PLSPS 2025 Assignment - Parallelising Fractal Zoom Animations

Mikhail Cassar

September 2025

Contents

1 Introduction

The Mandelbrot Set

The **Mandelbrot set** is one of the most famous examples of a fractal in mathematics. It emerges from a simple recursive rule applied to complex numbers and exhibits an infinitely detailed boundary with self-similarity, see `https://en.wikipedia.org/wiki/Mandelbrot_set` for more details.

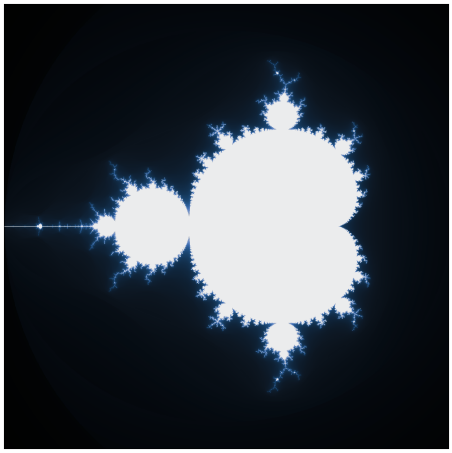

Figure 1: The Mandelbrot set visualised on the Complex Plane

The Mandelbrot set is a subset of the complex numbers $c \in \mathbb{C}$, for which the sequence z_n, generated by the following recurrent relation:

$$z_n = z_{n-1}^2 + c, \quad \text{with } z_0 = 0$$

remains bounded as $n \to \infty$ i.e. the magnitude $|z_n|$ does not diverge to ∞.

The recursive process can also be equivalently represented as a function iteration. Let $f_c(z) = z^2 + c$ be a function parametrised by c. The n-th term of the sequence can then be written as follows, where f_c^n denotes the n-fold composition of f_c with itself.

$$z_n = f_c^n(0)$$

With this, we can define the Mandelbrot set more formally as follows.

$$\mathbb{M} = \{c \in \mathbb{C} : \lim_{n\to\infty} |f_n^c(0)| < \infty\} \tag{1}$$

This implies that a complex number $c \in \mathbb{C}$ belongs to the Mandelbrot set $\mathbb{M}$ if the sequence $f_c(z) = z^2 + c$, starting from $z_0 = 0$, remains bounded.
Since it is not possible to compute an infinite number of iterations, we approximate this condition using an **escape time** algorithm, shown in Figure 2.

```julia
function mandelbrot(c::Complex, M::Int)
    z = c  # Start with z_0 = 0 + c
    for iter in 1:M
        if z.re^2 + z.im^2 > 4  # Check escape condition
            return smoothiter(iter, M, z)
        end
        z = z^2 + c
    end
    return M  # Considered bounded
end
```

Figure 2: The `mandelbrot` function, defined in `src/sequential.jl`. Go to Section 6 for more details.

The Sequential Implementation

The goal of this assignment is to parallelise a sequential program that generates a zooming fractal animation for some specific point in the Mandelbrot set, the `generate_fractal_seq!` function, shown in Figure 3.

```julia
function generate_fractal_seq!(FP::FractalParams,
                               data::Array{Float64, 3})

    N, F, M, center, alpha, delta = extract_params(FP)

    @assert size(data) == (N, N, F) "Size mismatch: data has size
    ↪  $(size(data)), but expected ($N, $N, $F)"

    t = @elapsed begin
        @inbounds for f=1:F                     # Frames
            local_delta = delta * alpha^(f -1)
            x_min = center[1] - local_delta
            y_min = center[2] - local_delta
            dw = (2 * local_delta) / N

            @inbounds for j in 1:N              # Columns
                y = y_min + (j - 1) * dw
                @inbounds for i in 1:N          # Rows
                    x = x_min + (i - 1) * dw
                    z = Complex(x, y)
                    data[i, j, f] = mandelbrot(z, M)
                end
            end
        end
    end
    t
end
```

Figure 3: The `generate_fractal_seq!` function, defined in `src_sequential.jl`. For a more detailed overview of this function, refer to Section 6.

