Mandelbrot Set

Tea Pashovska
University of Primorska – FAMNIT
Slovenia
89221054@student.upr.si

ABSTRACT

This paper describes how we created a program that draws the Mandelbrot set using three approaches: a simple version (sequential implementation), a parallel version that uses threads, and a distributed version that uses MPI. The program includes a graphical user interface (GUI) where users can explore the fractal by zooming and panning. We tested the performance of all versions and showed that parallel and distributed computing can make the program run much faster.

1 INTRODUCTION

The Mandelbrot set is a famous fractal that is created by repeating a simple math formula: $f_c(z) = z^2 + c$. For each point c on the screen, we check how many times we can repeat the formula before the result gets too big. This process is repeated for every pixel, which takes a lot of time, especially for high-resolution images.

To speed things up, we used parallel and distributed programming. This means we split the work between multiple threads or even different computers. We made three versions of the program:

- Sequential: the basic version that runs on one thread
- Parallel: uses Java threads to do the work faster
- Distributed: uses MPJ Express to run on multiple processes

We compared how fast each version runs and what benefits they bring. Each version was tested on different image sizes to see how they scale. The results helped us understand where parallel and distributed computing are most useful.

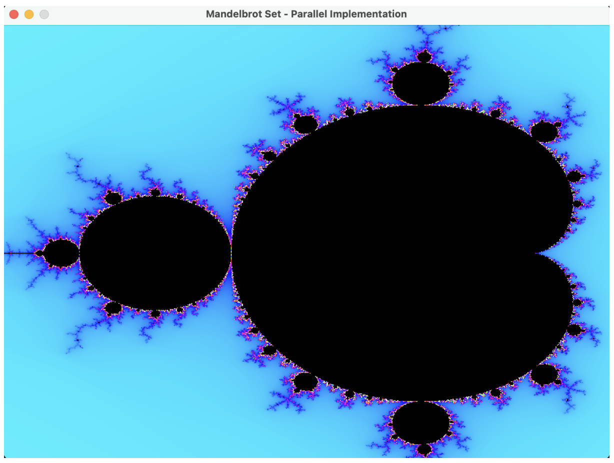

Figure 1: First look of the Mandelbrot Set

2 GRAPHICAL INTERFACE AND FEATURES

The GUI was built using Java Swing. When the program starts, users can enter the image size and choose between parallel or distributed mode.

- You can set width and height of the image
- You can choose which mode to run
- You can use keys to move and zoom into the fractal

The image updates right after each key press, so it's easy and fun to explore the fractal. Zooming and panning are controlled using arrow keys and +/-. Each change triggers a new rendering.

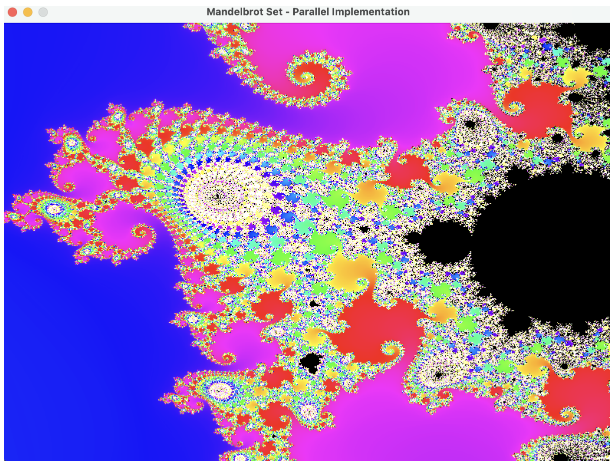

Figure 2: Zoomed look of the Mandelbrot Set

3 IMPLEMENTATION DETAILS

3.1 Sequential Algorithm

This version checks each pixel one by one using a loop. For each pixel, it calculates how quickly it escapes from the Mandelbrot set.

```
public static int computeColor(double zx, double zy) {
    double cX = zx, cY = zy;
    int i = 0;
    while (zx * zx + zy * zy < 4 && i < MAX_ITER) {
        double temp = zx * zx - zy * zy + cX;
        zy = 2.0 * zx * zy + cY;
        zx = temp;
        i++;
    }
    return i;
}
```

This method is slow but simple and is good for understanding how the Mandelbrot set works.

3.2 Parallel Implementation

In the parallel version, we split the image into horizontal rows and give them to different threads. Java's `ExecutorService` helps us manage the threads.

```
int threads = Runtime.getRuntime().availableProcessors();
ExecutorService executor = Executors.newFixedThreadPool(threads);
for (int y = 0; y < height; y += blockSize) {
    executor.execute(new MandelbrotWorker(...));
}
```

This makes the program much faster on modern computers. The number of threads is based on how many CPU cores the computer has, which helps make full use of the hardware. By splitting the image into row blocks, each thread can work on a separate part of the image without interfering with others. This not only improves speed, but also keeps the code simple and efficient.

3.3 Distributed Implementation

In the distributed version, we use MPJ Express to run the program on multiple processes. The main process sends parts of the image to other workers, and they send back the results.

