The Mandelbrot set

Lucas Jakin

UP FAMNIT
E-mail: 89211034@student.upr.si

In this paper, we present a unified framework for rendering the [1]Mandelbrot set. We will demonstrate how to solve the problem of rendering the Mandelbrot set in three ways and that is: serially (Sequential), with multiple threads (Parallel) or message passing between multiple single-threaded processes (MPI). The obtained performance measurements indicate performance gain in a case of different modes. Finally, we also describe the design of a simple GUI that interactively display the set (JavaFX).

1 Introduction

The Mandelbrot set is a two-dimensional set of complex numbers c for which the function

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

does not diverge to infinity when iterated starting from *z = 0*. In other words, the set

$$\{f_c^n(z) : n \in N\}$$

is bounded in absolute value. The start value of $f_c(0)$ is 0. To generate a graphical representation of the Mandelbrot set you calculate the speed of the convergence of every point in the set. The convergence value is mapped to a color. So every point in the set has a color and the result is a graphical representation of the Mandelbrot set.

Algorithm 1 Check convergence

```
0: procedure CHECK(ci, c, steps)
   z ← 0, zi ← 0
     for i=0 do steps
       ziT ← 2 × (z × zi)
       zT ← z × z − (zi × zi)
       z ← zT + c
       zi ← ziT + ci
       if (z × z + zi × zi ≥ 4.0) then return i

         return steps
```

The if-statement is used to check if the number converged. The condition is a simplification of $|x| > 2$. If z is greater than 2, the function diverges very fast so it leaves the method with the return. The number of needed steps i are the values of convergence (convergenceValue). The variable *steps* is needed to interrupt the loop for the divergent numbers.

This algorithm can easily be parallelized, and adapted to use multiple threads, processes and machines. Thus, the rest of the paper will describe the methods used for parallelization, and the measured performance results for different degrees of parallelism.

2 Design

In order to parallelize our computation, we must decide on a way to distribute the work among several (abstract) workers. Assuming that we have a fixed number of workers throughout the program execution, it would be the easiest to assign exactly one region of the resulting image to each worker (number of cores in the processing unit) and it would store the data in the region it is responsible for. However, one drawback of this approach is that, due to the structure of the Mandelbrot set, and the nature of the algorithm being used, some areas of the set require a much greater amount of processing than the others. We can mostly overcome this problem by having a larger number of regions than there are workers. This is done by prescribing a block size, and partitioning image into roughly equally-sized blocks. This design decision implies that we should design our program around a job queue, and possibly multiple workers that fetch jobs from that queue. Each worker will construct (render) their own block and when all workers are finished, the whole resulting image will be reconstructed. A lot of code can be reused between the serial and all variants of the parallel implementations of the algorithm.

3 Problem description

In this section we will describe the problems this interface is designed to solve. We will specify how exactly should each of the 3 execution modes of the program behave.

3.1 Sequential mode:

This is the most basic mode, that uses a single thread for everything. First it should state the position and dimensions of the final image. Then the image should be split into smaller parts, in this case the single thread will draw

a single line at the time and at the end render the resulting image. The program executed sequentially is very slow and it needs a lot of processing time each time the dimension of the window is increased. As we will see in testing, each time the dimension gets over 2000 units, the set will take a lot of time to render it (not render it at all).

3.2 Parallel (multi-threaded) mode:

Here we have a queue of threads (same number as cores in the processing unit) and a list of jobs. Each thread will work on their own job and will take care of it. After the worker threads have finished their work, they send back their chunks to the main thread, which will reconstruct the final image.
The algorithms below shows how would this operate:

Algorithm 2 Thread jobs

```
0: procedure THREAD-POOL(thPoolSize, lines)
   while index < ThPoolSize do
     task ← newLineTask
     threadPool ← executeTask(task)
     jobs ← addTask(task)
   end while
   for each task in jobs[] do
     drawChunkImage()
   end for
```

- The algorithm represents the creation of the thread pool and of the list of jobs that the threads must do. Each thread receives the task (LineTask class - runnable) which is drawing a part of the final image (chunk image).

