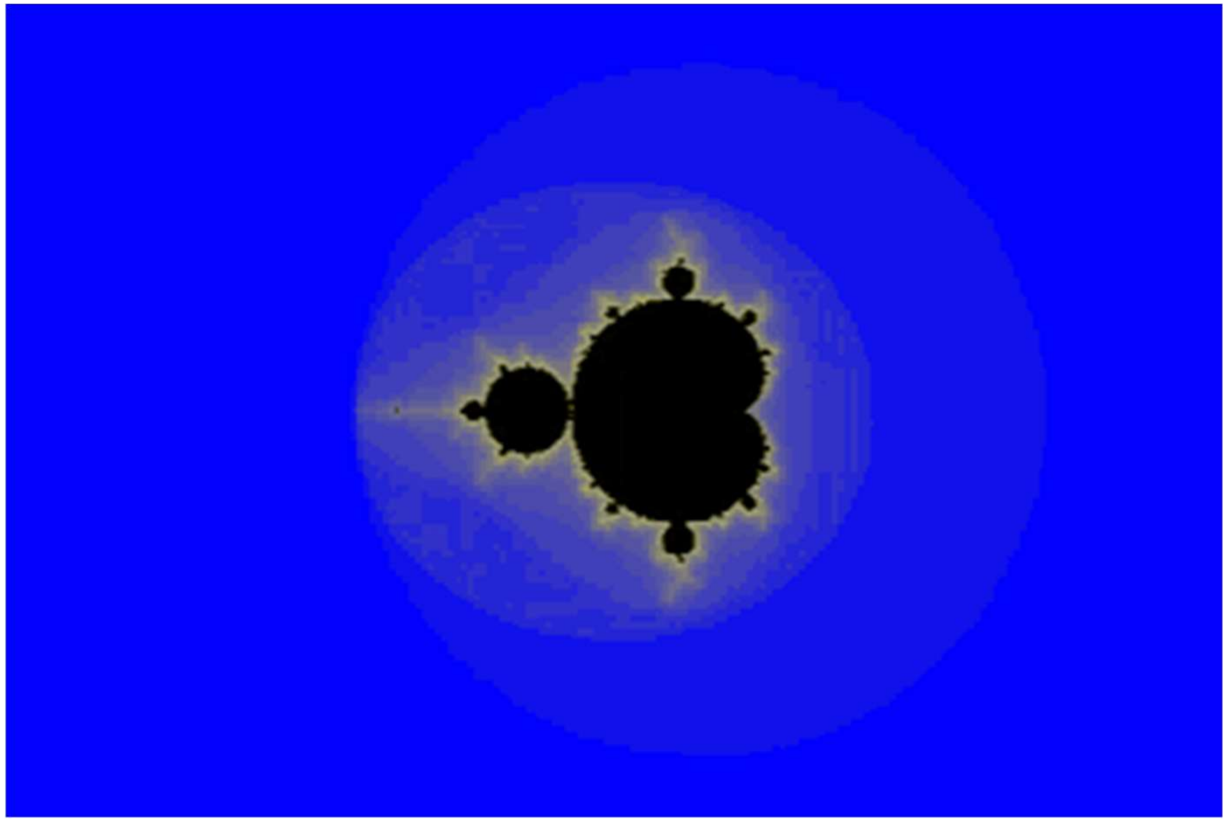
Parallelising the Mandelbrot algorithm (CPU)

Tia C[redacted] – [redacted] – CMP202

What’s Mandelbrot?

- Mandelbrot is an algorithm that creates a symmetrical *geometric fractal*
- Zooming into the created image will eventually show the set repeating itself.
- The equation for this algorithm is incredibly complex, which means it requires a lot of processing power to be calculated and rendered.



What’s the problem?

- Mandelbrot requires a lot of processing power…
- You can use a CPU or a GPU to calculate and render the algorithm/image.
- But it takes a long time even with powerful hardware…
- So how do we make it more efficient?

First of all, we need to see how long the algorithm takes to run normally.

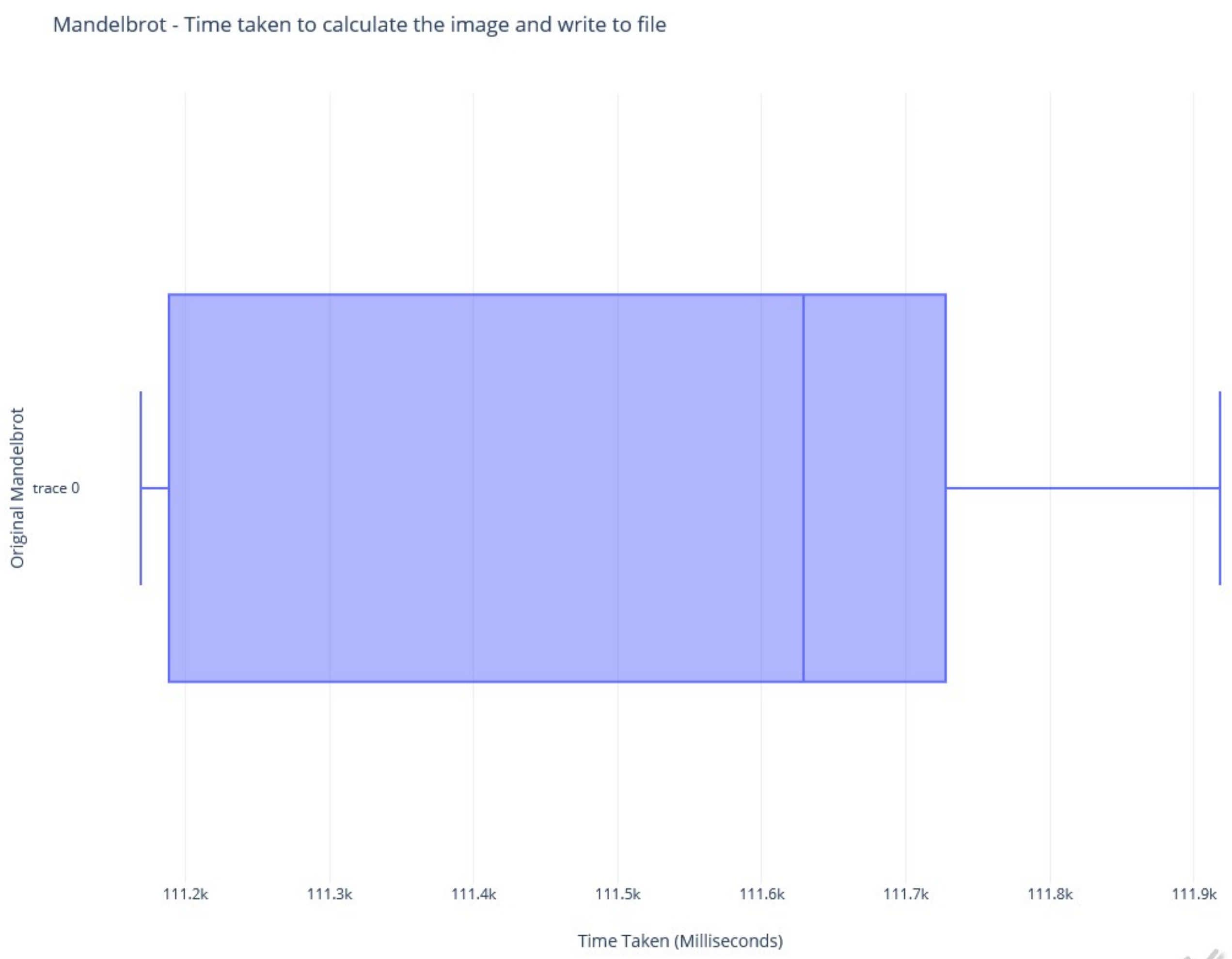
For testing purposes...

The Mandelbrot set has been set to the colours, Orange and Purple and the set co-ordinates, image size and iteration are all the same.

The clock is started as the Mandelbrot set is calculated and is stopped once the image has been written to the file.

To ensure there is enough data to give an accurate result of time and efficiency, the program was ran a total of five times

Original Mandelbrot program



How can we make it more efficient?

Instead of calculating and rendering the whole image…

Why not, split it into sections instead?

Instead of trying to render the whole image, it renders a section at a time instead!

Mandelbrot – Image split up in to sections

Image has been split into sections

Sections are calculated separately, then written to file

Means the overall time taken is reduced

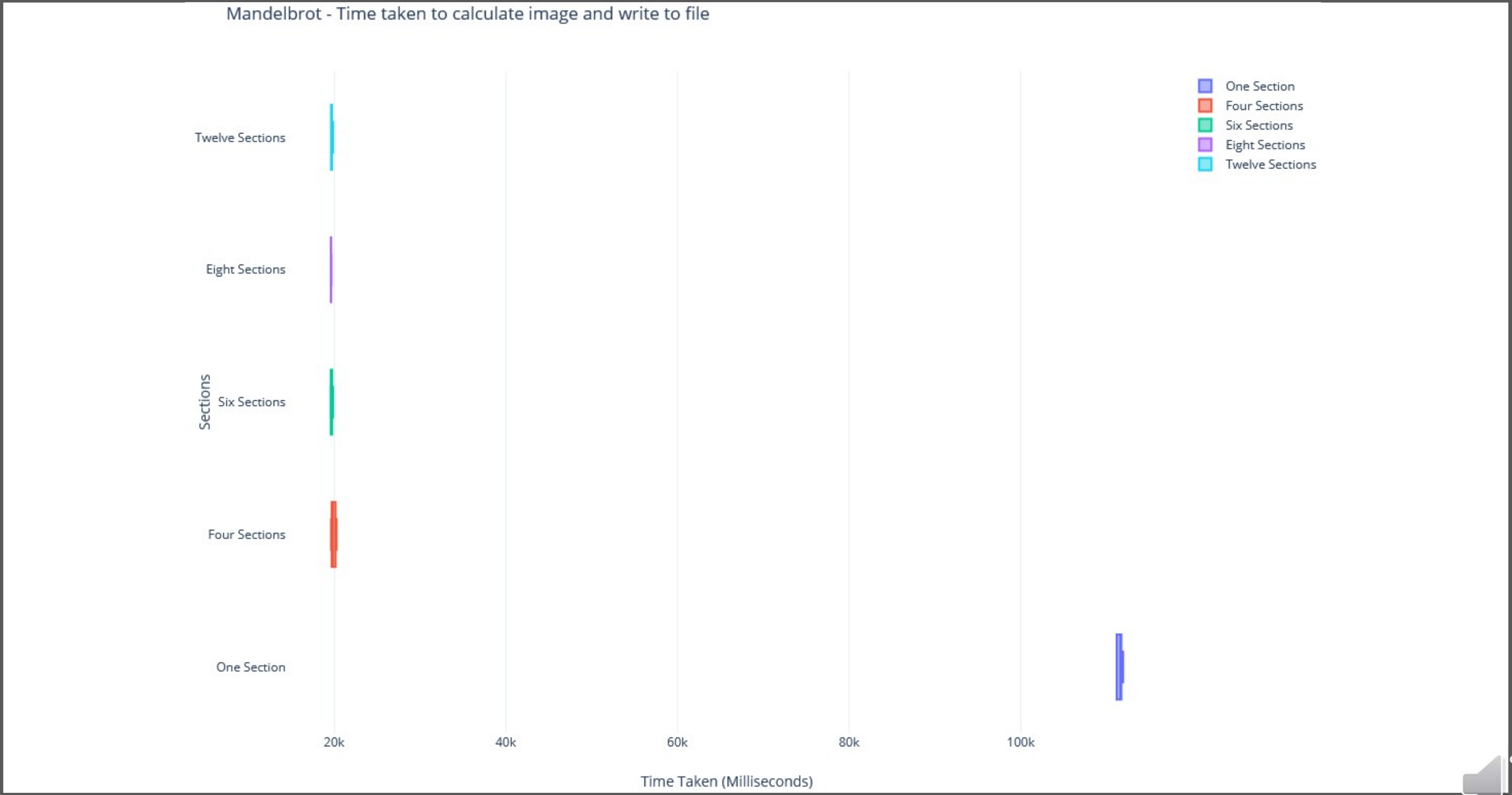
For testing purposes...