We run it using:

```
mpjrun.sh -np 4 -cp .:$MPJ_HOME/lib/mpj.jar MandelbrotMPI
```

This approach works well for very large images or when using a cluster. Each process works independently on its assigned rows, which allows the program to run across multiple machines or CPU sockets. MPJ Express handles the communication between processes, making it easier to send and receive pixel data. While this setup adds some overhead, especially for smaller images, it becomes more efficient as the problem size grows.

4 TESTING AND RESULTS

To see how fast each version runs, we tested them on different image sizes. We turned off the GUI during testing to make sure only the computation time was measured.

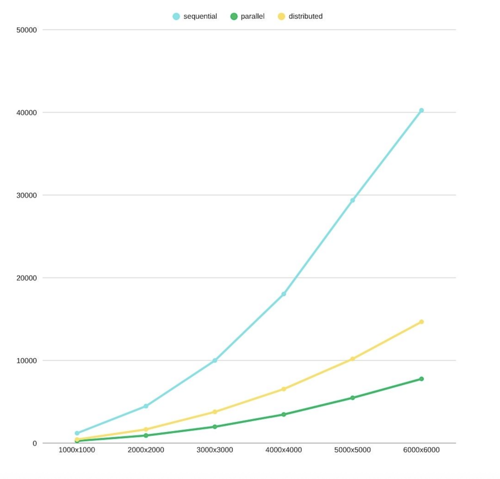

Figure 3: Execution Time vs Image Size for each mode

Table 1: Execution Time in milliseconds

Size	Sequential	Parallel	Distributed
1000x1000	1196	257	433
2000x2000	4467	911	1654
3000x3000	9982	1972	3771
4000x4000	18025	3459	6532
5000x5000	29363	5473	10177
6000x6000	40245	7764	14667

5 SPEEDUP ANALYSIS

Speedup tells us how many times faster the parallel and distributed versions are compared to the sequential version.

$$\text{Speedup} = \frac{T_{sequential}}{T_{method}}$$

Table 2: Speedup vs. Sequential

Size	Parallel	Distributed
1000x1000	4.65	2.76
2000x2000	4.90	2.70
3000x3000	5.06	2.65
4000x4000	5.21	2.76
5000x5000	5.36	2.89
6000x6000	5.18	2.74

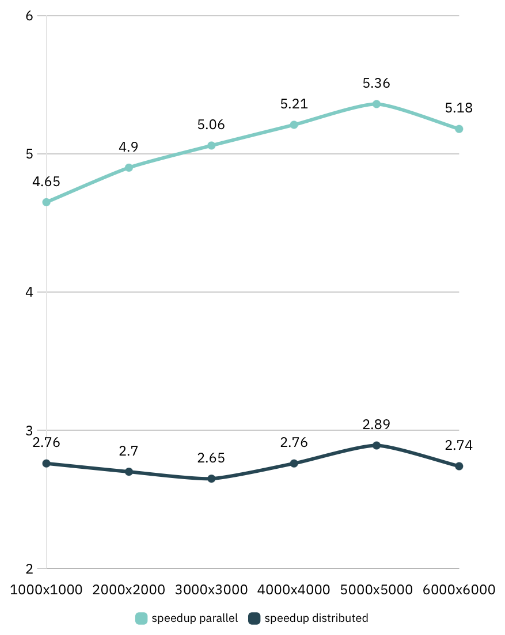

Figure 4: Speedup Over Sequential

6 DISCUSSION

The parallel version was much faster than the sequential one, with about 5 times speedup. This is because it used all the CPU cores effectively.

The distributed version was a bit slower than the parallel one at small sizes, but it gets better as the image size increases. This is because starting up MPI processes and sending data takes time.

In general, parallel programming is great for desktop performance, while distributed programming is better when you have more machines available. Another interesting point is how easily the parallel version scaled with image size. The speedup stayed consistent across all tests, which shows good efficiency. The distributed version, while more complex, gave us the chance to explore message-passing and how coordination between processes works. Even though it was slower for small tasks, it helped us understand how distributed systems handle workloads, and why communication overhead matters in real applications.

7 CONCLUSION

In this project, we built a Mandelbrot set renderer in Java using three different approaches. We added a GUI and explored performance with testing. Parallel processing gave the best results on one computer, while distributed processing can help with larger workloads.

REFERENCES

[1] Bhimani, R. and Baker, M. (2020). *MPJ Express: A Java Message Passing Library*. Available at: http://mpj-express.org/

[2] Oracle. (2024). *Java Platform, Standard Edition 17 API Specification - ExecutorService*. Available at: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ExecutorService.html

[3] Wikipedia contributors. (2024). *Mandelbrot set*. Wikipedia. Available at: https://en.wikipedia.org/wiki/Mandelbrot_set

[4] Ponce-Campuzano, L. (2023). *The Mandelbrot Set*. Complex Analysis: A Visual and Interactive Introduction. LibreTexts. Available at: https://math.libretexts.org/Bookshelves/Analysis/Complex_Analysis_-_A_Visual_and_Interactive_Introduction_(Ponce_Campuzano)/05%3A_Chapter_5/5.05%3A_The_Mandelbrot_Set

[5] Cicourel, I. (2020). *SwingWorker in Java*. Codementor. Available at: https://www.codementor.io/@isaib.cicourel/swingworker-in-java-du1084lyl