Mandelbrot Set

Luca Arrighetti
CS student at UP famnit
Koper, Slovenia
`luca.arrighetti@gmail.com`

August 16, 2024

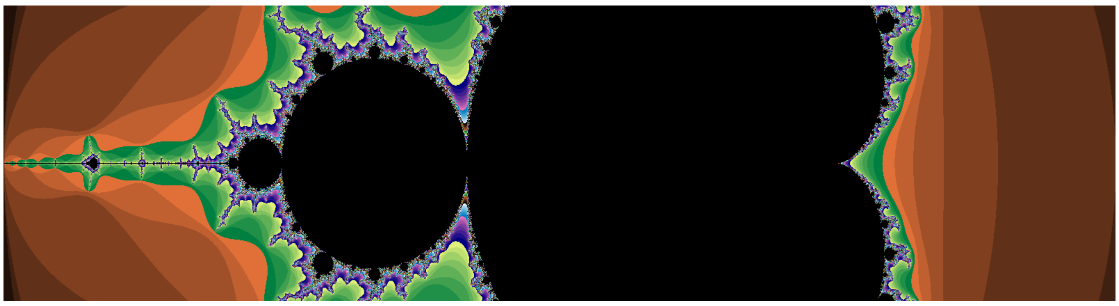


Figure 1: Mandelbrot set

Abstract

A Mandelbrot set is a set of points from a set complex numbers. In this paper we present a comprehensive framework for rendering the Mandelbrot set using the escape time algorithm. The core concept is to explore different execution models, including sequential, parallel, and distributed modes. The implementation is done in Java. For parallel processing, we employ a divide and conquer approach with ForkJoinPool. For distributed processing, we adopt a similar but less dynamic approach with MPI where each process independently computes a portion of the image before returning the results to a central process for final assembly. We present performance evaluations for each method.

1 Introduction

The Mandelbrot set is defined in the complex plane as the complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge to infinity when iterated starting at $z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc., remains bounded in absolute value.

The *escape-time algorithm* is used to generate a visual representation of the Mandelbrot set. This algorithm involves iterating a calculation for each point (x, y) in the plot area of the complex plane. For each point c, the sequence $\{f_c^n(z)\}$ is evaluated to determine whether it diverges. The number of iterations required for divergence is used to assign colors to each pixel, with different colors indicating how quickly the point bails out of the Mandelbrot set.

The escape condition is checked in each loop iteration. If the point escapes, the loop ends, and the pixel can be colored based on the number of iterations. Points that do not escape after the maximum iteration tend to go to infinity and are typically colored black to indicate they are part of the Mandelbrot set as seen in Figure 1.

Algorithm 1 Optimized Escape Time Algorithm [1]

```
Require: $x0, y0$                                   ▷ Initial complex coordinates
 1: $x2 \leftarrow 0$
 2: $y2 \leftarrow 0$
 3: while ($x2 + y2 \leq 4$ and $iteration < max\_iteration$) do
 4:     $y \leftarrow 2 \times x \times y + y0$
 5:     $x \leftarrow x2 - y2 + x0$
 6:     $x2 \leftarrow x \times x$
 7:     $y2 \leftarrow y \times y$
 8:     $iteration \leftarrow iteration + 1$
 9: end while
```

To render such an image, the region of the complex plane we are considering is subdivided into a certain number of pixels. Since calculating each pixel is an independent task, the algorithm can be parallelized efficiently.

2 Parallelization solution

In the calculation of whether a pixel belongs to the Mandelbrot set, each pixel's computation is independent. However, dividing the image into fixed blocks and assigning these to workers can lead to inefficiencies. Due to varying iteration counts across different regions of the Mandelbrot set, some workers may finish their tasks earlier than others, resulting in idle time and suboptimal resource utilization.

To address this issue instead of dividing the image into fixed-size blocks, we use a divide-and-conquer strategy to divide the image block into smaller sub-blocks, and this process is recursively applied until the sub-blocks are sufficiently small for asynchronous processing.

By generating more tasks than there are workers, this approach ensures that workers remain busy. When a worker completes its assigned task, it can immediately take on a new task from the pool of available sub-blocks, thus minimizing idle time and improving overall computational efficiency.

