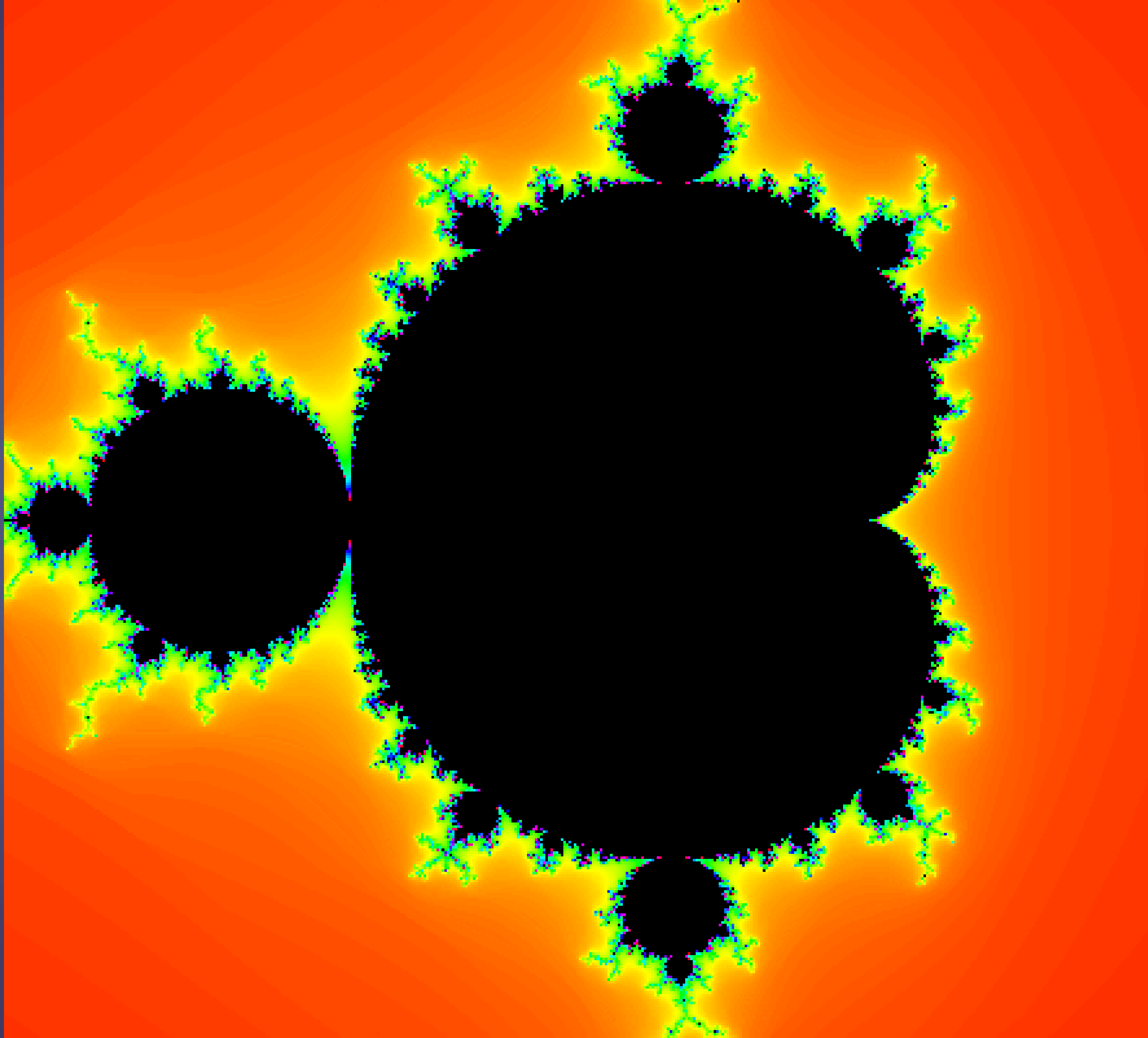
# Using CPU Multithreading to Parallelise the Mandelbrot Algorithm

ISAAC BASQUE-RICE
(1901124) – CMP202

# What is the Mandelbrot Set?

- A symmetrical geometric fractal
- Can zoom in to the set infinitely, will eventually repeat itself
- Makes use of a complex equation that means generation can take some time and/or a lot of processing power



# What is the problem?

- Requires a lot of processing power
- It can take a long time even with good hardware
- Need to find a way to make it more efficient