The Mandelbrot set has been set to the colours, Orange and Purple and the set co-ordinates, image size and iteration are all the same.

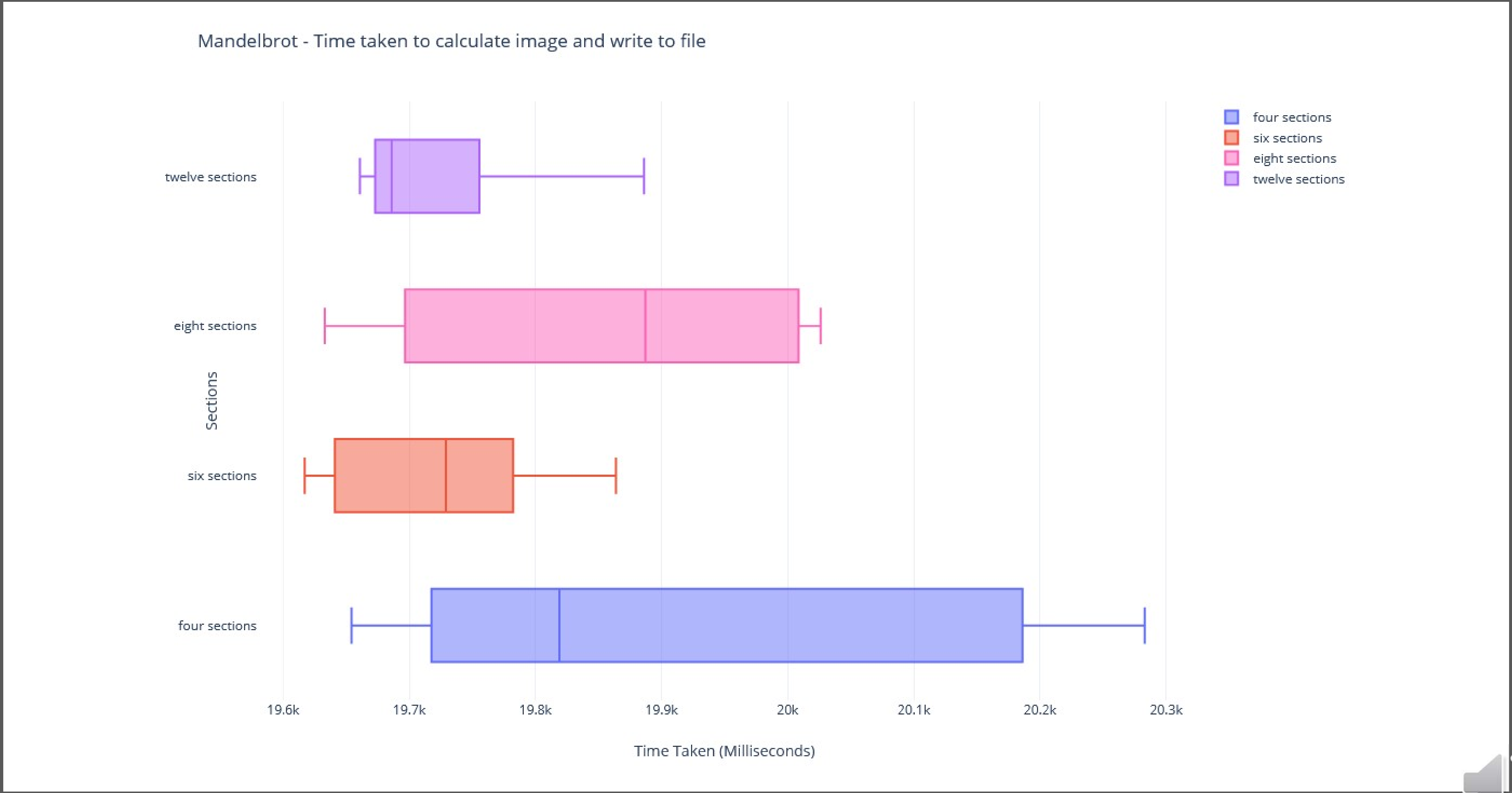
The clock is started as the Mandelbrot set is calculated and is stopped once the image has been written to the file.

The program is ran five times in total, this ensures enough data to give an accurate result of time and efficiency.

The image is split into one, four, six, eight and twelve sections respectively to get an accurate insight as to how efficient splitting the image is.

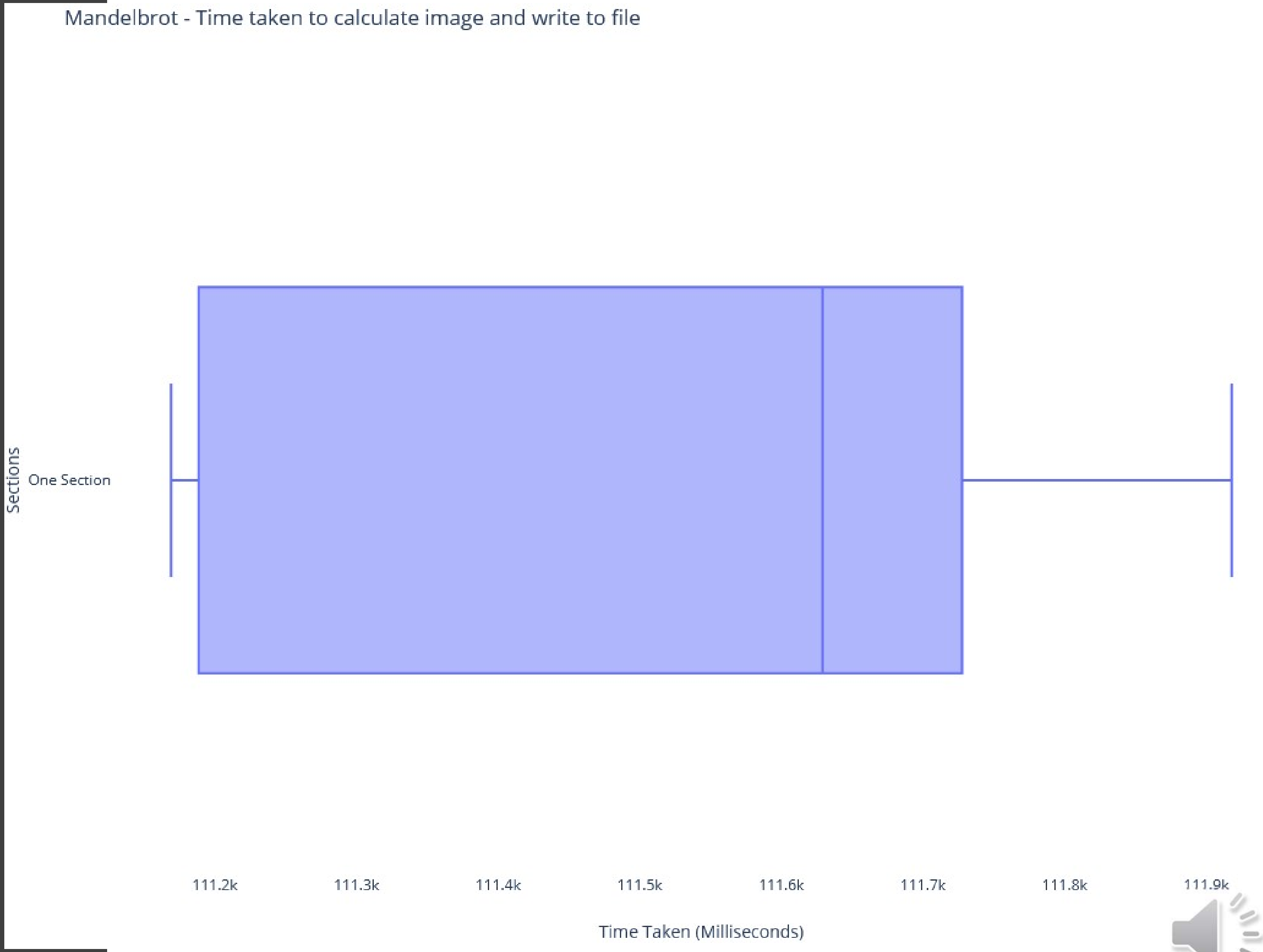


Mandelbrot - Time taken to calculate image and write to file
Sections
Twelve Sections
Eight Sections
Six Sections
Four Sections
One Section
20k
40k
60k
80k
100k
Time Taken (Milliseconds)
One Section
Four Sections
Six Sections
Eight Sections
Twelve Sections


Mandelbrot - Time taken to calculate image and write to file
twelve sections
eight sections
six sections
four sections
Sections
19.6k
19.7k
19.8k
19.9k
20k
20.1k
20.2k
20.3k
Time Taken (Milliseconds)
four sections
six sections
eight sections
twelve sections