3.3 Distributed (MPI) mode:

In this mode, we have to have at least two processes running, the root process or the server and at least one worker. The server handles the job loading, and waits for job requests from the workers. The workers should request jobs from the server, and process them. When the workers are done processing their jobs, they send back their chunks to the root process, which will use them to reconstruct the final image. With having in mind this idea it turns out that the principle for doing distributed part is very similar to the parallel part. The slight difference is that the distributed program creates a PNG file that represents the Mandelbrot set instead of rendering a JavaFX frame containing the set. To make this explanation easier to understand we can look at the next algorithm:

The following algorithm(Number 3) explains the communication between the root process and all the other workers. First of all it creates the final image that we will be then shown as result. The root process draws its own section and receives all the remaining sections from each process. It then reconstructs the final image by grouping all the received sections. All non root processes send their part when they are finished.

Algorithm 3 MPI processing

```
0: procedure MAIN
0:   finalImage ← createImage()
0:   if processNumber = 0 then
0:     createSection() - its own section
0:     for i=1 to MPI.size do
0:       received ← RecvMessage()
0:       finalImage ← drawSection(received)
0:     end for
0:     result ← saveImage(finalImage)
0:   else
0:     sendMessage(section) - other processes
0:   end if
```

4 Experimental results

This project was implemented with [2]JavaFX, which is a Java library used to develop Desktop applications. JavaFX is starting to replace **swing** since it provides more functionalities. For the distributed part I helped myself by using a dedicated library called [3]MPJ Express that allows the execution of parallel applications for multicore processors.

All three parts were tested on a Linux system (UBUNTU) that was installed inside Virtual Box, so the final results may be a little misleading. Besides they were tested on a laptop with an i7-1165G7 CPU which has 4 cores. VirtualBox was set up to use 2 cores out of 4 total. I have tested parallel part with the number of threads being the same as the number of cores available. The distributed part was tested using 2 and then 4 processes.

Figure 1: #seconds (milliseconds) to render the set

In Figure 1 we can see the difference in speed of the program being executed sequentially and parallely.

Figure 2 shows us the minor difference in speed while running on 2 or 4 processors.

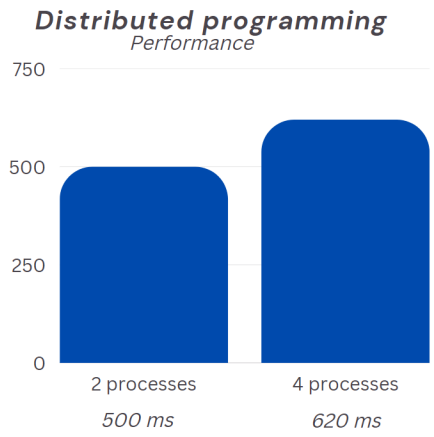

Figure 2: Analysed tweets per second

4.1 Discussion

By analysing both graphs we can see that:

- The sequential program is the slowest. It needs almost 12500 ms to 15198 ms (800*600) to render the whole frame of the Mandelbrot set. Moving trough the frame is actually not that slow, the rendering after resizing the window becomes slow instead.
- Parallel and distributive programs have very similar results, since the parallel one is quicker by just some milliseconds. Also this is expected result since both modes were tested with same number of threads/processes.
- Both sequential and parallel programs enable the user to move around and zoom in/out the set. By pressing the ENTER key, a snapshot of the set will be stored on the same directory as the programs are located.

5 Conclusion

The project has been an exciting exploration of the fascinating world of fractals, parallel computing, and graphical user interfaces using Java and JavaFX. Leveraging the visualization capabilities of JavaFX, we are able to generate high-resolution images of the Mandelbrot set in real-time. One of the key highlights of the project is the utilization of parallel computing techniques using Java's MPJ Express library. By breaking down the computation into smaller tasks and distributing them across multiple processors, we can achieve significant performance improvements in generating the Mandelbrot set for various zoom levels and resolutions. This demonstrated the power of parallel computing. In conclusion, the Mandelbrot set project has been an engaging and enriching experience that highlights the incredible capabilities of Java and JavaFX for both scientific computing and graphical user interface development. By combining the beauty of fractals with the power of parallel computing, we can create an interactive and visually stunning application that not only entertains but also educates users about the wonders of mathematics and computer science.

References

[1] Mandelbrot Set

[2] JavaFX

[3] MPJ Express