2 Parallel Implementation

The Parallel Implementation

The sequential implementation shown in Figure 3, generates F frames, each of resolution $N \times N$, resulting in N^2 pixels per frame. For each pixel, the `mandelbrot` function is called, which performs up to M iterations. Thus, the total computational workload of the `generate_fractal_seq!` function has an upper bound of:

$$\mathcal{O}(F \cdot N^2 \cdot M). \tag{2}$$

We can consider three parallelisation strategies for distributing the workload in Equation 2 among processors.

- `mandelbrot` **Iterations** (M) – **Not parallelisable** due to dependecies introduced by the recurrence $z_n = z_{n-1}^2 + c$.
- **Pixels** (N^2) – **Parallelisable**, however results in communication overhead at each frame and adds complexity when distributing jobs.
- **Frames** (F) – **Parallelisable**, each frame corresponds to a different zoom level and be computed independently (using the closed-form expression $\delta_f = \delta_0 \alpha^f$) without inter-processor communication.

In this assignment we will focus on **frame-based** parallelism, where each processor is assigned one (or more) frame.

Load Imbalance

Computing a frame involves evaluating all $N \times N$ pixels, each requiring up to M iterations in the `mandelbrot` function. As a result, the computational cost per frame is approximately $\mathcal{O}(N^2 M)$.

The workload per job may vary since the the workload per pixel varies, points inside the Mandelbrot set take longer to compute than those outside. This may lead to **load imbalance** where some processors finish early and stay idle.

Load Balancing Strategies

To mitigate load imbalance, the workload should be distributed so that all processors perform roughly the same amount of work.

- **Static Load Balancing** assigns frames evenly divided among processors before runtime. This method will still result in load imbalance, as some frames require more work than others.

- **Dynamic load balancing** assigns frames at runtime as processors become idle, ensuring that all processors are working during runtime. Dynamic strategies are preferable when workload is unpredictable, since we cannot trivially determine which pixels in a frame take the longest to compute.

Replicated Workers Model

In this assignment you will implement a **Replicated Workers** dynamic load balancing strategy. A master process maintains a queue of unprocessed frames. As workers finish their current task, they request the next available frame. This approach keeps all processors busy, regardless of how computationally expensive each frame is.

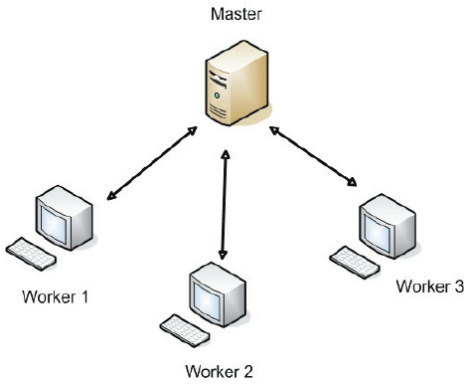

Figure 4: Replicated Workers Model

3 Assignment

Objectives

1. **Implement Replicated-Workers Model**
   - Implement the master and worker logic in the `src/parallel.jl` file.
   - Master logic - `generate_fractal_par!` function.
   - Worker logic - `work!` function.
2. **Test your Implementation**
   - Validate your parallel implementation on DAS-5 using the `jobs/testing_job.sh` jobs script.
   - Your implementation should pass at least 75% of the total tests.
3. **Benchmark your Implementation**
   Measure the timing of your parallel implementation on DAS-5 using the `jobs/benchmark_job.sh` job script.
   - **Strong Scaling Experiment**: For each of the following fixed problem sizes, measure how the runtime changes as the number of worker processors P increases.
   - The problem sizes to be tested are $(N, F) \in \{(1024, 30), (2048, 5), (512, 120)\}$.

$$P \in \{2^0, 2^1, 2^2, 2^3, 2^4, 2^5\}$$

4. **Prepare a Short Report**
   - Fill in the **(a)** Design, **(b)** Results, **(c)** Analysis, and **(d)** Conclusion sections in the report.

Grading

Objectives	**Weight**
Parallel Implementation	40%
Testing	10%
Benchmarking	25%
Report	25%

Table 1: Grade Distribution

4 Deliverables

Submissions that do not follow these instructions may receive a penalty.