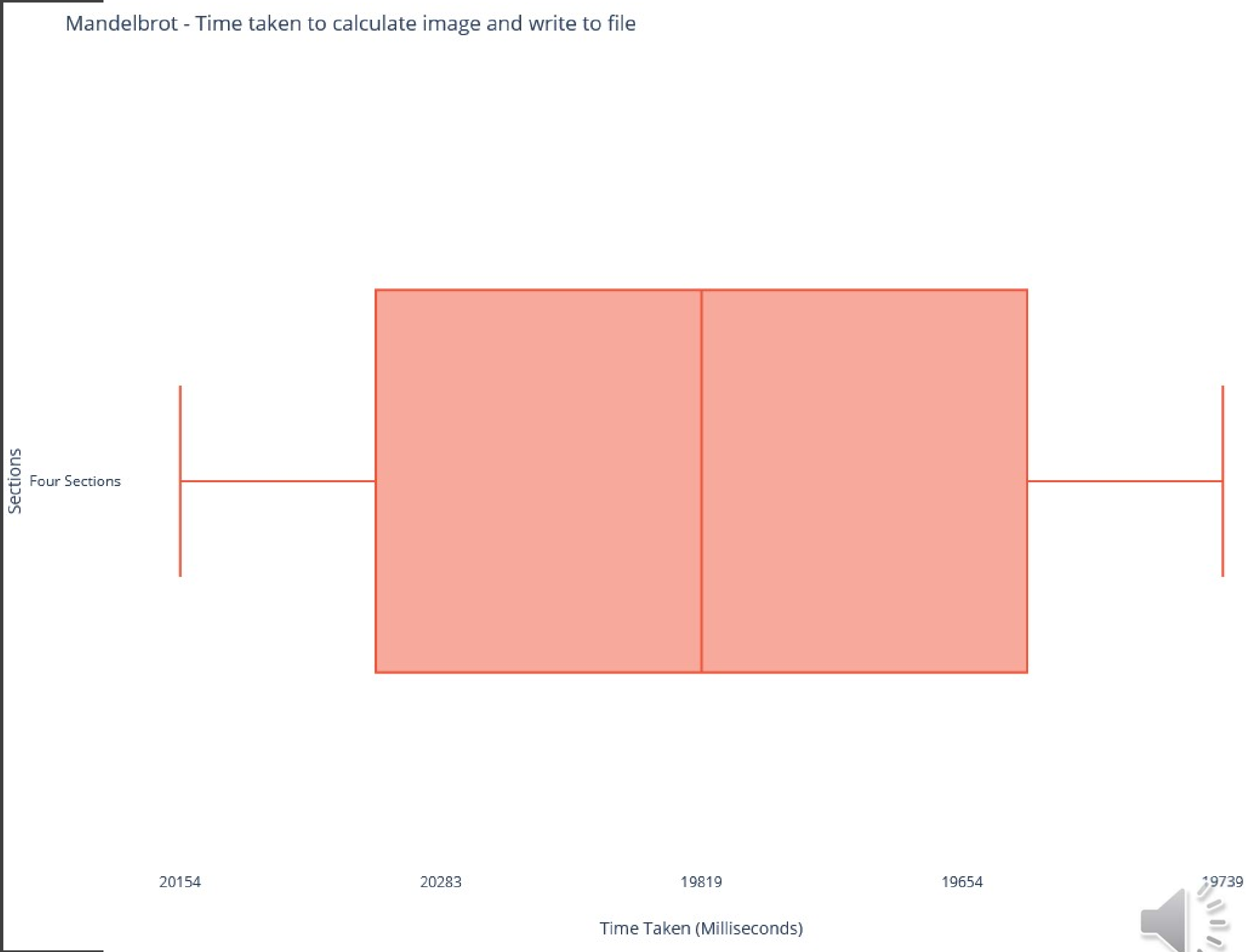
One Section Timings –

- 111918ms
- 111664ms
- 111169ms
- 111629ms
- 111195ms



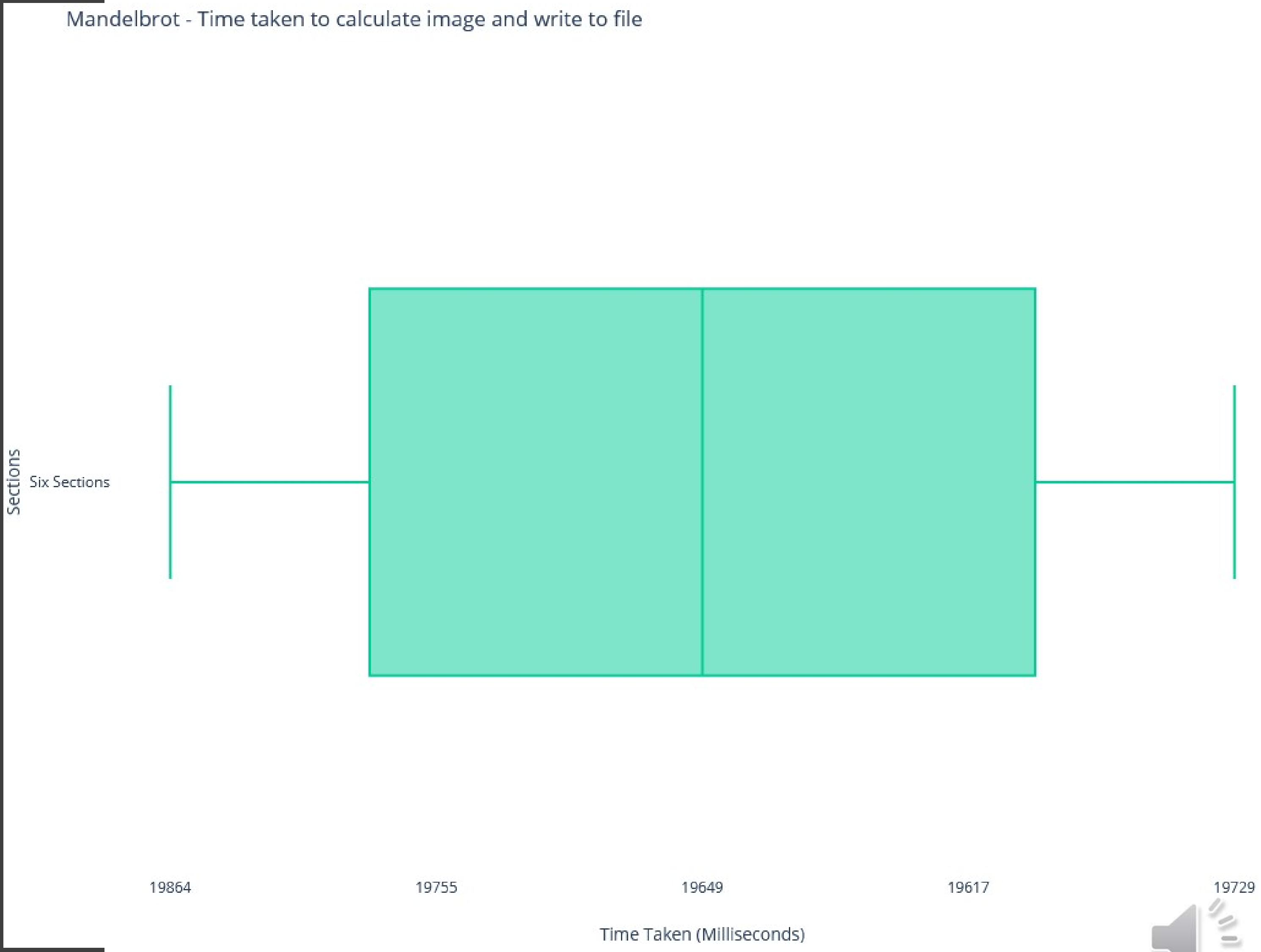
Four Sections Timings –

- 20154ms
- 20283ms
- 19819ms
- 19654ms
- 19734ms



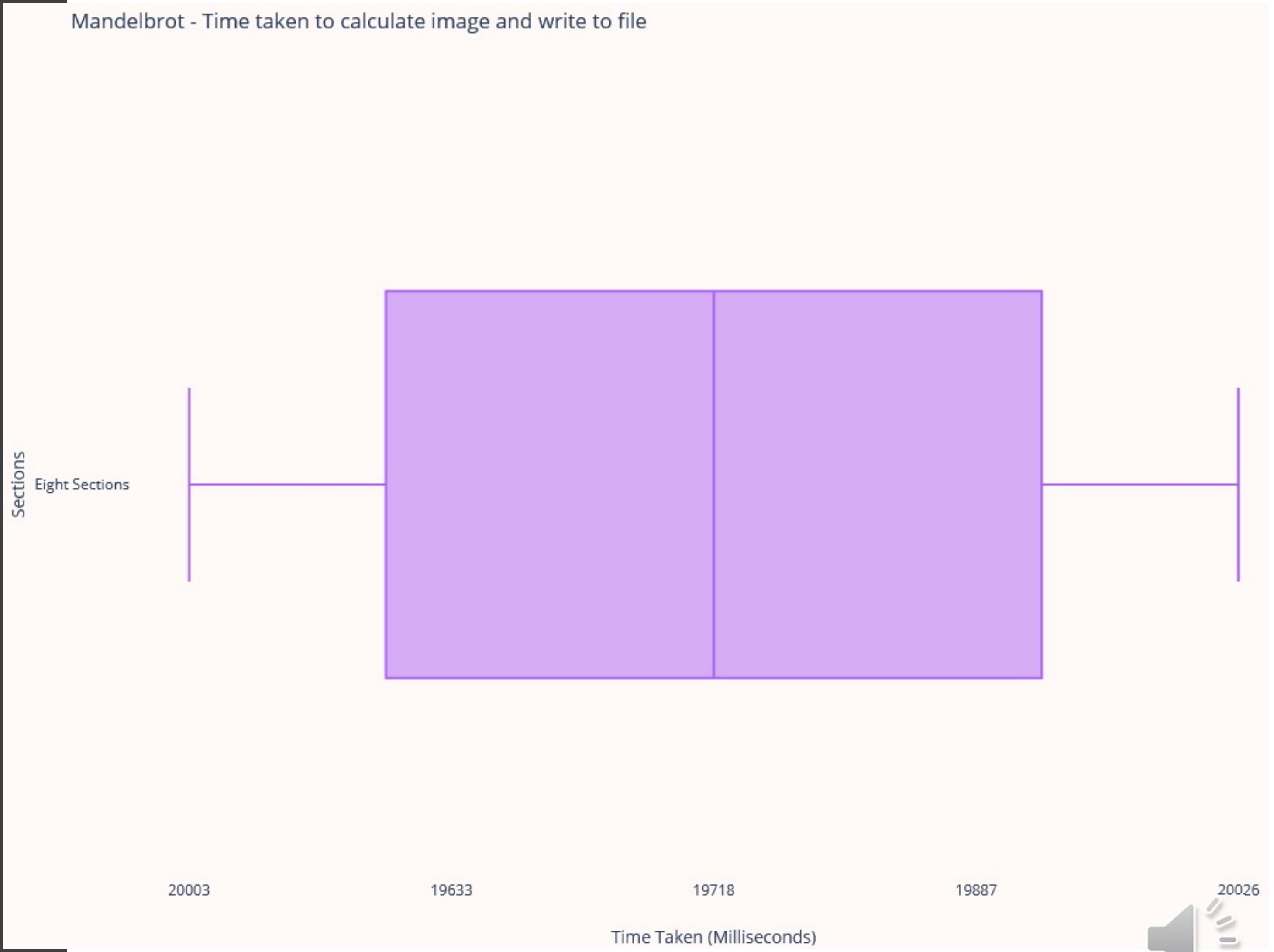
Six Sections Timings –

- 19864ms
- 19755ms
- 19649ms
- 19617ms
- 19729ms



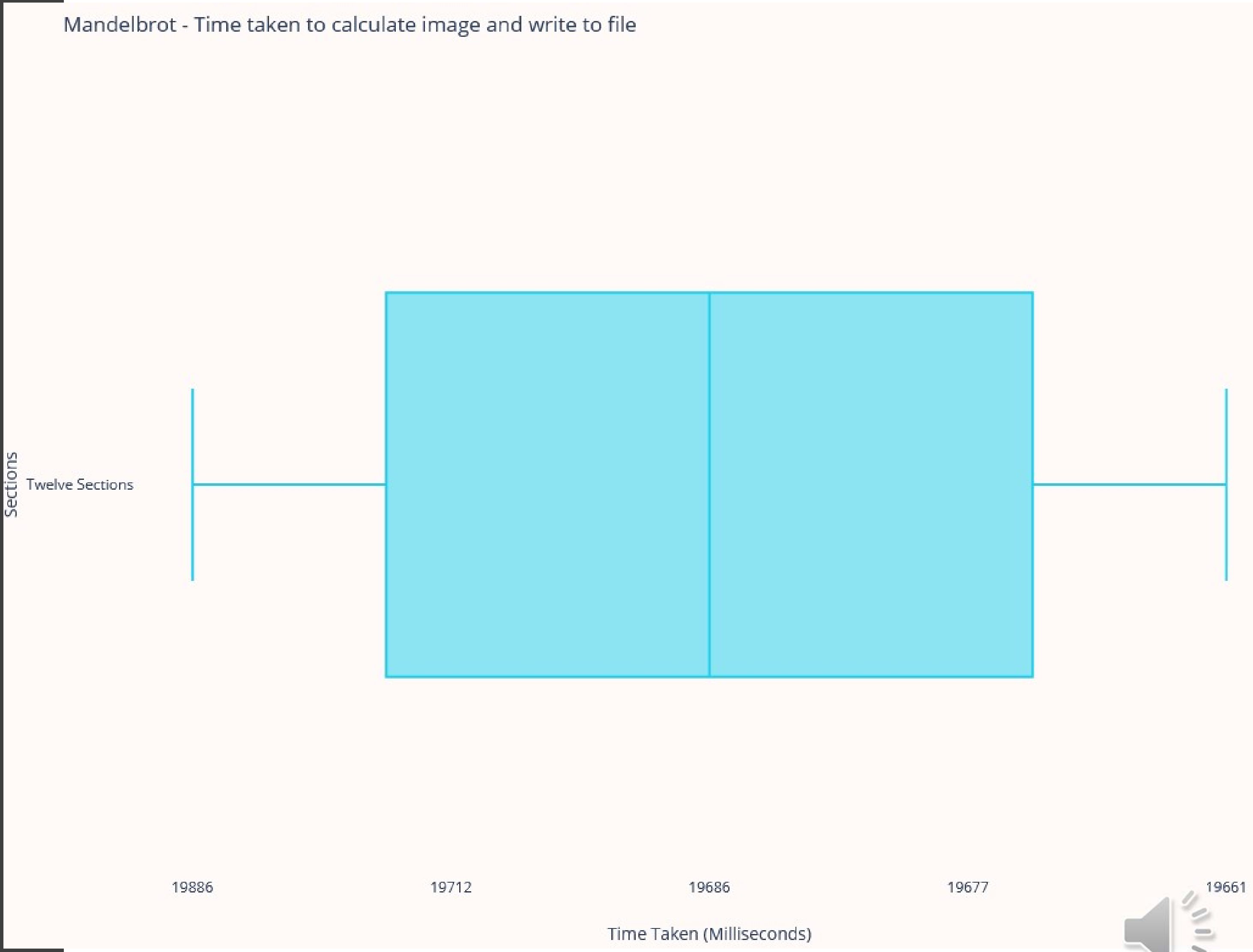
Eight Sections Timings –

- 20003ms
- 19633ms
- 19718ms
- 19887ms
- 20026ms



Twelve Sections Timings –

- 19886ms
- 19712ms
- 19686ms
- 19677ms
- 19661ms



How much faster is splitting the image?

- Four – 82% faster *(91810ms)*
- Six – 82% faster *(91100ms)*
- Eight – 82% faster *(91742ms)*
- Twelve – 82% faster *(91943ms)*

It’s still a bit too slow…

