum execution time of 0.002234 seconds and achieved a maximum speedup of 7.605192. However, the efficiency is only around half (0.447079) compared to running the program with only 2 nodes (0.977817). **Beyond 17 nodes,** the execution time will increase instead of decreasing, consequently, the speedup will decrease, which is not desirable and results in **wasted resources**. Therefore, if the **speed** of running the model is the priority, **17 nodes are the best** option. However, if **efficiency** is considered, only **2 nodes are sufficient** compared to the serial execution, resulting in the execution time being halved (0.008672 compared to 0.016990), while the efficiency remains approximately the same.

- The **weak scaling** test has revealed that by increasing the size of the problem proportionally with the increasing number of threads, the **execution time** remained **approximately constant until 32** nodes. Beyond that, the execution time started to grow sharply, despite the increase in threads, which is not beneficial as it consumes a lot of resources and yields worse results. With 32 nodes, we reached the maximum speedup of 8.75 among the 2 to 32 nodes. Therefore, if the **size of the image** is **small**, **smaller nodes** are a better choice, while for a **larger image size,** a l**arger number of threads** till 32 is the better option.

- ## Possible improvements on the code:

1. **Enhance error handling** for better feedback and graceful handling of edge cases.
2. **Optimize** the Mandelbrot algorithm to improve computational efficiency.
3. Implement more efficient **memory management** techniques to reduce overhead.
4. Experiment with **different parallelization strategies** to maximize performance gains.

The Codes, Slurms and Readme file are available on **Github** at:

https://github.com/SNB-Cs-Ds/hpc_projects/tree/main/project_3_OMP_Mandelbrot-Set