Mandelbrot Set

Branko Petrevski
FAMNIT
Koper, Slovenia
89221032@student.upr.si

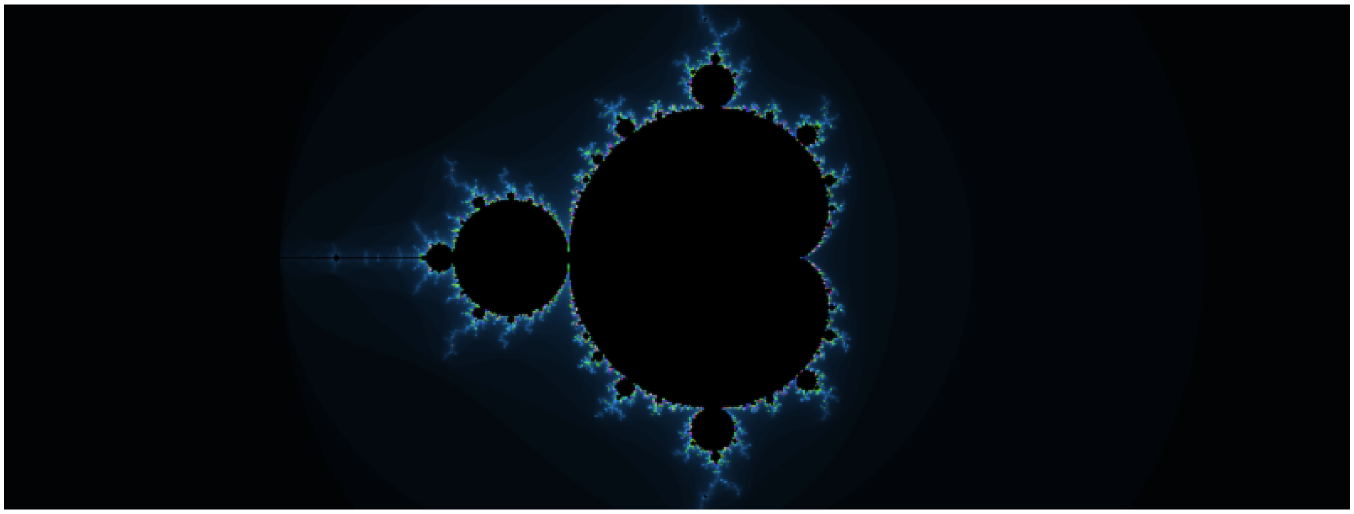

Figure 1: Representation of the mandelbrot set.

ABSTRACT

In this LaTeX paper is presented plotting of the mandelbrot set in the programming language Java using the set of graphics packages JavaFX. It contains the options panning and zooming on the canvas as well as saving an image of the current view-port with custom dimensions. The Java program works in sequential, parallel and distributive mode. The parallel part divides the image in even parts among the current number of threads and uses CyclicBarrier to await all threads to finish their computation before updating the fractal canvas.

ACM Reference Format:
Branko Petrevski. 2024. Mandelbrot Set. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Mandelbrot set is a mesmerizing and complex mathematical idea which has captivated the interest of mathematicians, scientists as well as artists. At a very fundamental level, the Mandelbrot set is an example of what we call a fractal. That means it possesses self-similarity at all scales. In order to understand how it is built, we need to explore the land of complex numbers: an extension (beyond) our traditional number line where now there exists a dimension called imaginary. The Mandelbrot set is generated by iterating a super simple little formula, plugging in complex numbers over and over again to see what happens. complex numbers c for which

$$f_c(z) = z^2 + c$$

may not diverge when iterated from $z = 0$. More explicitly,

$$\left\{f_c^n(z)\right\}_{n\in\mathbb{N}}$$

is pointwise-bounded in absolute value; here we define

$$f_c^0(z) = z \quad \text{and} \quad f_c^{n+1}(z) = f_c(f_c^n(z)).$$

The color version yiels a graphical representation of the set: to every point c in the plane is assigned an appropriate colour depending on does (and how fast) sequence fn converges. That is, rendering pixel (i,j) if they correspond to a complex num ber c by escape time processing algorithm.

