# Introduction

This is a Python program that calculates the Mandelbrot set using a client-server architecture and produces a PGM image depicting the resulting fractal.

# Execution

A server is started using the following arguments:

```
python server.py [IP_adress] [Port]
```

Where "IP_adress" is the adress the server should use and "Port" is the port it should listen on.

The client uses the following arguments:

```
python client.py [min_c_re] [min_c_im] [max_c_re] [max_c_im]
        [max_n] [x] [y] [division] [List of servers]
```

All the arguments up to "max_n" are parameters for the mandelbrot calculations. "x" and "y" compose the size of the resulting image (and thus the amount of detail), while "division" dictates how many vertical chunks the workload should be divided into before being distributed among the servers. As such, "y" needs to be evenly divisible by "division". "List of servers" is a list of servers the client should connect to and send work to.

Example run of the program, where each line is run on a different terminal, either on the same computer or on different ones:

```
python server.py localhost 3333

python server.py localhost 4444

python client.py -1 -1.5 2 1.5 300 1000 1000 5 localhost:3333
                        localhost:4444
```

This divides the 1000*1000 pixels image into 5 vertical 200 pixel chunks that get divided among the two servers. The servers then work through one chunk at a time (with one server getting 3 chunks while the other gets 2), sending the resulting data back to the client when ready. It is then written to the resulting image by the client. Note that all servers must be started before the client so that they can connect properly.

# Workings

## Client

Using its input parameters, the client creates descriptions of work and then allocates it to as many threads as there are servers. Each thread connects to its appropriate server and sends a message containing information on a chunk of work and waits for the server to send back the resulting image information. Said information is then put in the correct index in a list shared by all threads. This process repeats until all the work is done, at which point the thread sends a termination message to the server and concludes itself. Once all the threads have been joined by the master thread, the result list is written into a PGM-file, producing an image of the fractals.

## Server

A server sits and listens on its specified port until a client thread connects. Then, it will parse a request message and either calculate the requested data or terminate, depending on the message. Once calculations are complete, the data will be sent to the client thread and the server will wait until the next message.

# Tests

Having developed the program on a laptop, my resources for testing are limited. I've simply run everything locally on the same machine, which severly increases the time it takes to create complex images, since the work isn't actually distributed between computers. Even so, the program works as evident by the pictures created in the following two tests (Image 1 & Image 2):

```
python client.py -1 -1.5 2 1.5 300 5000 5000 10 localhost:3333
                                                localhost:4444
```