To make the process even more efficient, we should make full use of the equipment available.

We will create a ‘threaded’ version of the program using the CPU.

Mandelbrot – Threaded program

Using threads to calculate and render the image rather than splitting into sections

Very similar to splitting into sections, however the sections will be calculated and rendered at the same time.

What does the threaded program do?

Calculates and displays the number of threads that can run on the machine.

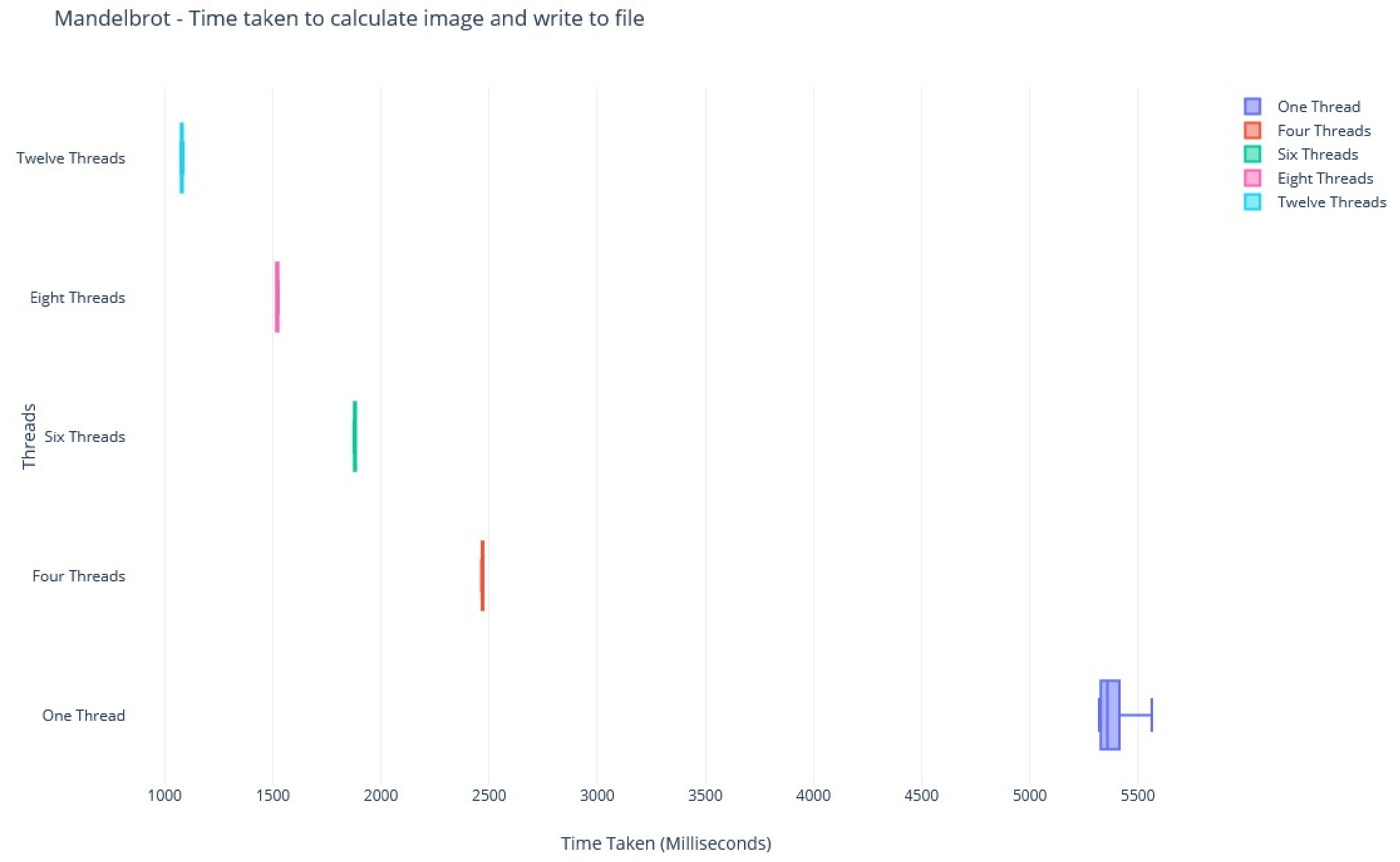
Allows the user to select the number of threads they would like to run.

Allows the user to select two colours that they would like to create the image with.

Calculates and generates the image, then writes to a .tga file.