Conference'17, July 2017, Washington, DC, USA
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Algorithm 1 Compute Pixel

```
procedure ComputePixel($cX, cY$)
    $zy \leftarrow 0$
    $zx \leftarrow 0$
    $iter \leftarrow 0$
    while $iter < maxIter$ and $zx^2 + zy^2 < 4$ do
        $tmp \leftarrow zx^2 - zy^2 + cX$
        $zy \leftarrow 2.0 \times zx \times zy + cY$
        $zx \leftarrow tmp$
        $iter \leftarrow iter + 1$
    end while
    return $iter$
end procedure
```

The `computePixel` method calculates the number of iterations required for a point in the complex plane to escape the Mandelbrot set's boundary. It initializes the complex number components `zx` and `zy` to zero and iteratively updates them using the formula `zx = zx * zx - zy * zy + cX` and `zy = 2.0 * zx * zy + cY` until the magnitude exceeds 4 or the maximum number of iterations is reached. The method returns the iteration count, which is used to determine the color of the pixel. Since every pixel is calculated independently, the algorithm naturally allows for parallelisation. It can be easily adapted in a threaded, process or distributed way on several machines. The following sections of this paper will describe the parallelization approaches we used and provide performance results with different levels of procs.

2 DESIGN OVERVIEW

The program aims to draw the Mandelbrot set in Java using JavaFX for GUI and rendering. This program generates the fractal image in an optimized manner using both sequential and parallel computation. Main class that will initialize the JavaFX application, setup canvas for rendering and manage Zooming/Panning of drawing.

In this non-parallel approach, we implemented a regular sequential computation that iterates over all pixels on the canvas and computes the corresponding point in complex plane for them one by one to then use escape time algorithm to color each pixel. They describe this method because of how easy it is to implement, which makes it practical for low iteration counts or images that are not too big. Its biggest weakness is that it is single-threaded and therefore slow for large images or high iteration counts.

The parallel computing solution divides the canvas into pieces that are worked on by different threads and supports synchronization via a ‘CyclicBarrier‘ mechanism to properly update the information when all thread have done their jobs. In this way, the computation is accelerated by multi-core processors and works faster for a large number of images or iterations. As a trade-off, they result in more complex implementations and have various thread synchronization issues.

The ‘computePixel‘ method, which states the escape time algorithm, applies Mandelbrot formula in a loop to understand how fast points are running from a certain boundary. Number of iterations is used to determine the color at each pixel and its calculated in calculateColor method. This allows us to use some optimization techniques like looking up if a point is inside either the main cardioid or period-2 bulb, so we can quickly skip it and not waste time computing an orbit. This value provide optimal balance between how accurate the fractal result later, and performance in generation it.

Algorithm 2 Main Algorithm

```
Initialize screen, iterations, zoom, positions
Parse arguments for width, height, mode
if distributive mode then
    Initialize MPI
    if MPI rank is 0 then
        Print settings
    end if
else
    Print settings
end if
Initialize pixel array
Execute mode (sequential, parallel, distributive)
Print commands
while input is not 'q' do
    Read input
    for each command do
        if empty or computing then
            Continue
        end if
        Handle command
        Start computation
        Print state
        Execute computation
        Finish computation
    end for
end while
if distributive mode then
    Finalize MPI
end if
```

3 PROBLEM DESCRIPTION

We will first discuss the issues that the CyclicBarrier and MPI interfaces are intended to address before going over how they are implemented. To achieve that, we will outline the precise actions that each of the program's four execution modes should take.

- **Sequential mode**: The following mode uses a single thread to execute all the necessary calculations and operations. First, it loads the width and height arguments and based on them initializes the application, canvas creation, and starts calculating pixels to display the Mandelbrot set. It is slower for bigger dimensions and a higher number of iterations.
- **Parallel mode**: In this mode, we divide the computational work of the previous method into a new method among a specified number of threads, which take care of an equal part of the fractal computation and return the results on

the canvas as soon as all of them are done. Then the canvas updates the image.

- **Distributive mode**: Same as parallel but instead of threads, we are using MPI to distribute the work among different nodes in a cluster. The results are then sent back to the main node and the canvas updates the image.

4 CYCLICBARRIER AND THREADS

In addition to specifying the *problem description* to be used in In the Mandelbrot project, a `CyclicBarrier` is used to synchronize multiple threads during the parallel computation of the fractal. Each thread processes a distinct chunk of the canvas, computing the Mandelbrot set for its assigned section. The `CyclicBarrier` ensures that all threads complete their computations before updating the canvas with the results. This approach leverages multi-core processors to significantly speed up the rendering process compared to a single-threaded implementation. By dividing the workload among threads and synchronizing them with a `CyclicBarrier`, the project achieves efficient and concurrent computation, enhancing performance for large images and high iteration counts.

The `CyclicBarrier` is initialized with the number of threads and a barrier action that updates the canvas. Each thread computes its chunk and calls the `await` method on the barrier, blocking until all threads reach the barrier. Once all threads have called `await`, the barrier action is executed, updating the canvas with the computed image. This method ensures that the canvas is only updated once all threads have finished their computations, maintaining synchronization and improving performance.
In Java, a thread is a lightweight process that allows concurrent execution of code. Threads are essential for performing multiple tasks simultaneously within a program, improving performance and responsiveness. Java provides the `Thread` class and the `Runnable` interface to create and manage threads.

