Assignment 1a: "Mandelbrot"

Given is a C++ program that generates the Mandelbrot set image and saves it to disk in a "PPM" format.

The resulting image (see Figure 1) should display the Mandelbrot set for the given resolution, where pixels within the set are represented in black* (RGB: 0, 0, 0) and those outside in white (RGB: 255, 255, 255).

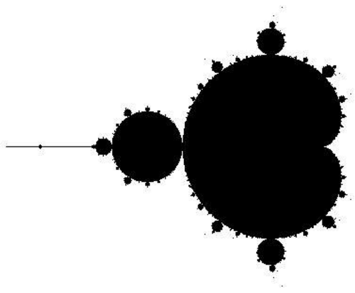

Figure 1

A nice and interactive description of how the Mandelbrot set is generated can be found at this link: https://complex-analysis.com/content/mandelbrot_set.html. The algorithm is also explained later in this document."

Requirements summary:

- Use **C++ Multithreading** to develop a parallel version of the Mandelbrot program (Mandelbrot.hpp)
- The resulting image should display the Mandelbrot set in a 512x384 resolution, which is provided to the program via input arguments.
- Implement the required features according to the specification below.
- Achieve a speedup of 13+

Color Representation:

- Pixels belonging to the Mandelbrot set should be displayed in black (RGB: 0, 0, 0).
- Pixels not part of the Mandelbrot set should be displayed in white (RGB: 255, 255, 255).

Implementation Details:

Your parallel implementation should be placed in the Mandelbrot.hpp file. All extensions and changes should be made inside this class.

Data Handling:

- Image data is stored in a one-dimensional array. However, elements can be accessed via image[x][y], where x is a row and y is a column. This will return a "Pixel".
- Image and Pixel structs have all fields and methods set to public for ease of usage and testing on the Moped.

*Note: We represent pixels in black for the sequential version of the code, and we only mention the black color in the descriptions. Parallel versions use a simple formula to compute a different grayscale value for each thread.

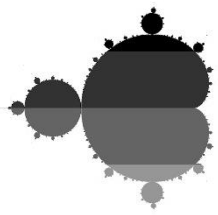

Figure 2

Detailed Description

1. Mandelbrot [implement in mandelbrot.hpp]

On construction, this class creates a new Image data structure with the input dimensions (width and height) and sets all pixel values to {0, 0, 0} (black). Optionally, it can change the value of the maximum iterations used for the Mandelbrot kernel. The member function interfaces should not be changed.

Afterward, it iterates over all pixels in this image. For each pixel, it maps the pixel coordinates (x, y) to the corresponding complex number (cx, cy) in the complex plane using the following formula:

Extend/implement following functions:

1. `int compute(int num_threads = 1)`

   This function **creates num_threads threads**, all of which call the worker function. This member function needs to allocate work for the given number of threads and makes sure that spawned threads have completed. All functionality related to multithreading has to be contained inside this class. The function returns the number of pixels inside the Mandelbrot set.

2. `int worker(int num_threads, int thread_id=0)`

   This function iterates over all pixels in the image. For each pixel, it maps the pixel coordinates (x, y) to the corresponding complex number (cx, cy) in the complex plane, using the following formula:

```
double dx = ((double)x / image.width - 0.75) * 2.0;
double dy = ((double)y / image.height - 0.5) * 2.0;

std::complex<double> c(dx, dy);
```

Here is a pseudo algorithm for this function:

```
For each pixel (x, y) in the image:
    Map the pixel coordinates (x, y) to the corresponding complex number (c_x, c_y) in the complex plane
        where the real part c_x ranges from -2.5 to 1 and the imaginary part c_y ranges from -1 to 1.

    Call the check_pixel function

    If the check_pixel function returns "false" value:
        The pixel (x, y) is not in the Mandelbrot set.
        Assign a color representing that it is not within the set (white - RGB: 255, 255, 255).

    If the check_pixel function return "true":
        Assign a color representing that it diverges (black - RGB: 0, 0, 0).
```

Note that this function calls the `check_pixel` function, which determines whether the pixel belongs to the Mandelbrot set. If the `check_pixel` function returns "`false`" value, we set the color of that pixel to white; otherwise, we set it to black, indicating it is in the Mandelbrot set.

Note also that this function computes a color for pixels that are in the set.

```
unsigned char color = (255*(thread_id+1))/num_threads;
```

Each thread will be assigned a different grayscale value, which will then be visible on the output image. Since each color channel gets the same value, the output will be a grayscale image.

`bool check_pixel(std::complex<double> c)` (already implemented)

For each given complex number, this function computes a complex number **z** by incrementing it from its initial zero value. The number is incremented using the following formula:

$$z_{n+1} = z_n^2 + c$$

Where C is the input value provided by the `worker` function.

