Fractal Generation using Parallel Strategies

C

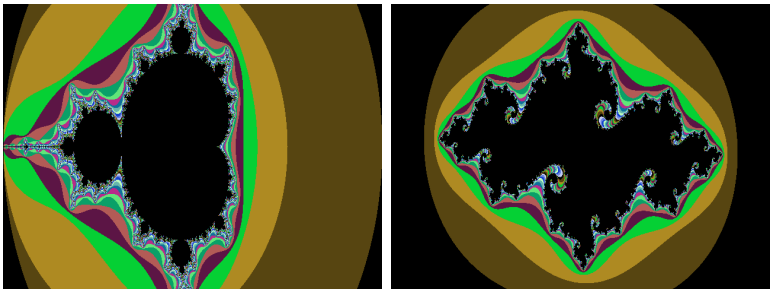

Figure C.1: Left: Julia fractal; Right: Mandelbrot fractal

C.1 Fractals

The word Fractal was coined by Benoît Mandelbrot, and it means "irregular" or "broken" on one hand, and "fragment" on the other. A fractal is generated from a mathematical formula, often simple, but when applied iteratively, produces graphically appealing results.

There are two categories of fractals: geometric fractals, which continuously repeat a standard pattern, and random fractals, which are generated using computers.

One of the main characteristics of fractals, which are geometric structures, is self-similarity. This can be described as the property of the fractal's structure being similar across different scales. In other words, self-similarity is symmetry across scales. This self-similarity can be exact or statistical.

This guide provides the code to generate fractals and save them as PGM image files, like those in Figure J.1. This task is designed to be performed on Linux, but if you wish to view these files on Windows, you can start by converting them to PNG or BMP format using a utility such as `convert` from the ImageMagick package, available for Windows, Linux, or macOS.

```
$ convert julia.pgm julia.png
```

C.2 Work to Develop

Part I

Exercise C.1

Start with the provided code for generating fractal images, whether of the Julia fractal or the Mandelbrot fractal. Compile and execute it. Observe how execution times can become extremely long when generating high-resolution images. See the following example.

For the initial tests, you may use a low image resolution (HD, for example), but for the final tests and the writing of the project report, you should use a 4K resolution (3840x2160), taking care to adapt the value of the necessary iterations to ensure the detail reaches the pixel level.

-20% if not done

Exercise C.2

Note that the generation of the fractal is evolutionary, with the number of iterations being important to create an increasingly detailed image (where the detail, and thus the number of necessary iterations, are a function of the image resolution).

Create a modified version of the code to save an image (PGM format) per iteration, in the format `fractal_%02d.pgm`[1] (e.g., `julia_35.pgm`), showing the evolution of the fractal's construction "over time."`-20% if not done`

Convert the set of images into an MP4 video format using the **ffmpeg** application as follows:

View the generated video and observe the evolution of the fractal. Measure the exact time taken to generate each of the fractal's images. These times will be used as a baseline for the improvements to be introduced with OpenMP.

The videos must have a minimum duration of 30 seconds to allow for comparisons, both with and without OpenMP optimization.

20%

[1] analise the use of `sprintf...` function

Exercise C.3

Create a modified version of this code that uses OpenMP to distribute the computation across the available processors on a single machine. You should modify only the `void Generate(int)` function without changing the `julia()` and `mandelbrot()` functions. 15%

Exercise C.4

Perform a detailed study on the impact of each OpenMP directive on the overall execution time, calculating the speedup for each modification and critically analyzing each change, as well as the solution with the best overall speedup. 20%

Exercise C.5

Create a new version of this code that also includes MPI to distribute the computation across the available nodes. You should guarantee the correctness of the results and choose the best communication strategies. 15%

Part II

Exercise C.6

Starting from the matrix of the generated fractal image, we will simulate the diffusion of colours over time. Thus, we will successively generate new images where the image I_{k+1} is generated from I_k by diffusion. For a pixel with coordinates i, j, its value is given by

$$\begin{aligned} I_{k+1}(i,j) &= (1-\alpha)I_k(i,j) + \alpha\frac{1}{8}\left[I_k(i-1,j-1) + I_k(i-1,j) + \right. \\ &+ I_k(i-1,j+1) + I_k(i,j-1) + I_k(i,j+1) + \\ &+ \left. I_k(i+1,j-1) + I_k(i+1,j) + I_k(i+1,j+1)\right], \end{aligned}$$

where α is the parameter that controls diffusion, weighting the contribution of the pixel in the same position versus its 8 neighbours. Extreme values result in no diffusion ($\alpha = 0$) or very rapid diffusion through the neighbours ($\alpha = 1$).

At the end of generating each new image, it should be saved in a file named in the format "output%04d.jpg" (see the `sprintf` function), where %04d will be replaced by the file's sequence number.

Insert the code to perform diffusion in the designated area of the `Diffusion(...)` function, taking advantage of OpenMP to speed up the process. Note

that to enable the call to the diffusion function, you must pass two additional parameters: the number of diffusion epochs (iterations) and the alpha factor, which should be between 0 and 1.

```
$ make
$ ./fractal 3840 2160 100 0.5
```

30%

Exercise C.7

Adapt the code to take advantage of MPI to distribute the execution over a cluster of machines and improve the performance. Describe the results.

20%

Exercise C.8

Prepare a brief report describing the approaches and comparing the execution times without any parallelism support vs OpenMP, vs OpenMP+MPI. **This report must be submitted along with the source files in a ZIP archive. Note that this report will weight along with the oral discussion the final grade.**.

Mandatory