To create a thread, you can either extend the `Thread` class and override its `run` method or implement the `Runnable` interface and pass an instance of it to a `Thread` object. The `start` method is called to begin the execution of the thread, which internally calls the `run` method.
These parts are implemented in the method computeParallel as follows:

```java
private void computeParallel() {

    long t0 = System.currentTimeMillis();

    this.barrier = new CyclicBarrier(cores, () ->
        {
        isComputing = false;
    });

    this.writableImage = new WritableImage((int)
        DEFAULT_SCREEN_WIDTH, (int)
        DEFAULT_SCREEN_HEIGHT);
    this.pixelWriter = writableImage.
        getPixelWriter();

    int chunkSize = (int) DEFAULT_SCREEN_HEIGHT /
        cores;

    for (int i = 0; i < cores; i++) {
        int startRow = i * chunkSize;
        int endRow = (i == cores - 1) ? (int)
            DEFAULT_SCREEN_HEIGHT : startRow +
            chunkSize;
        new Thread(new MTask(i,
                startRow,
                endRow,
                barrier,
                (int) DEFAULT_SCREEN_HEIGHT,
                (int) DEFAULT_SCREEN_WIDTH,
                cores,
                xPos,
                yPos,
                zoom,
                maxIter,
                writableImage,
                pixelWriter,
                graphicsContext)).start();
    }

    // Wait for all tasks to finish
    try {
        barrier.await();
        System.out.println("All threads have
            reached the barrier. Barrier action
            executed.");
    } catch (Exception e) {
        e.printStackTrace();
    }
    // Output computation time
    System.out.println("Parallel time: " + (System
        .currentTimeMillis() - t0) + "ms ");
    isComputing = false;
}
```

5 TECHNICAL REMARKS

The program was written in Java with some helper libraries to implement it. Certain aspects of the project rely on JavaFX features for rendering Mandelbrot set and user interactions, written purely as an academic exercise in order to get familiar with a lightweight tool bundled into recent versions of JVM. The program is further intended to be highly portable, and can run on any instance of the Windows or Linux operating system that has a JRE version which recent enough. The rest of the program configuration (including job specification) is loaded from command line parameters and we just write output to a PNG image because it is the most widely supported image format by mainstream image viewer. More technical details are in the subsections below.

5.1 Thread Synchronization

The use of `CyclicBarrier` ensures that all threads complete their computations before updating the canvas. This synchronization mechanism is crucial for maintaining consistency in the rendered image.

5.2 Parallel Computation

By dividing the canvas into chunks and assigning each chunk to a separate thread, the program leverages multi-core processors to speed up the Mandelbrot set computation. This parallel approach significantly reduces the computation time compared to a single-threaded implementation.

5.3 Dynamic Thread Management

The number of threads is dynamically set to the number of available processors (`Runtime.getRuntime().availableProcessors()`). This ensures optimal utilization of system resources.

5.4 Barrier Action

The barrier action updates the canvas with the computed image once all threads have reached the barrier. This ensures that the canvas is only updated when all computations are complete, preventing partial updates.

5.5 Error Handling

The code includes error handling for `InterruptedException` and `BrokenBarrierException` during the barrier synchronization, ensuring that any issues during thread execution are properly managed.

5.6 Zoom and Navigation

The `handleKeyPress` method allows for interactive zooming and navigation of the Mandelbrot set. This enhances the user experience by providing real-time feedback on the fractal exploration.

5.7 Color Calculation

The `calculateColor` method generates colors based on the iteration count, creating a visually appealing representation of the Mandelbrot set. Points inside the set are colored black, while points outside are colored based on their escape time.

5.8 Sequential and Parallel Modes

The program supports both sequential and parallel computation modes, allowing for performance comparison and flexibility in execution.

5.9 Canvas Resizing and Saving

The `saveImage` method allows users to save the fractal image as a PNG file with user-specified dimensions. This feature is useful for exporting high-resolution images of the fractal.

5.10 JavaFX Integration

The use of JavaFX for the graphical user interface provides a modern and responsive platform for rendering the Mandelbrot set and handling user interactions.