- Build the parallel implementation using the Julia `Distributed` package.
- Your submission must run correctly on the DAS-5 cluster using the provided job scripts.
- Final submission must be a .zip file named `PLSPS_Assignment_[VUNetID].zip` to Canvas. The archive must contain the following two files:
  - The parallel implementation, `parallel.jl`.
  - The report (in PDF format), named `PLSPS_Report_[VUNetID].pdf`.
- **The final submission deadline is 23/10/2025 at 21:00**.

5 Contact

For any questions about this assignment, contact either Mikhail Cassar (`m.cassar@vu.nl`) or Chiara Michelutti (`c.s.michelutti@student.vu.nl`).

6 Appendix

Implementation Details for `mandelbrot` function

Implementation The function iterates the sequence $z_{n+1} = z_n^2 + c$ up to a fixed number of steps, `max_iters`.

- If at some iteration less than `max_iters` the sequence $|z_n|$ exceeds a certain threshold (typically 2), this indicates that the sequence diverges and $c \notin M$.
- If the sequence remains bounded within the threshold for all `max_iters` iterations, we assume that $c \in M$.

Optimised escape condition To avoid computing the square root in $|z_n| > 2$, (since $|x| = \sqrt{x^2}$) we use the following equivalent escape condition.

$$\mathbf{Re}(z_n)^2 + \mathbf{Im}(z_n)^2 > 4$$

Return value The `mandelbrot()` function in Figure 2 returns the number of iterations it took for the sequence to escape ($c \notin \mathbb{C}$), or `max_iters` if it did not escape ($c \in \mathbb{C}$). This approach is useful for visualisation, as it allows us to represent not just membership, but how quickly each point diverges.

Smooth Iterations The `smooth_iterations` function in Figure 5 is used to create a continuous colouring effect when visualising the Mandelbrot set. Find more information here.

```
1 function smooth_iterations(iter::Int, max_iters::Int, curr_z::Complex)
2     return iter - (log(log(abs(curr_z))) -
      ↪  log(log(max_iters)))/log(2.0)
3 end
```

Figure 5: `smooth_iterations` function, defined in `src/sequential.jl`

Implementation Details for `generate_fractal_seq!` function

Here we will discuss the process of visualising a square subset of the complex plane $\mathbb{C}$, implemented in the `generate_fractal_seq!` function. We will **(1)** define a square region in the complex plane $\mathbb{C}$, then **(2)** discuss how to zoom into such a region, and **(3)** then define a mapping from the image plane to the complex plane.

Define a Square subset in $\mathbb{C}$

Every complex number $c = x + iy \in \mathbb{C}$ can be represented in two dimensions as $(x, y) \in \mathbb{R}^2$, allowing us to treat $\mathbb{C}$ as a $2D$ space. The *Real* and *Imaginary* numbers are on the x-axis and y-axis respectively.

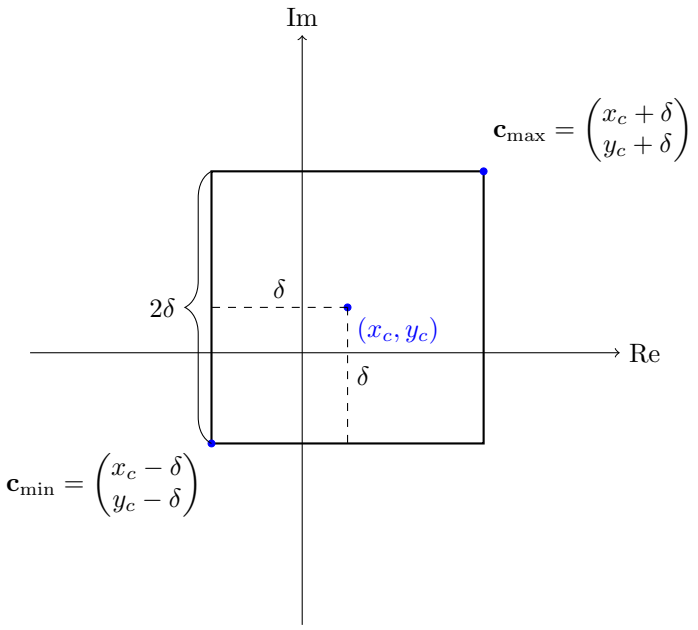

Figure 6: Region in the complex plane defined by `center` and δ.

