Mandelbrot & Parallelism

William Frid

Spring 2024

Introduction

The Mandelbrot set is a famous 2 dimensional set of imaginary numbers that can be used for generating patterns. More precisely, if one has a canvas, and let for instance the x-axis represent the real numbers, and y-axis represent the imaginary numbers (plus an offset for more effect), then calculating the depth of the complex number collected. This depth can then alongside a maximum value be used for color selection for that specific pixel.

These calculations can be done easily using the formula $z_{n+1} = z_n^2 + c$ and some experimentation to find an algorithm that can generate some "fun" colors. However, performing these calculation, let's say maximum depth 1024 over $1920x1080 = 2073600$ positions will results in a tremendous amount of calculations. This is where parallelism comes in handy as we can speed up the process by dividing the workload among multiple processes and threads on our CPUs.

Mandelbrot

The first thing required to work with the Mandelbrot set is support for complex numbers. This can easily be built for Elixir trough basing them on tuples ($: cmpx, 3, 4 = 3 + 4i$) and the four functions `new/2`, `add/2`, `sqr/1`, and `absolute/1` (no snippet included since these are unambiguous).

When complex numbers exists, one can start working on the calculating based on the formulas seen in the introduction above. This can be carries out with just the two functions seen below: `mandelbrot/2`, and `test/4`. `mandelbrot/2` takes two arguments, a complex number, and a maximum depth. The maximum depth is important to prevent the program from diving too deep in the problem causing too long duration of calculations. It's only purpose is to prepare a call to `test/4`, which is where the calculations takes place.

The functions `test/4` takes the four arguments, run-number, last complex number, original complex number, and maximum depth. It then checks, if the absolute value of the current complex number is greater than 2, if so,

it stops the loop as the depth is 2. Otherwise, it dives deeper and adds the current complex number to the original one. This goes on until the depth is found.

```
def mandelbrot(c, m) do test(0, Cmplx.new(0, 0), c, m) end
def test(i, _z, _c, m) when i == m - 1 do 0 end
def test(i, z, c, m) do
    if (Cmplx.absolute(z) > 2) do
        i
    else
        test(i + 1, Cmplx.add(Cmplx.sqr(z), c), c, m)
    end
end
```

Colors & Rendering

Calculating the colors can be done in many different ways depending on the desired outcome and access to computational power (more on thus under section parallelism). One way of doing this it to create a function `convert/2` that takes two arguments, depth, and max. It calculates the fraction $\frac{depth}{max}$ and then does something with this. One option is seen below, which is based on a sinusoidal wave.

```
def convert(depth, max) do
if (depth > max) do convert(max, max) end
    x = (depth / max) * 10
    red     = round(abs(:math.sin(x*(1/3)*:math.pi) * 255))
    green   = round(abs(:math.sin(x*(2/3)*:math.pi) * 255))
    blue    = round(abs(:math.sin(x*(3/3)*:math.pi) * 255))
    {:rgb, red, green, blue}
end
```

On top of this, a function is required to generate the PPM-image data. This can be done using the three functions `mandelbrot/6`, `row/5`, and `columns/5`. Where `mandelbrot/6` starts recursive calls between the other two functions that generates lists (columns) withing a list (rows) containing tuple (RGB data). This will be further discusses under section Parallelism as that sections covers more complex and reliable versions of these functions.

Base Run

As mentioned in the introduction, generation of Mandelbrot patterns is a heavy operation for computers (when done on a single thread) to perform as it requires a large amount of calculations. This was tested trough the

generation of two images, see table 1 below. The results clearly shows that this is not a viable solution as one might strive for higher resolution and depth to achieve more beautiful patterns and zoom opportunities.

Width	Height	Depth	Time (s)
1920	1080	64	16
1920	1080	1024	84

Table 1: Benchmark exploring the time required to generate Mandelbrot patterns using a single thread.

Parallelism

To improve the calculation speed one can implement parallelism. That is, perform the calculations more effectively by diving the workload among multiple threads (in this case). Since this lab relies on solely Elixir, one has to do with only having access to CPU threads and no CUDA cores, but this will still suffice. Note that the computer used for this report had access to 6 cores all running at 4 GHz, that is support for 12 threads.

Parallelism for this task can be achieved by modifying how the program performs the rendering of the image. The idea is to spawn a new process for each row in the image, listen for each success, and assemble the resulted rows in the correct order. This can all be started with a modified version of `mandelbrot/6` that looks something as the snippet below. Note how the function `collect/2` is called in the end. This is a new "state" for the process that is made to receiver successfully generated rows in the correct order.

```
def mandelbrot(width, height, x, y, k, depth) do
   trans = fn(w, h) ->
    Cmplx.new(x + k * (w - 1), y - k * (h - 1)) end
    rows(width, height, trans, depth, self())
    collect(height, [])
end
def collect(h, rows) when h == 0 do rows end
def collect(h, rows) do
    receive do
        {:row, ^h, row} -> collect(h - 1, [row | rows]) end
end
```

The new `row/5` is too somewhat different as it now takes the caller process as an argument. This function is the the spawning the new processes causing the parallelism. This is done trough the Elixir function `spawn/1` that it uses to call upon `columns/5`. Now, `columns/5` isn't complicated at all. It

simply generates the row, and sends the result via `send/2`, and the response is caught by `collect/2`.