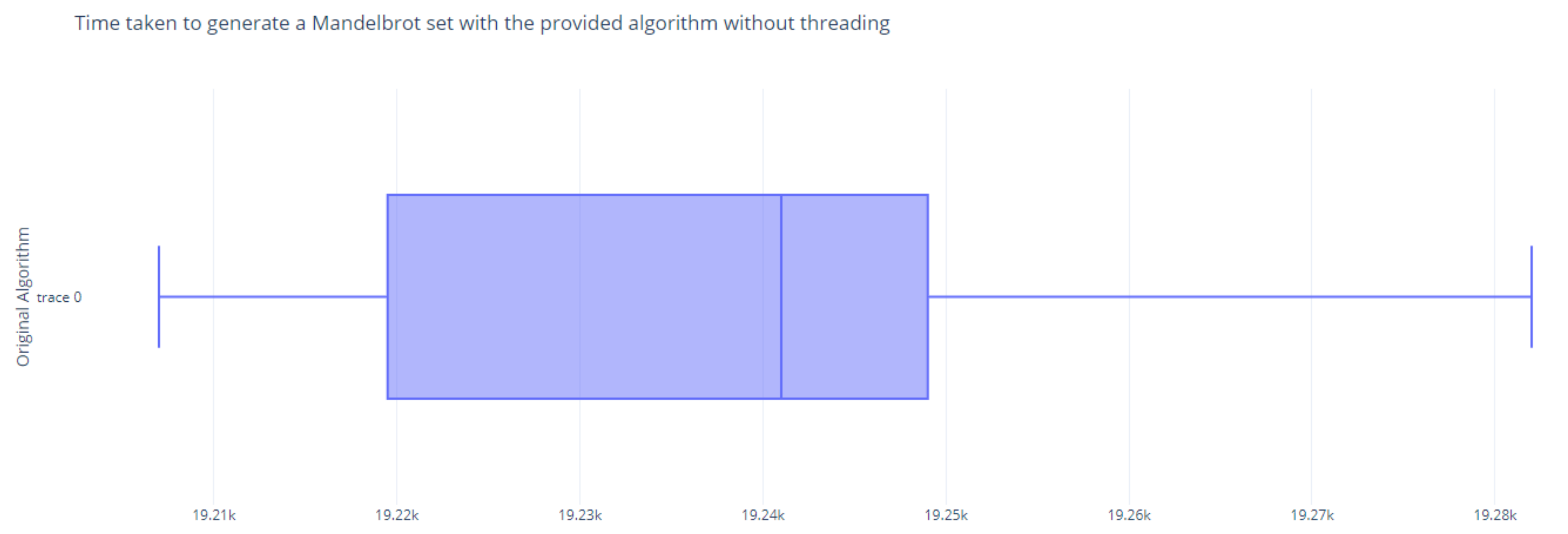
# How was the algorithm tested?

- The program allows you to choose colour
- The program was run 9 times per thread count
- Timing begins when generation begins and ends when it's written to the file

# My Hardware/Software Configuration

- Dell Inspiron laptop with Manjaro Linux installed
- CLion IDE running Clang compiler
- Intel Core i3-6006 CPU @ 2.0GHz

# How long does it take to run on its own?




# Original Mandelbrot Program

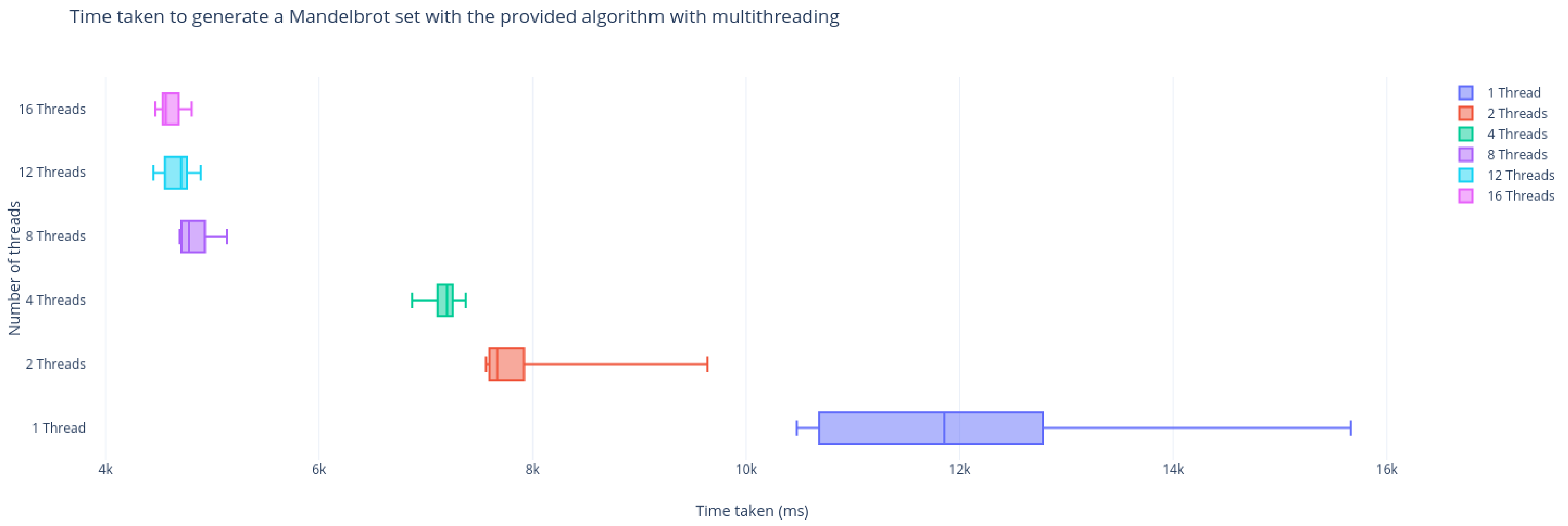
# How can we make this more efficient?