Figure 6 shows a square subset defined in $\mathbb{C}$ using two parameters. Let the `center` of this square region be $(c_x, c_y) \in \mathbb{R}^2$, and let $\delta > 0$ denote half the width and height of the square (note that 2δ is the length of the square). Then, the region spans from $\mathbf{c}_{\min}$ to $\mathbf{c}_{\max}$.

$$\mathbf{c}_{\min} = \begin{pmatrix} x_c - \delta \\ y_c - \delta \end{pmatrix}, \quad \mathbf{c}_{\max} = \begin{pmatrix} x_c + \delta \\ y_c + \delta \end{pmatrix} = \mathbf{c}_{\min} + 2\delta \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{3}$$

Zooming into a Region

The square parametrisation defined in Section 6 allows us to easily perform a zoom operation within the square. Performing a zoom operation involves keeping the `center` point fixed and scaling the length 2δ of the region by some zoom factor $\alpha \in [0, 1]$.

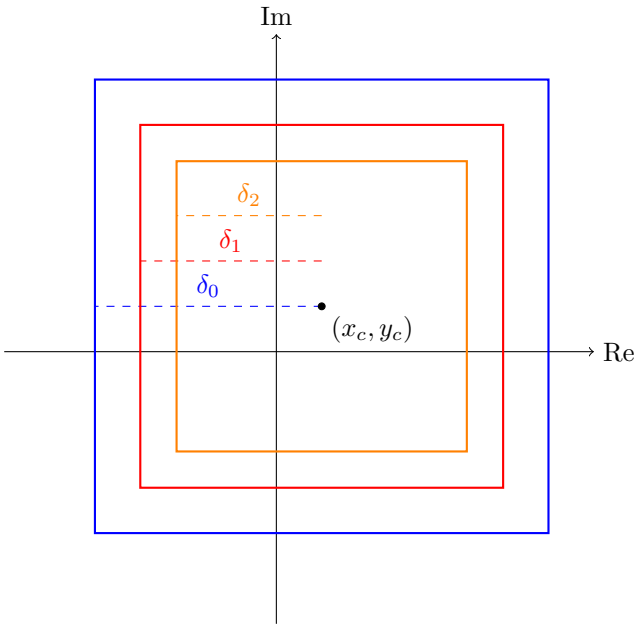

Figure 7: Zooming into a region centered at (x_c, y_c) with exponentially decreasing $\delta_f = \delta_0 \cdot \alpha^f$.

Figure 8: Region in the complex plane defined by `center` and δ_0.

The `generate_fractal_seq!` function generates a zooming fractal animation by rendering a sequence of F frames. At each frame, the half-length δ is progressively decreased based on α, while the `center` point remains fixed as shown in Figure 8.
The zoom level δ_f at frame $f \in \{0, 1, ..., F-1\}$ is computed by applying the zoom factor α, resulting in exponential decay.

$$\delta_f = \delta_0 \cdot \alpha^f$$

where δ_0 is the initial scale δ of the region.

Mapping Pixels to $\mathbb{C}$

To visualise the Mandelbrot set, we first define a mapping from each pixel in an $N \times N$ image to a corresponding point in the complex plane.

The image is treated as a grid of pixels with integer coordinates (i, j), where $i, j \in \{0, 1, \ldots, N-1\}$. The goal is to map each pixel to a complex number $c = x + iy \in \mathbb{C}$, where (x, y) lies in a square region of the complex plane (as defined in Section 6).

To discretise this region into pixels, we divide each axis, having length 2δ, into N evenly spaced steps. This yields a pixel spacing of:

$$dw = \frac{2\delta}{N}.$$

Each pixel coordinate (i, j) is linearly mapped to a point $(x, y) \in \mathbb{R}^2$ within the complex region using:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{c}_{\min} + dw \cdot \begin{pmatrix} i \\ j \end{pmatrix}.$$

This formula performs a linear interpolation from the bottom-left corner ($\mathbf{c}_{\min}$) of the region to the pixel location by scaling the pixel indices by the step size dw. As a result, we obtain a uniform grid of complex values, each of which corresponds to a pixel in the image and serves as input for the Mandelbrot iteration.