For testing purposes…

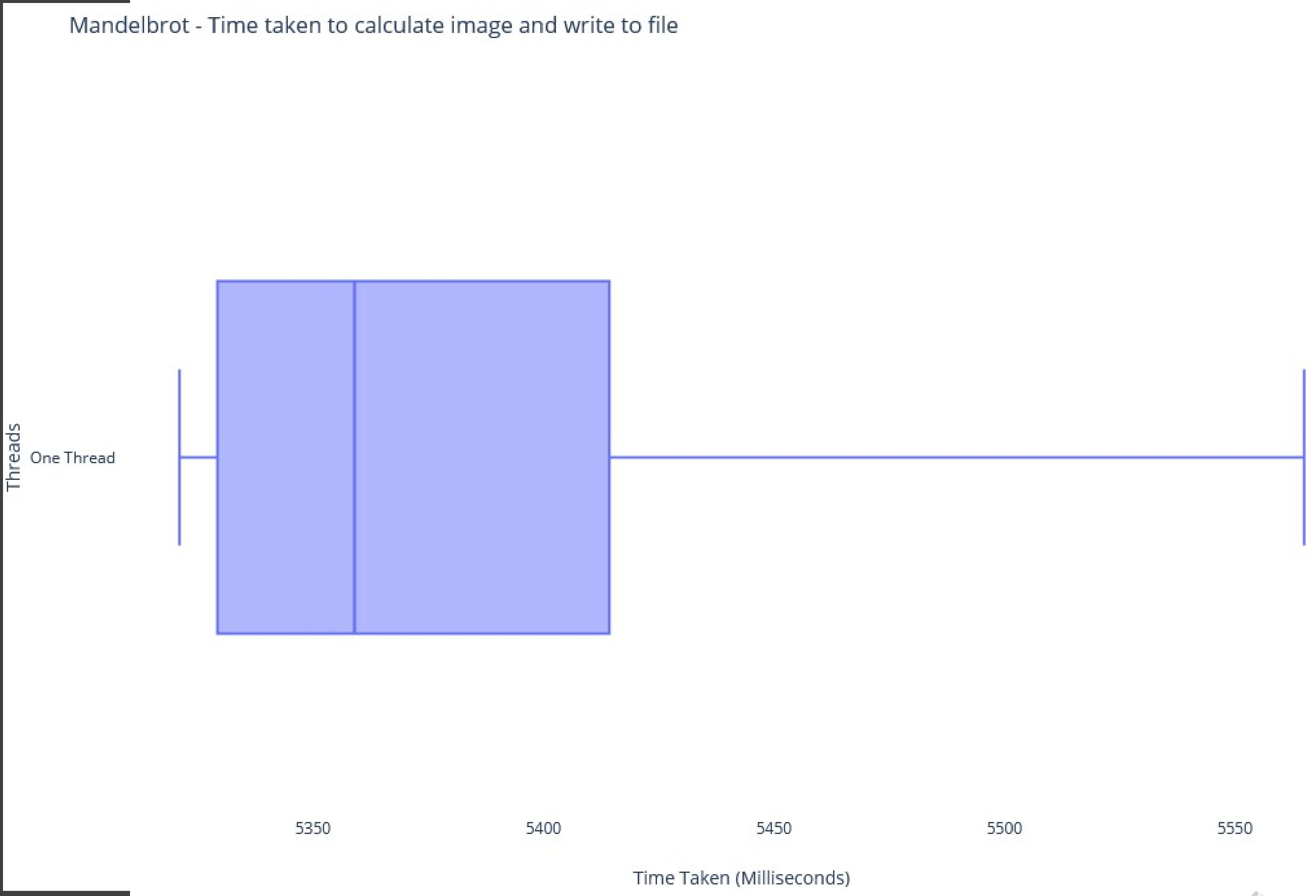
- The Mandelbrot set has been set to the colours, Orange and Purple and the set co-ordinates, image size and iteration are all the same.
- The clock is started as the Mandelbrot set is calculated and is stopped once the image has been written to the file.
- The program is ran five times in total, this ensures enough data to give an accurate result of time and efficiency.
- The program will be run with one, four, six, eight and twelve threads respectively to get an accurate insight as to how efficient threading the program is.
- The CPU in the testing machine is an AMD Ryzen 5 2600 Six-Core Processer capable of running 12 threads concurrently.



Mandelbrot - Time taken to calculate image and write to file
Threads
Twelve Threads
Eight Threads
Six Threads
Four Threads
One Thread
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
Time Taken (Milliseconds)
One Thread
Four Threads
Six Threads
Eight Threads
Twelve Threads