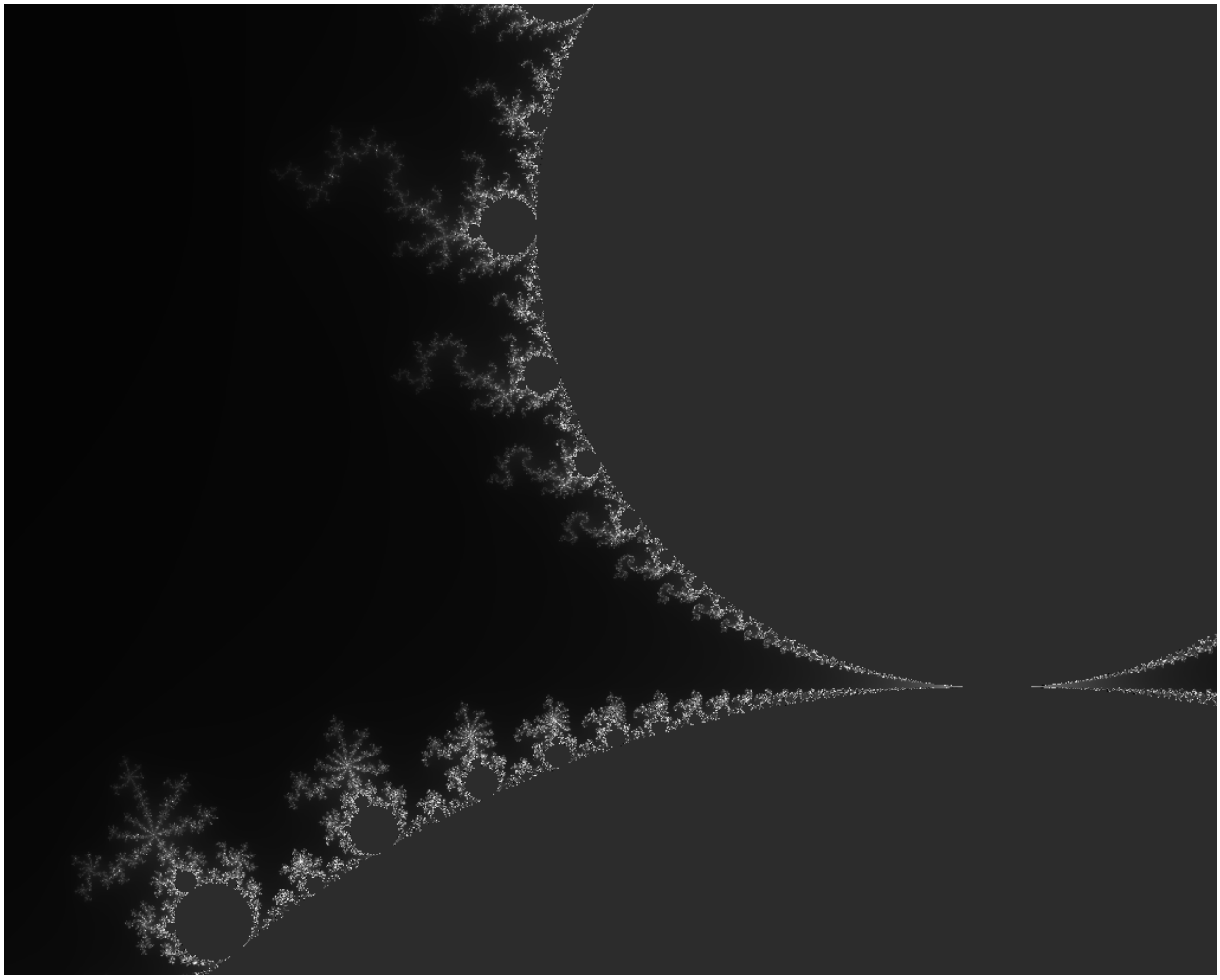

**Image 1: A zoomed in view of the resulting 5000*5000 image showcasing the smaller fractals in the top left of the image.**

```
python client.py -1 -1.5 2 1.5 300 1000 1000 10 localhost:3333
```