Improved Run

The results for the improved version, using parallelism, can be seen in table 2. We see that this new implementation came with a drastic increase of performance and that way more detailed patterns could be generated. Beneath the table, you can see the resulting image 1 and the zoome din version image 2.

Width	**Height**	**Depth**	**Time (s)**
1920	1080	64	2
1920	1080	1024	15
7720	4320	1024	435

Table 2: Benchmark exploring the time required to generate Mandelbrot patterns using a multiple threads.

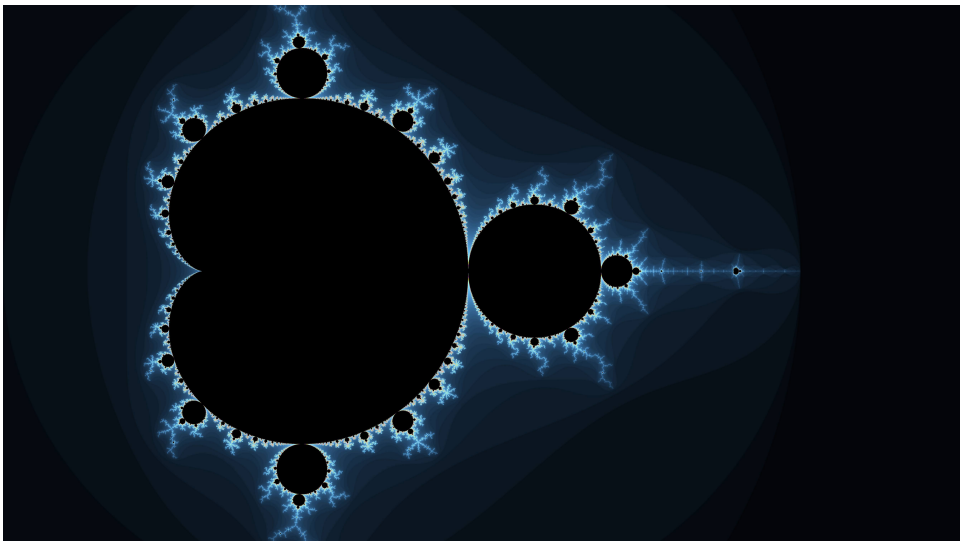

Figure 1: Mandelbrot pattern rendered at $7720x4320$ pixels at a depth of 1024.

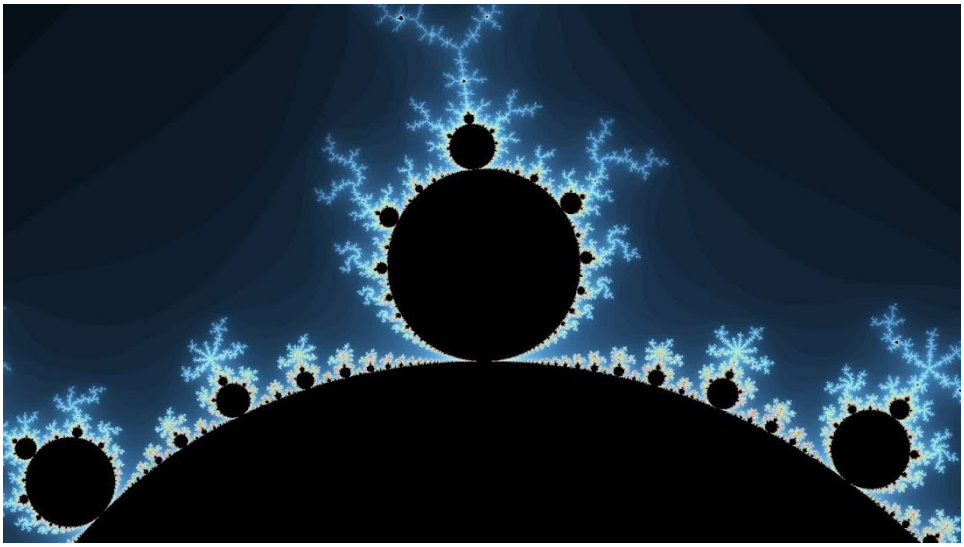

Figure 2: Zoomed in version of the high resolution image.

Discussion & Conclusion

For better results, that is faster rendering, one could for instance improve the program by doing some of the following: Change where some calculations are made. Reduce size of tuples (extremely small improvement). Create a simpler color conversion function. Add an additional language to gain access to CUDA cores.

The improvement achieve with just implementing the parallelism is however good enough since it clearly illustrates how parallelism can be used to reach faster calculations. It also shows that it doesn't have to be complicated and can in fact even be simpler than concurrency as seen in the lab "Philosophers" done earlier in this course.