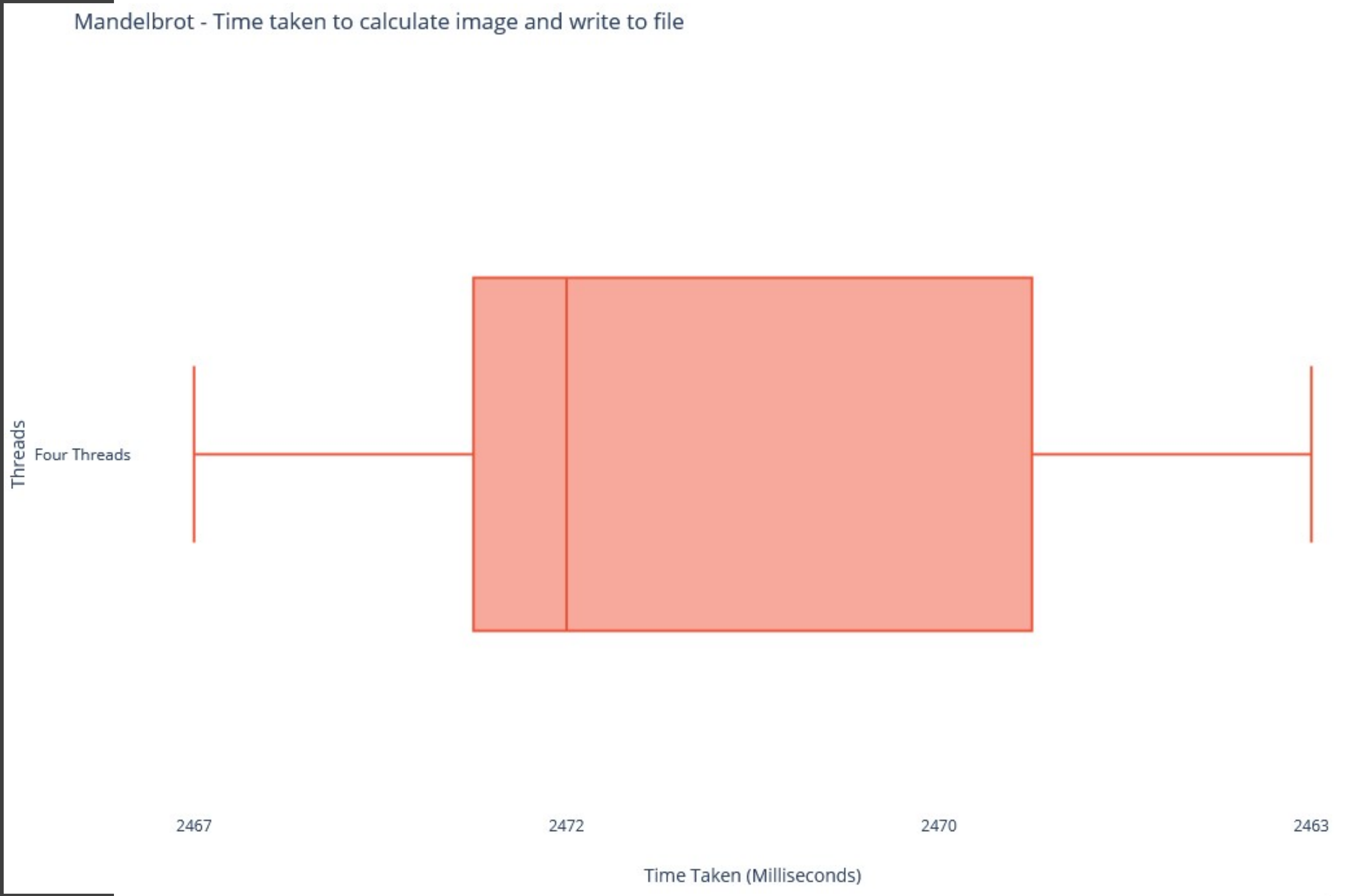
One Thread Timings –

- 5595ms
- 5359ms
- 5364ms
- 5332ms
- 5321ms



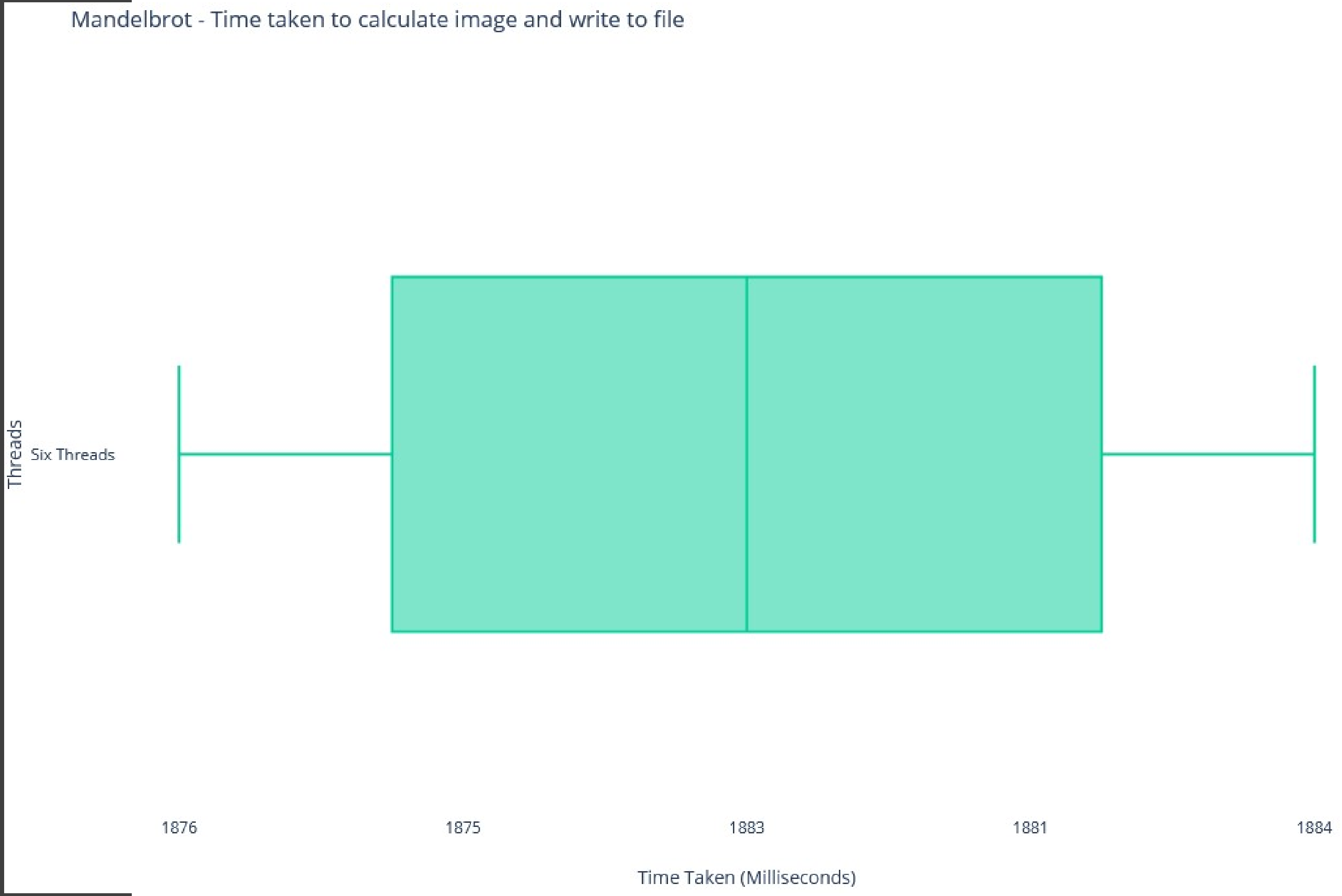
Four Threads Timings –

- 2467ms
- 2472ms
- 2470ms
- 2472ms
- 2463ms



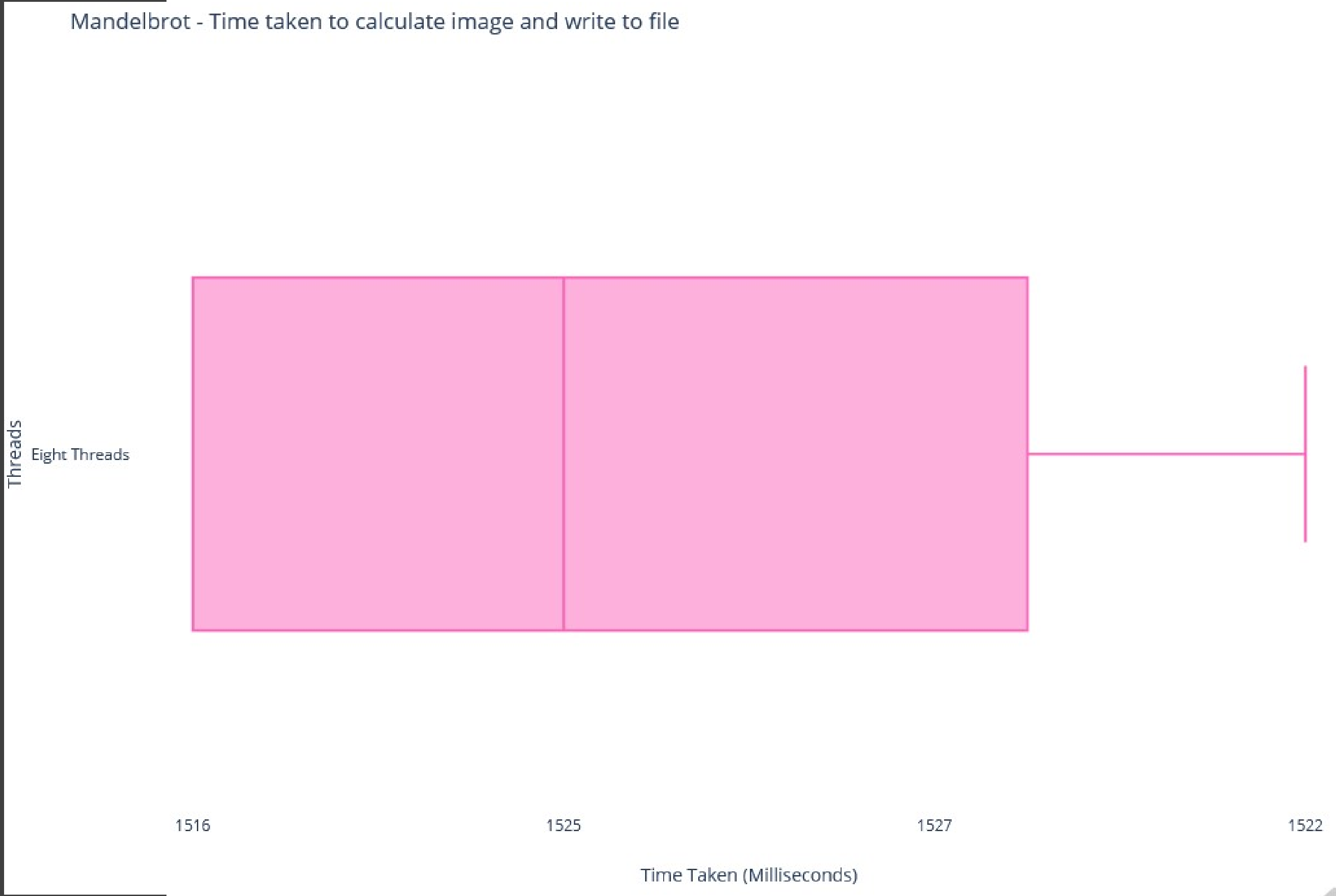
Six Threads Timings –

- 1876ms
- 1875ms
- 1883ms
- 1881ms
- 1884ms



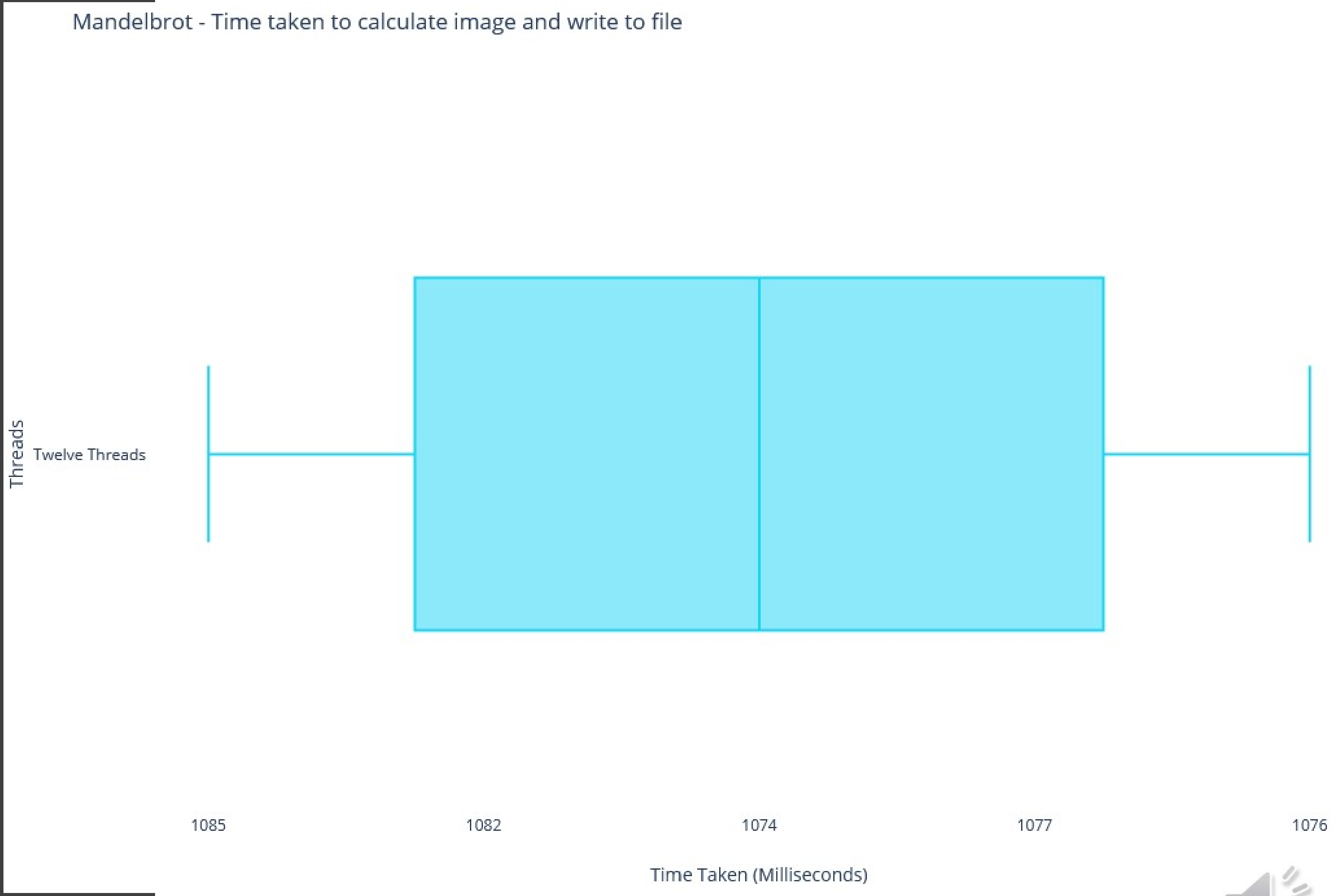
Eight Threads Timings –

- 1516ms
- 1516ms
- 1525ms
- 1527ms
- 1522ms



Twelve Threads Timings –

- 1085ms
- 1082ms
- 1074ms
- 1077ms
- 1076ms



How much faster is threading compared to the original program?

- One – 95% faster *(106270ms)*
- Four – 97% faster *(109159ms)*
- Six – 98% faster *(109748ms)*
- Eight – 98% faster *(110077ms)*
- Twelve – 99% faster *(110552ms)*

How much faster is threading than splitting the image?

- Four – 87% faster *(17349ms)*
- Six – 90% faster *(17848ms)*
- Eight – 92% faster *(18335ms)*
- Twelve – 94% faster *(18609ms)*

Results

- Splitting the image worked to reduce the time – but only when split into around six sections, anymore and it was slower…
- Threading the image worked to reduce the time by a large amount and worked as expected – *more threads = more efficient*
- In conclusion, parallelising the program using CPU multi-threading is the most efficient way to calculate the Mandelbrot set.

So, how did I parallelise the Mandelbrot set?

Splitting the computation of the Mandelbrot Set up into a desired amount of **threads** to carry out an equal distribution of work.

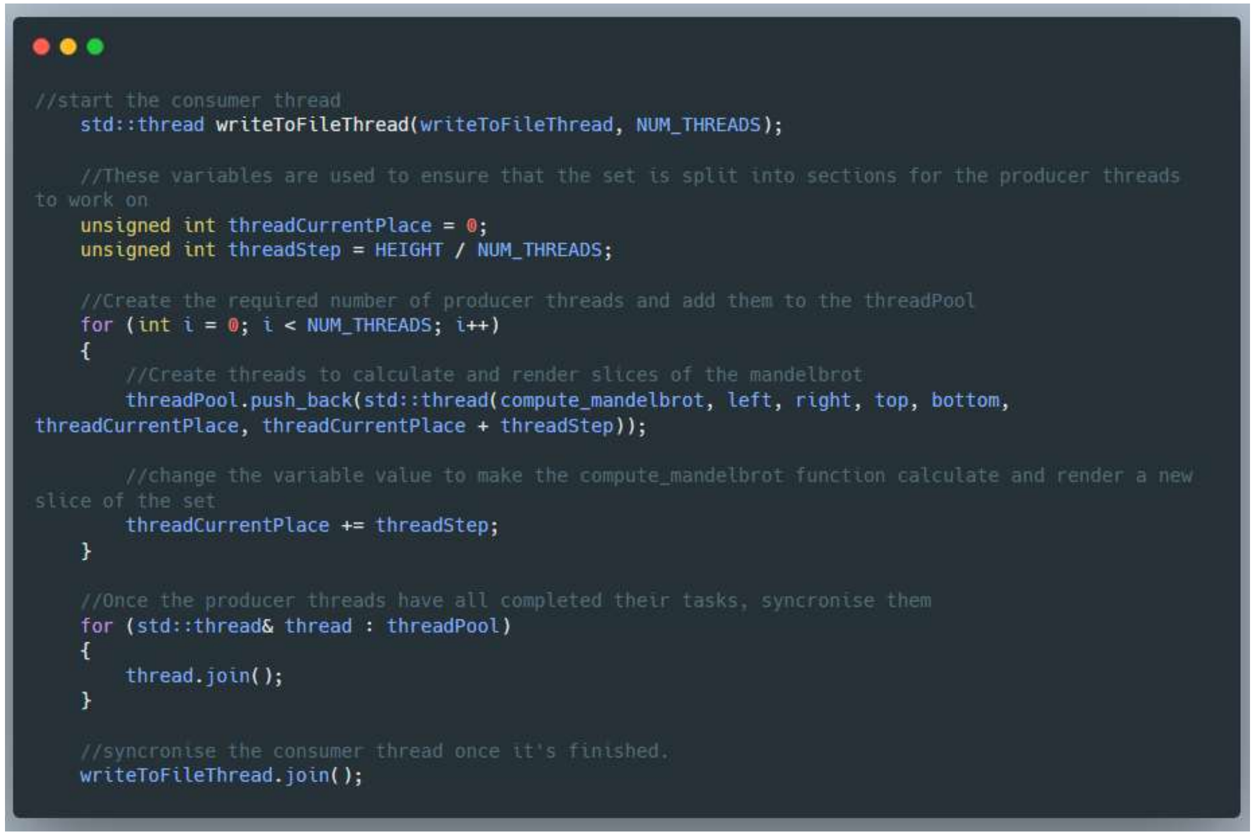
Utilising a **vector** to store/run all the threads to be used by the application.