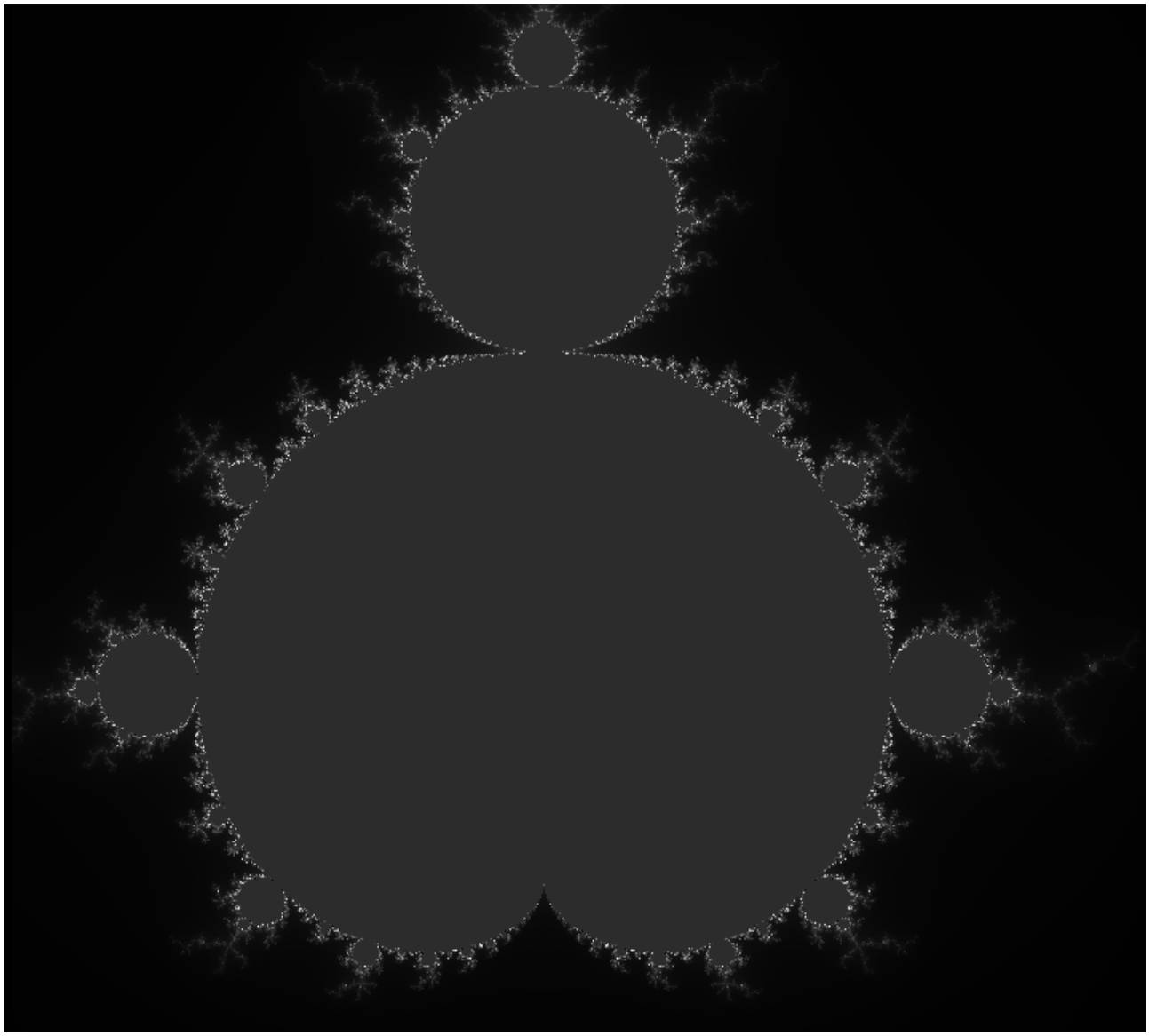

**Image 2: A full view of the resulting 1000*1000 image showcasing all of the fractals in lower resolution than in Image 2.**

# Discussion

I am pretty satisfied with the result of my efforts, as I am certain it will scale with an increase in available computers. But as I've worked on it, I've compiled some things that should be improved before turning this architecture into an actual service:

There is no logic put in place to safeguard from connection problems. It is always assumed that the provided server names are correct and that the connections don't fail. This is obviously not viable outside of my local tests, so there should be logic implemented to check that server names are correct and make it possible for the client to reconnect to a server. Should the connection drop, I'm pretty sure a server will either crash or get stuck in a loop.

Being written in Python as an excuse for me to practice the language more, the Mandelbrot calculations are very inefficient. I saw a few ways to actually compile that portion of the code using some external libraries to speed it up, but I decided against implementing them for the sake of portability and simplicity. Python is not fit for heavy-duty calculations, so even if one is determined to keep the majority of my solution, the actual Mandelbrot computing should be done in a different language.

As seen earlier in the execution instructions, the work is split on rows of image data and not squares. I tried to divide the work into squares, but I ran into problems splitting matrices, iterating over them and then recombining them. The program still meets the overarching goal of splitting up the work, but as-is, it's not possible to do it as finely as requested.

Lastly, a server is single-threaded, meaning that one process can only connect to a single client-thread. For an actual service, the server should be able to listen for multiple connections and allocate a thread for each one.