Algorithm 2 Parallel Mandelbrot Task Subdivision

```
 1: function PARALLELM(startX, startY, endX, endY)
 2:     if BlockSize(startX, startY, endX, endY) < minBlockSize then
 3:         ComputeMandelbrot(startX, startY, endX, endY)
 4:     else
 5:         $(midX, midY) \leftarrow FindCenter(startX, startY, endX, endY)$
 6:         PARALLELM(startX, startY, midX, midY)           ▷ Top-left quadrant
 7:         PARALLELM(midX, startY, endX, midY)             ▷ Top-right quadrant
 8:         PARALLELM(startX, midY, midX, endY)             ▷ Bottom-left quadrant
 9:         PARALLELM(midX, midY, endX, endY)               ▷ Bottom-right quadrant
10:     end if
11: end function
```

3 Distributed solution

In the MPI distributed mode for drawing the Mandelbrot set, the image is divided into horizontal strips, with each process assigned a specific strip based on its rank. The root process (rank 0) gathers the computed image data from all processes, which each has handle a portion of the image height asynchronously. Once all strips are collected, the root process assembles these strips to form the complete image.

Algorithm 3 MPI Mandelbrot

```
1: **Input:** Image dimensions $W \times H$
2: **Root process (rank 0):**
3: **if** rank == 0 **then**
4:     Divide the image into $P$ horizontal strips, where $P$ is the number of processes
5:     **for** each process $p$ from 1 to $P - 1$ **do**
6:         Send strip $p$ of the image to process $p$
7:     **end for**
8:     Compute strip 0 of the image
9:     **for** each process $p$ from 1 to $P - 1$ **do**
10:        Receive strip $p$ from process $p$
11:    **end for**
12:    Assemble all strips into the complete image
13: **else**
14:    Receive strip from root
15:    Compute strip
16:    Send strip to root
17: **end if**
```

4 Technical remarks

The program was implemented in Java using java.util.concurrent [2] library for parallelization and MPJ express [3] for the distributed version.

The implementation includes a JavaFX [4] GUI for choosing width and height values and able to live-draw the Mandelbrot set in a new windows as well as to permit zooming, panning and being able to resize window on runtime. The live drawing can be toggled off to save the Mandelbrot image to disk as a PNG file instead. The program can be run live in sequential and parallel mode as for distributed mode a separate program has been made as MPJ express cannot be made to run with JavaFX-Maven project dependencies. The drawing of the graphics are independent of the computational threads and the program adapts on the number of Cores and Physical CPU's its being ran on.

5 Performance results

The testing has been made on hardware that uses the the AMD Ryzen 5-7500F 6-Core CPU. Each core supports 2 threads. All of the cores have been utilized for the testing.

Testing has been made by starting with an image of 1000x1000 and increase both width and height by 1000 to 10.0000. All tests have been made 5 times per width and height for every mode and then averaged by how much time (ms) it takes to compute the Mandelbrot test without using graphics.

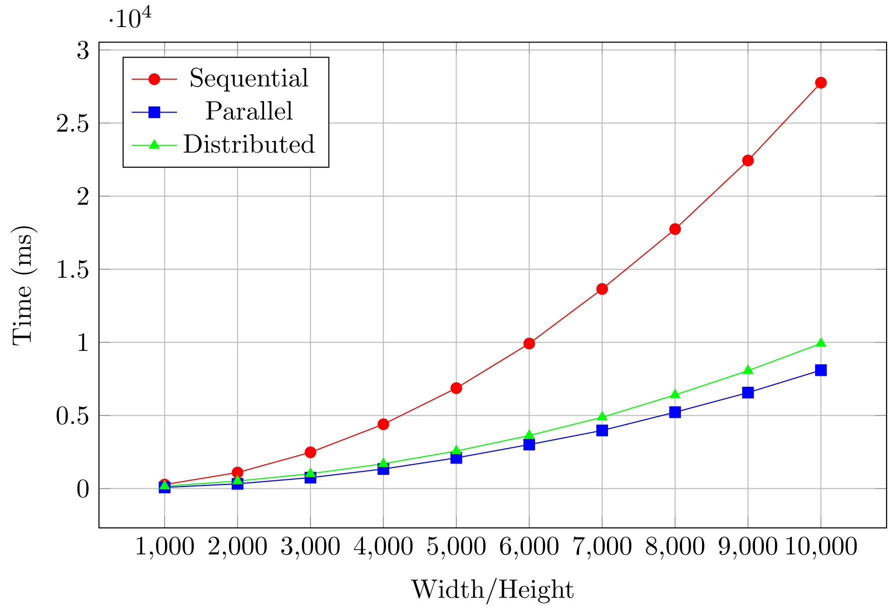
Table 1: Performance Comparison of Sequential, Parallel, and Distributed Methods