Creating a separate **thread** to manage the TGA output of the final image.

Using a **mutex** in conjunction with a **conditional variable** to block the consumer thread until all other producer threads have finished.

Created an **atomic variable** to be shared by all threads, to ensure it was thread safe and that no errors would occur.

How were the threads used?



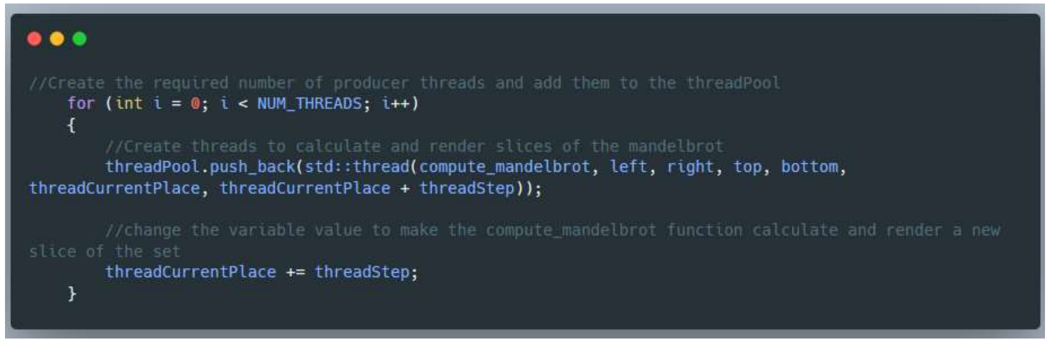
```
//start the consumer thread
    std::thread writeToFileThread(writeToFileThread, NUM_THREADS);

    //These variables are used to ensure that the set is split into sections for the producer threads
to work on
    unsigned int threadCurrentPlace = 0;
    unsigned int threadStep = HEIGHT / NUM_THREADS;

    //Create the required number of producer threads and add them to the threadPool
    for (int i = 0; i < NUM_THREADS; i++)
    {
        //Create threads to calculate and render slices of the mandelbrot
        threadPool.push_back(std::thread(compute_mandelbrot, left, right, top, bottom,
threadCurrentPlace, threadCurrentPlace + threadStep));

        //change the variable value to make the compute_mandelbrot function calculate and render a new
slice of the set
        threadCurrentPlace += threadStep;
    }

    //Once the producer threads have all completed their tasks, syncronise them
    for (std::thread& thread : threadPool)
    {
        thread.join();
    }

    //syncronise the consumer thread once it's finished.
    writeToFileThread.join();
```

- I used threads within a thread pool to split the set into equal sections, that ran at the same time as each other to make the program more efficient.

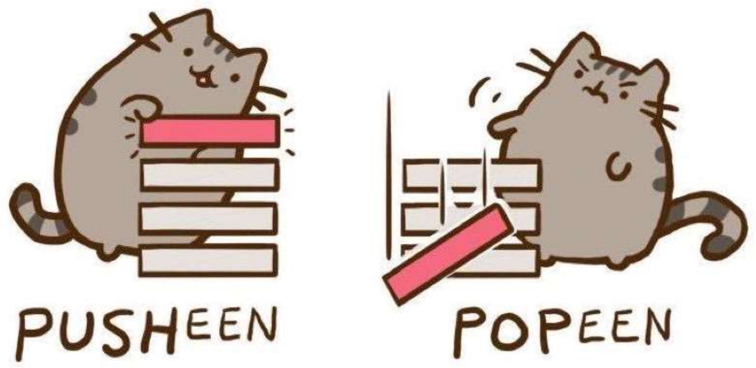
- I also used a thread to manage the writing of the image to a file.

How was the vector used?

- The vector was used to create a *pool* of threads.
- Better than an array as vector is dynamic
- Program uses push_back to add the threads to vector when it is created.

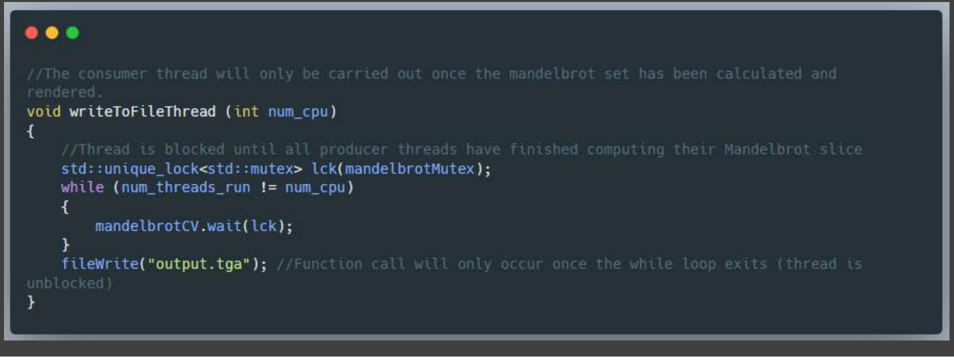


```
//Create the required number of producer threads and add them to the threadPool
    for (int i = 0; i < NUM_THREADS; i++)
    {
        //Create threads to calculate and render slices of the mandelbrot
        threadPool.push_back(std::thread(compute_mandelbrot, left, right, top, bottom,
threadCurrentPlace, threadCurrentPlace + threadStep));

        //change the variable value to make the compute_mandelbrot function calculate and render a new
slice of the set
        threadCurrentPlace += threadStep;
    }
```



What are mutexes and how did I use them?

- Mutexes are a class within C++ that are able to control access to certain objects within a program
- In this program, we have used a mutex to control the writeToFileThread which is the **consumer thread.**
- This means that **producer threads** calculating the Mandelbrot set, must wait for each of the threads to be finished before writing the image to file.

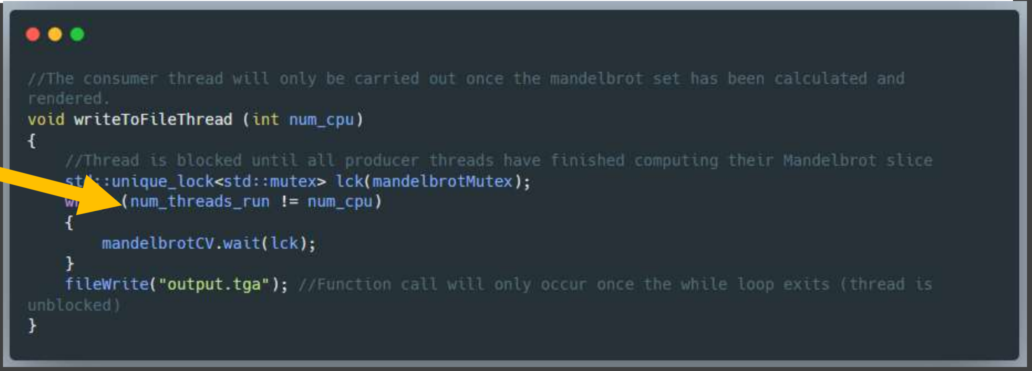


```
//The consumer thread will only be carried out once the mandelbrot set has been calculated and
rendered.
void writeToFileThread (int num_cpu)
{
    //Thread is blocked until all producer threads have finished computing their Mandelbrot slice
    std::unique_lock<std::mutex> lck(mandelbrotMutex);
    while (num_threads_run != num_cpu)
    {
        mandelbrotCV.wait(lck);
    }
    fileWrite("output.tga"); //Function call will only occur once the while loop exits (thread is
unblocked)
}
```

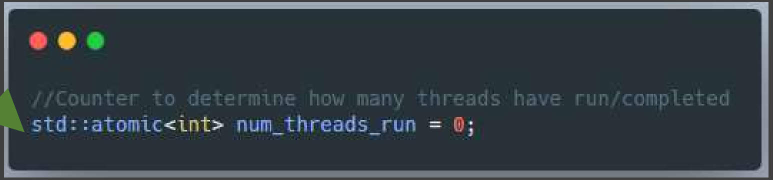
What are atomic variables and how did I use them?

- I used an atomic variable to ensure that the variable storing the number of threads run would be ‘thread-safe’ as it is being written to and read by threads.

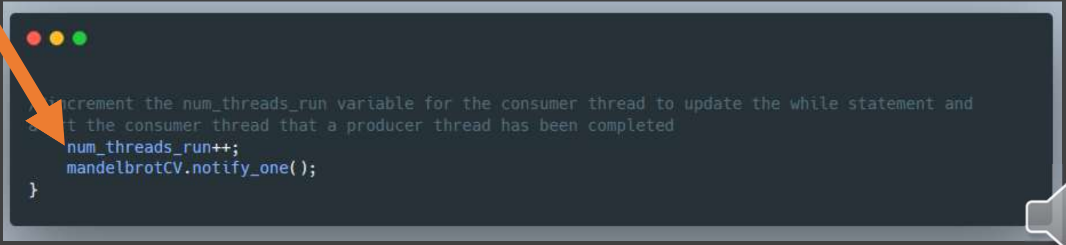
- The atomic variable counts when the thread has been completed, which is then used to determine when to unlock the mutex.



```
//The consumer thread will only be carried out once the mandelbrot set has been calculated and
rendered.
void writeToFileThread (int num_cpu)
{
    //Thread is blocked until all producer threads have finished computing their Mandelbrot slice
    st[illegible]::unique_lock<std::mutex> lck(mandelbrotMutex);
    wh[illegible](num_threads_run != num_cpu)
    {
        mandelbrotCV.wait(lck);
    }
    fileWrite("output.tga"); //Function call will only occur once the while loop exits (thread is
unblocked)
}
```