Here is the pseudo code for the `check_pixel` function:

```
    Set initial values:
        z = 0 + 0i
        iteration = 0

    While iteration < max_iterations:
        Calculate the new value of z using the Mandelbrot formula:
            z = z^2 + c

        If the magnitude (absolute value) squared of z becomes greater than 4:
            The pixel (x, y) is not in the Mandelbrot set.

            Break out of the loop / return false

        Increment the iteration counter.

    If the loop completes without z becoming greater than 4:
        The pixel (x, y) is in the Mandelbrot set.
        Return true
```

Note that the **max_iterations** value is set to **2048**, unless changed via command line arguments.

The check_pixel function returns a boolean value. If a pixel does not belong to the Mandelbrot set, this function returns false, otherwise it returns true, signifying that the pixel is in the Mandelbrot set.

2. "`Pixel`" struct [already implemented in pixel.hpp]

The Pixel struct is used to store information about a single pixel, denoted by the values for red, green, and blue colors (RGB) in the [0-255] range. Therefore, the values for the colors should be represented using unsigned 8-bit integers. The following fields should be created:

- **r** – unsigned integer allowing [0-255] range representing the value for the red color channel.
- **g** – unsigned integer allowing [0-255] range representing the value for the green color channel.
- **b** – unsigned integer allowing [0-255] range representing the value for the blue color channel.

The `Pixel` struct can be printed, and the output shows r, g and b values separated by a comma and a single space `', '`, e.g., a white pixel would be printed as:

255, 255, 255

Code examples:

```
let p1 = Pixel{255, 0, 0}; // create a pixel with red channel set to 255
cout << p1; // prints "255 0 0"
```

3. "Image" struct [already implemented in image.rs]

The Image struct is used to allow easier traversal through matrix and the vector holding its data values. The following fields should be created:

- `width` – an unsigned integer value.
- `height` – an unsigned integer value.
- `data` – a vector holding the pixel data of the image.

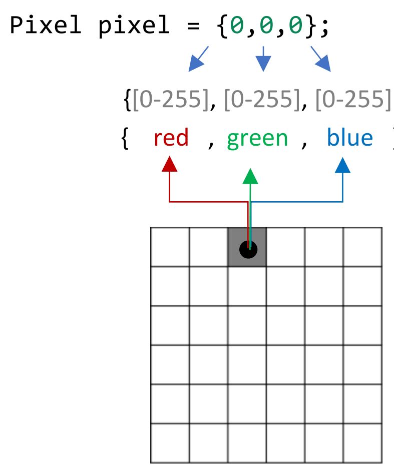

Image data structure

Code examples:

```
    Image image(200,200); // new 200x200 image

    // set the value of the element at (0,3)
    image[0][3] = {255, 255, 255};
```

4. Parsing input [already implemented in helper.hpp]

Your program should accept the following three parameters:

- **num-threads** – an integer representing the desired number of threads.
- **max-iterations** – an integer representing the desired maximum number of iterations for the kernel.
- **height** – an integer representing the width of the image (required argument).
- **width** – an integer representing the width of the image (required argument).

5. Saving files

```
void save_to_ppm(std::string filename)
```

This function saves the data to .ppm format, it has already been implemented and is called from the main.

6. Expected Behavior

Assuming your executable is called “mandelbrot”, your code will be executed with the following input parameters:

```
./mandelbrot
```

The expected output is the following:

```
Generating Mandelbrot for 512x384 image (max_iterations: 2048)
Total Mandelbrot pixels: 74118
2.52454
```

Your program should not produce any warnings, no additional info about bad input parameters, or other print outs. The output and test results should achieve a match on the online platform. The last line represents the execution time of the program.

To compile at home use:

- `g++ -o mandelbrot -std=c++20 -lpthread -O2 main.cpp`

To compile on ALMA use:

- `/opt/global/gcc-11.2.0/bin/g++ -o mandelbrot -std=c++20 -lpthread -O2 main.cpp`

To run on ALMA use:

- `srun --nodes=1 ./mandelbrot --num-threads 16`

.

7. Submission Guidelines

A working program and with speedup measurements have to be submitted before the deadline on the online platform.

Additionally, you need to run your code on ALMA, measure speedup. The required speedup on ALMA is 13+ and you should measure speedup for 2, 4, 8, 16 and 32 threads (not on the frontend, but by using **srun** as shown above). The sequential execution time on Alma is around **2.57** seconds. Once you have the results, enter them on the online platform (the speedup.graph entry). The online platform should not be used for speedup measurements.

Deadline: Wednesday, 20.4.2024 23:59 on the online platform.