6 RESULTS

The tests were run on 4 core processor Intel i5 - 8300H with 8 threads for maximum number of iterations 2000 and for input string *"++++++++aaaaaaa++++++++++ssssssssss++++++++++++++awwww.+ ++++++++++++d–ssswwwwwwwwwwwwww++++++++++++++++++++"* Parameters that were varied was the width and height of the Mandelbrot canvas and were the following: 1000x1000, 2000x2000, 3000x3000 px. For each triple of parameters, the program was executed 7 times and the given results are the averages of those 7 times execution for every mode, sequential, parallel with 8 threads and distributive with 8 nodes. As we can see in the first graph, it illustrates the computation times for zooming and panning the Mandelbrot set with dimensions 1000x1000 using three different modes: sequential, parallel, and distributive. The x-axis represents different iterations, while the y-axis indicates the computation time in milliseconds. The sequential mode, represented by the blue line, consistently shows the highest computation times, peaking around 400 milliseconds at certain points. The parallel mode, depicted by the orange line, demonstrates significantly lower computation times compared to the sequential mode, maintaining a relatively stable performance with minor fluctuations. The distributive mode, shown by the green line, exhibits the lowest computation times overall, with a noticeable reduction in computation time as the iterations progress. Both parallel and distributive modes show a significant drop in computation time towards the end of the iterations, converging to minimal values. This analysis highlights the efficiency of parallel and distributive computing in reducing computation times for complex tasks like generating the Mandelbrot set. We can see similar for the second and third graph.

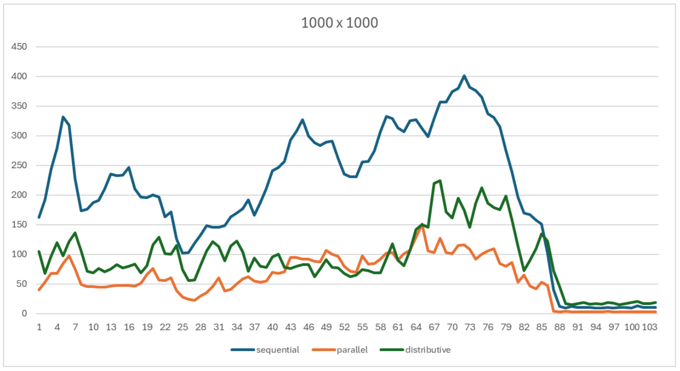

Figure 2: Speedup for width and height 1000

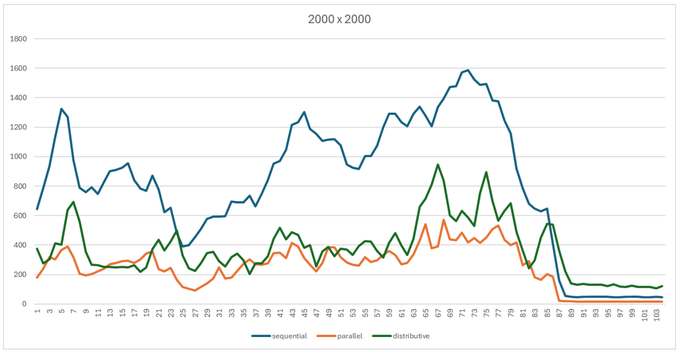

Figure 3: Speedup for width and height 2000

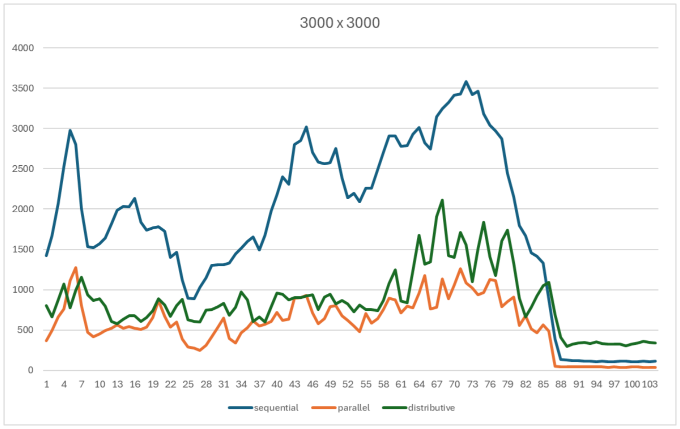

Figure 4: Speedup for width and height 3000

7 CONCLUSION

In conclusion, tests on a 4-core Intel i5-8300H with 8 threads and up to 2000 iterations reveal significant performance gains from parallel and distributed computing in Mandelbrot set generation. Parallel processing with Java threads and CyclicBarrier, and distributed computing with MPI, notably reduced computation times compared to the sequential approach.

For canvas sizes of 1000x1000, 2000x2000, and 3000x3000 pixels, parallel computing consistently outperformed sequential processing, while distributed computing achieved the best efficiency. Sequential times peaked at around 400 milliseconds, whereas parallel times were lower and stable, and distributed times were the fastest. Both parallel and distributed methods also showed reduced computation times as iterations increased.

These results highlight the effectiveness of parallel and distributed computing techniques in optimizing performance for complex tasks like Mandelbrot set rendering.