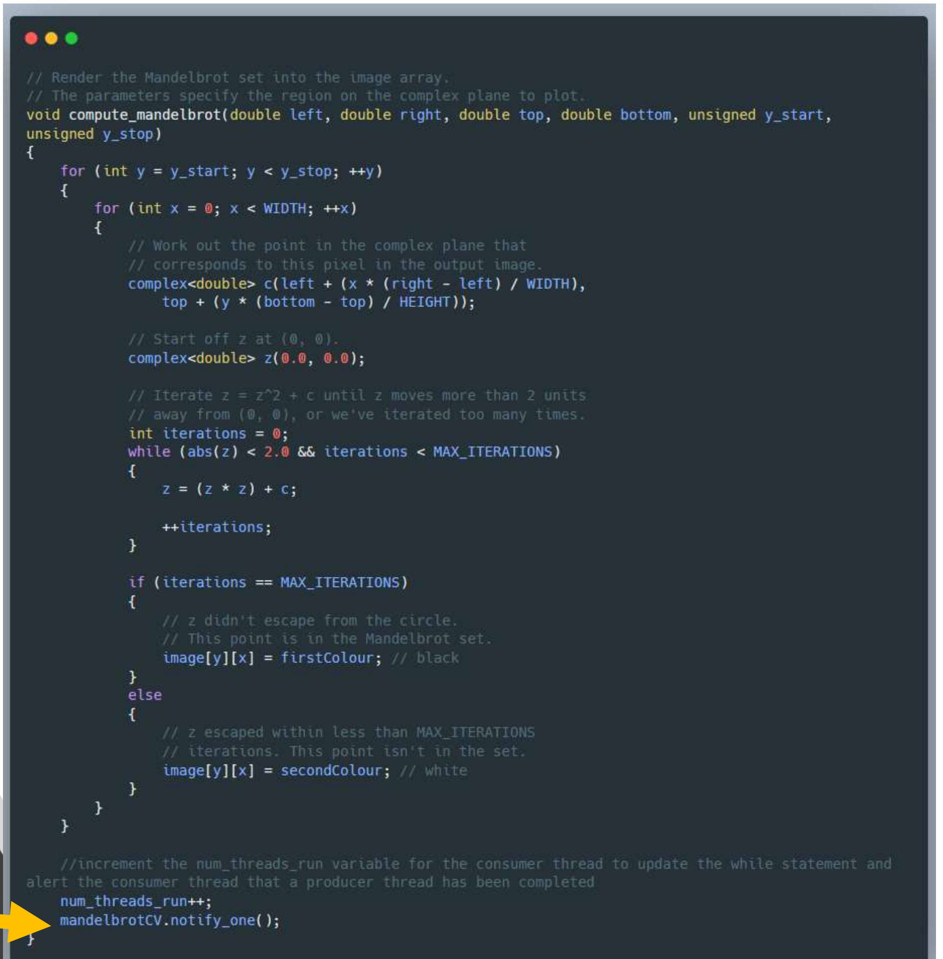
```
//Counter to determine how many threads have run/completed
std::atomic<int> num_threads_run = 0;
```



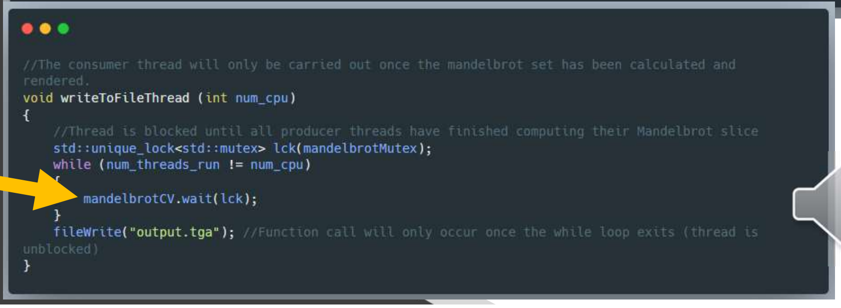
```
/[illegible]ncrement the num_threads_run variable for the consumer thread to update the while statement and
a[illegible]t the consumer thread that a producer thread has been completed
    num_threads_run++;
    mandelbrotCV.notify_one();
}
```

What are conditional variables and how did I use them?

- I used a conditional variable to identify when the producer threads have finished and tells the mutex and vector when a thread has run.
- It can be seen at the bottom of the compute_Mandelbrot function and within the consumer thread.
- .notify_one() is used to unblock the consumer thread; the consumer thread will only unblock once all threads have called on it through this method.
- .wait() is used to keep the consumer thread blocked, it is telling the mutex in the consumer thread to stay blocked as the while requirement has not been reached yet



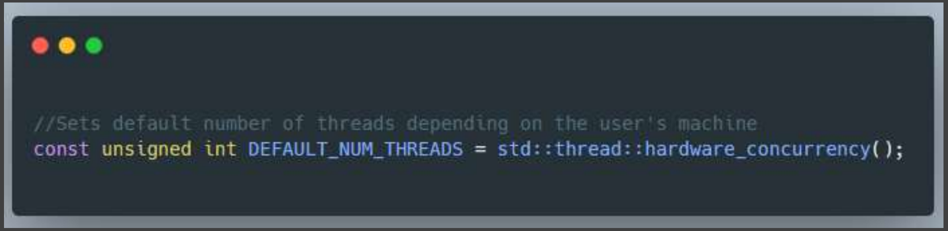
```
// Render the Mandelbrot set into the image array.
// The parameters specify the region on the complex plane to plot.
void compute_mandelbrot(double left, double right, double top, double bottom, unsigned y_start,
unsigned y_stop)
{
    for (int y = y_start; y < y_stop; ++y)
    {
        for (int x = 0; x < WIDTH; ++x)
        {
            // Work out the point in the complex plane that
            // corresponds to this pixel in the output image.
            complex<double> c(left + (x * (right - left) / WIDTH),
                top + (y * (bottom - top) / HEIGHT));

            // Start off z at (0, 0).
            complex<double> z(0.0, 0.0);

            // Iterate z = z^2 + c until z moves more than 2 units
            // away from (0, 0), or we've iterated too many times.
            int iterations = 0;
            while (abs(z) < 2.0 && iterations < MAX_ITERATIONS)
            {
                z = (z * z) + c;

                ++iterations;
            }

            if (iterations == MAX_ITERATIONS)
            {
                // z didn't escape from the circle.
                // This point is in the Mandelbrot set.
                image[y][x] = firstColour; // black
            }
            else
            {
                // z escaped within less than MAX_ITERATIONS
                // iterations. This point isn't in the set.
                image[y][x] = secondColour; // white
            }
        }
    }

    //increment the num_threads_run variable for the consumer thread to update the while statement and
alert the consumer thread that a producer thread has been completed
    num_threads_run++;
    mandelbrotCV.notify_one();
}
```



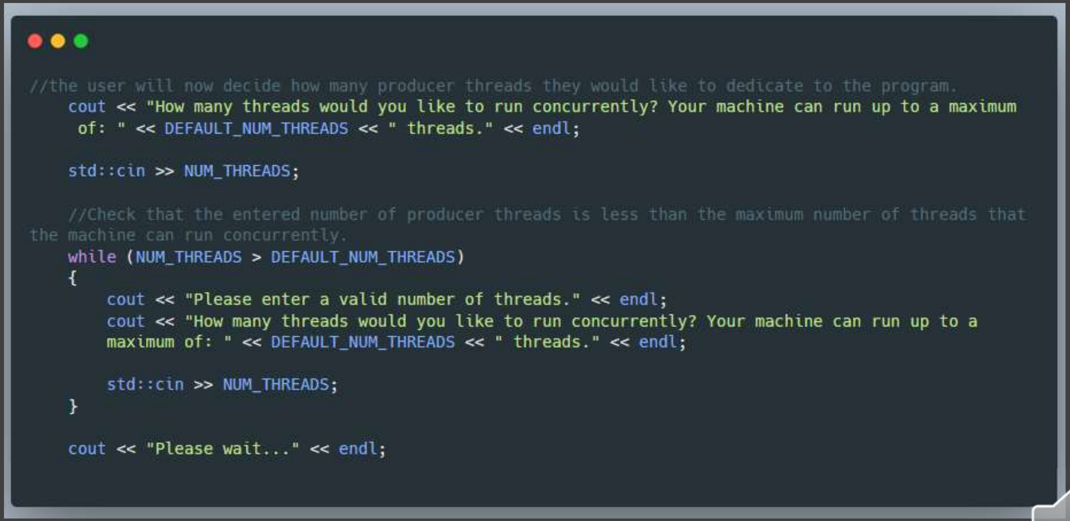
```
//The consumer thread will only be carried out once the mandelbrot set has been calculated and
rendered.
void writeToFileThread (int num_cpu)
{
    //Thread is blocked until all producer threads have finished computing their Mandelbrot slice
    std::unique_lock<std::mutex> lck(mandelbrotMutex);
    while (num_threads_run != num_cpu)
    {
        mandelbrotCV.wait(lck);
    }
    fileWrite("output.tga"); //Function call will only occur once the while loop exits (thread is
unblocked)
}
```

How did I calculate the maximum number threads?

- Not every user will have the same hardware or capability to run a set number of threads.
- To avoid any issues with the number of threads, I used a function that is a part of the thread class.
- It calculates the number of threads that can be run concurrently on the current machine.



```cpp
//Sets default number of threads depending on the user's machine
const unsigned int DEFAULT_NUM_THREADS = std::thread::hardware_concurrency();
```



```cpp
//the user will now decide how many producer threads they would like to dedicate to the program.
    cout << "How many threads would you like to run concurrently? Your machine can run up to a maximum
     of: " << DEFAULT_NUM_THREADS << " threads." << endl;

    std::cin >> NUM_THREADS;

    //Check that the entered number of producer threads is less than the maximum number of threads that
the machine can run concurrently.
    while (NUM_THREADS > DEFAULT_NUM_THREADS)
    {
        cout << "Please enter a valid number of threads." << endl;
        cout << "How many threads would you like to run concurrently? Your machine can run up to a
        maximum of: " << DEFAULT_NUM_THREADS << " threads." << endl;

        std::cin >> NUM_THREADS;
    }

    cout << "Please wait..." << endl;
```

References

- https://medium.com/swlh/c-mutex-write-your-first-concurrent-code-69ac8b332288 (Threading, Mutexes and Thread Vectors)
- https://thispointer.com/c11-multithreading-part-7-condition-variables-explained/ (Conditional Variables)
- https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency (No. Threads able to run concurrently)
- https://www.youtube.com/watch?v=oE_D3IgBJi8 (Atomic Variables and vectors)
- http://www.cplusplus.com/reference/vector/vector/push_back/ (Vectors)
- https://chart-studio.plotly.com/create/box-plot/#/ (Box Plot Creator)
- https://carbon.now.sh/ (Code snippet Creator)
- https://simple.wikipedia.org/wiki/Mandelbrot_set

Thank you for listening!

If you have any questions, please email me at 1602119@uad.ac.uk