EXERCISE

The Mandelbrot set

In this exercise, you are required to implement a parallel code that iteratively calculates Eq. (6) for a given section of the complex plane (or, in other words, that computes the Mandelbrot set).

The Mandelbrot set is generated on the complex plane $\mathbb{C}$ by iterating the complex function $f_c(z)$ whose form is

$$f_c(z) = z^2 + c \tag{6}$$

for a complex point $c = x + iy$ and starting from the complex value $z = 0$ so to obtain the series

$$z_0 = 0,\ z_1 = f_c(0),\ z_2 = f_c(z_1),\ \ldots,\ f_c^n(z_{n-1}) \tag{7}$$

The $Mandelbrot\ Set\ \mathcal{M}$ is defined as the set of complex points c for which the above sequence is bounded. It may be proved that once an element i of the series is more distant than 2 from the origin, the series is then unbounded.
Hence, the simple condition to determine whether a point c is in the set $\mathcal{M}$ is the following

$$\left| z_n = f_c^n(0) \right| < 2 \ \text{ or } \ n > I_{max} \tag{8}$$

where I_{max} is a parameter that sets the maximum number of iteration after which you consider the point c to belong to $\mathcal{M}$ (the accuracy of your calculations increases with I_{max}, and so does the computational cost).

Given a portion of the complex plane, included from the bottom left corner $c_L = x_L + iy_L$ and the top right one $c_R = x_R + iy_R$, an image of $\mathcal{M}$, made of $n_x \times n_y$ "pixels" can be obtained deriving, for each point c_i in the plane, the sequence $z_n(c_i)$ to which apply the condition (8), where

$$\begin{aligned} c_i &= (x_L + \Delta x) + i(y_L + \Delta y) \\ \Delta x &= (x_R - x_L)/n_x \\ \Delta y &= (y_R - y_L)/n_y\,. \end{aligned} \tag{9}$$

In practice, you define a 2D matrix `M` of integers, whose entries `[j][i]` correspond to the image's pixels. What pixel of the complex plane $\mathbb{C}$ corresponds to each element of the matrix depends obviously on the parameters $(x_L, y_L), (x_R, y_R), n_x,\ \text{and}\ n_y$.

Then you give to a pixel `[j][i]` either the value of 0, if the corresponding c point belongs to $\mathcal{M}$, or the value n of the iteration for which

$$\left| z_n\left(c\right) \right| > 2 \tag{10}$$

(n will saturate to I_{max}).

This problem is obviously embarrassingly parallel, for each point can be computed independently of each other.

Notes:

you may **directly produce an image file** using the very simple format `.pgm` that contains a grey-scale image. You find a function to do that, and the relative simple usage instructions, in in the followings.
In this way you may check in real time and by eye whether the output of your code is meaningful.

> **Note 1:** Mandelbrot set lives roughly in the circular region centered on $(-0.75, 0)$ with a radius of ~ 2.

> **Note 2:** the multiplication of 2 complex numbers is defined as
> $(x_1 + iy_1) \times (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$

Writing a `PGM` image

The `PGM` image format, companion of the `PBM` and `PPM` formats, is a quite simple and portable one. It consists in a small header, written in ASCII, and in the pixels that compose the image written all one after the others as integer values. A pixel's value in `PGM` corresponds to the grey level of the pixel.
Even if also the pixels can be written in ASCII format, we encourage the usage of a binary format.

The header is a string that can be formatted like the following:

`printf( "P5\n%d %d\n%d\n", width, height, maximum_value );`

where "`P5`" is a magic number, `width` and `heigth` are the dimensions of the image in pixels, and `maximum_value` is a value smaller than `65536`.
If `maximum_value < 256`, then 1 byte is sufficient to represent all the possible values and each pixel will be stored as 1 byte. Instead, if `256 <= maximum_value < 65536`, 2 bytes are needed to represent each pixel (that is why in the description of Exercise 1 we asked you that the matrix `M` entries should be either of type `char` or of type `short int`).

In the sample file `write_pgm_image.c` that you find the `Assignment03` folder, there is the function `write_pgm_image()` that you can use to write such a file once you have the matrix `M`.

In the same file, there is a sample code that generate a square image and write it using the `write_pgm_image()` function.
It generates a vertical gradient of N_x x N_y pixels, where N_x and N_y are parameters. Whether the image is made by single-byte or 2-bytes pixels is decided by the maximum colour, which is also a parameter.
The usage of the code is as follows

```
cc -o write_pgm_image write_pgm_image.c
./write_pgm_image [ max_val] [ width height]
```

as output you will find the image `image.pgm` which should be easily rendered by any decent visualizer .

Once you have calculated the matrix `M`, to give it as an input to the function `write_pgm_image()` should definitely be straightforward.