Width	Height	Sequential (ms)	Parallel (ms)	Distributed (ms)
1000	1000	275.0	80.2	167.8
2000	2000	1099.0	325.8	517.0
3000	3000	2481.2	748.8	1018.8
4000	4000	4401.0	1340.4	1700.8
5000	5000	6871.8	2100.0	2571.6
6000	6000	9920.8	3010.8	3631.4
7000	7000	13 653.2	3975.4	4880.6
8000	8000	17 748.0	5223.6	6408.6
9000	9000	22 441.4	6564.8	8067.0
10 000	10 000	27 761.4	8097.4	9929.2

Example of the sequential version quadratic growth pattern:

- 2000x2000 vs 1000x1000: $\frac{1099.0}{275.0} \approx 4$
- 4000x4000 vs 2000x2000: $\frac{4401.0}{1099.0} \approx 4$
- 8000x8000 vs 4000x4000: $\frac{17748.0}{4401.0} \approx 4$

All three methods increase time computation when increasing the image size, sharing the same growth pattern due to the algorithm's inherent complexity but each method scales differently. The time complexity of the Mandelbrot algorithm is $O(n^2)$, where n is the width/height of the square image. Given that the time quadruples when the input size doubles we confirm it with the quadratic growth pattern.



Sequential

The sequential method shows the least scalability with increasing problem size. Using one worker and knowing that computing each pixel is independent its a loss to not parallelize the Mandelbrot calculation. Therefore computing the Mandelbrot set proves as a very good example of the efficiency of concurrent programming.

Parallel

The parallel version is the fastest among the methods. Its performance could be increased by using a dynamic threshold for block size, which would optimize how the image is split based on its size. This adjustment would allow the system to adapt to different image dimensions more effectively, potentially improving processing speed.

Distributed

The distributed while slower then parallel appears having better scalability for smaller sizes but approaches quadratic growth for larger sizes. That is because the image is still small and dividing it in only 6 blocks is more efficient then the overhead associated with parallelizing the task in sub-tasks. For my testing the distribution version does not prove advantageous.

Distributed quadratic pattern growth:

- 2000x2000 vs 1000x1000: $\frac{517.0}{167.8} \approx 3.08$
- 4000x4000 vs 2000x2000: $\frac{1700.8}{517.0} \approx 3.29$
- 8000x8000 vs 4000x4000: $\frac{6408.6}{1700.8} \approx 3.77$

Adding more cores to the distributed method might improve performance however, the overhead associated with communication and synchronization between distributed nodes can offset these benefits. The distributed version should in theory improve if the Mandelbrot image is in such large scale that we require larger memory pools. But in our testing it cannot be confirmed.

6 Conclusion

For our implementation the algorithm works best when used with parallelization. The only bottleneck of the parallelization is how much memory it can manage before needing to use CPU clusters.

References

[1] Optimized Escape Time Algorithm. Wikipedia, The Free Encyclopedia. Accessed: 2024-08-16.

[2] Docs.oracle.com java.util.concurrennt.Java utility classes commonly useful in concurrent programming. Accessed: 2024-08-16.

[3] MPJ Express Proejct. Open source Java message passing library. Accessed: 2024-08-16.

[4] JavaFX. This library allows you to create Java applications with a modern, hardware-accelerated user interface.Accessed: 2024-08-16.