Split the image up and work on the chunks individually

If we can work on multiple chunks at once we can reduce the amount of time needed

# Testing this approach

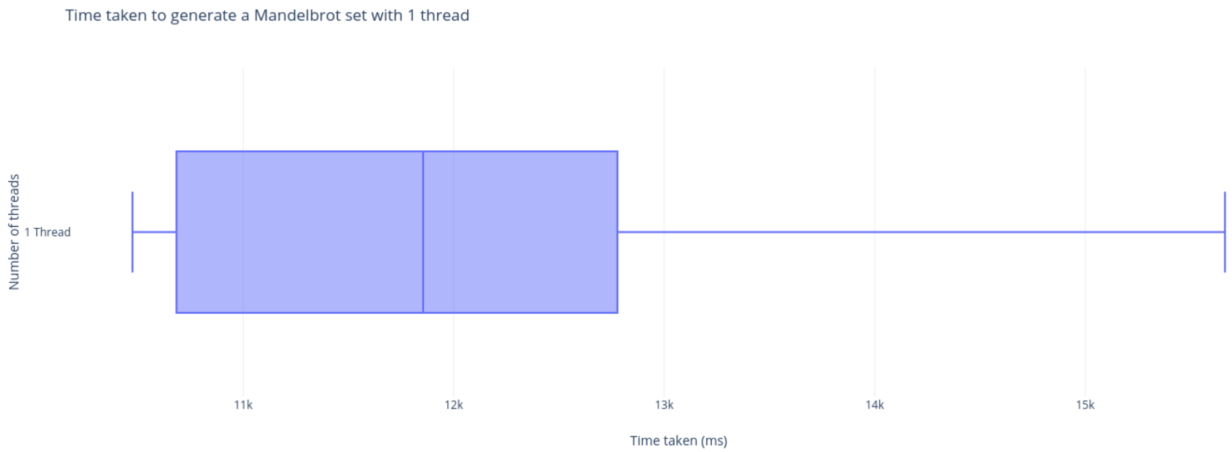
- Black background, coloured foreground
- Only difference from image to image is colour
- Clock begins when generation begins and ends when written to file
- Ran 9 times
- Split into 1, 2, 4, 8, 12, and 16 chunks
- Also function that outputs metadata about the image



Time taken to generate a Mandelbrot set with the provided algorithm with multithreading
Number of threads
16 Threads
12 Threads
8 Threads
4 Threads
2 Threads
1 Thread
4k
6k
8k
10k
12k
14k
16k
Time taken (ms)
1 Thread
2 Threads
4 Threads
8 Threads
12 Threads
16 Threads

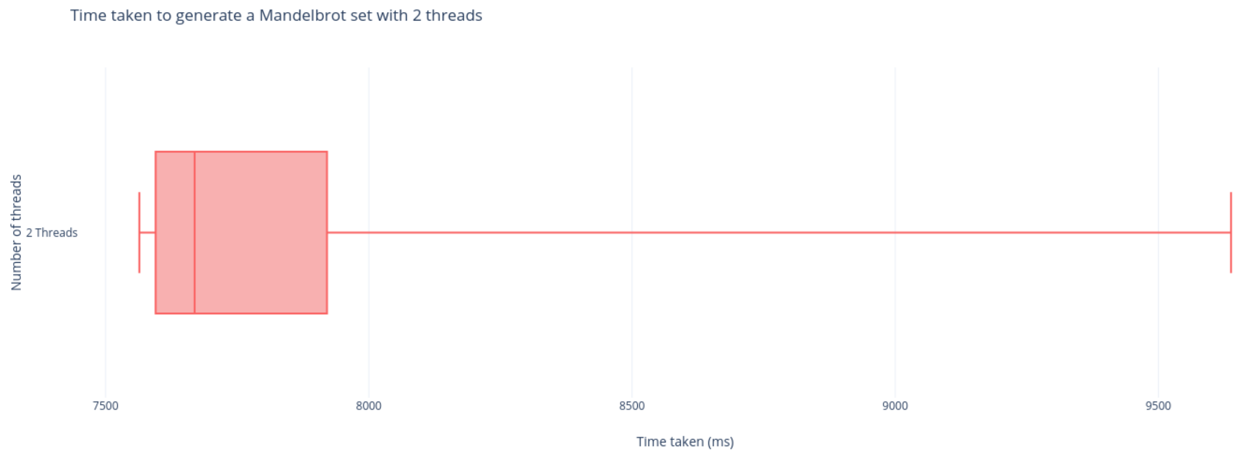
# 1 Thread Timings

- 15664 ms
- 10724 ms
- 10474 ms
- 10561 ms
- 13101 ms
- 12180 ms
- 11855 ms
- 12670 ms
- 11646 ms



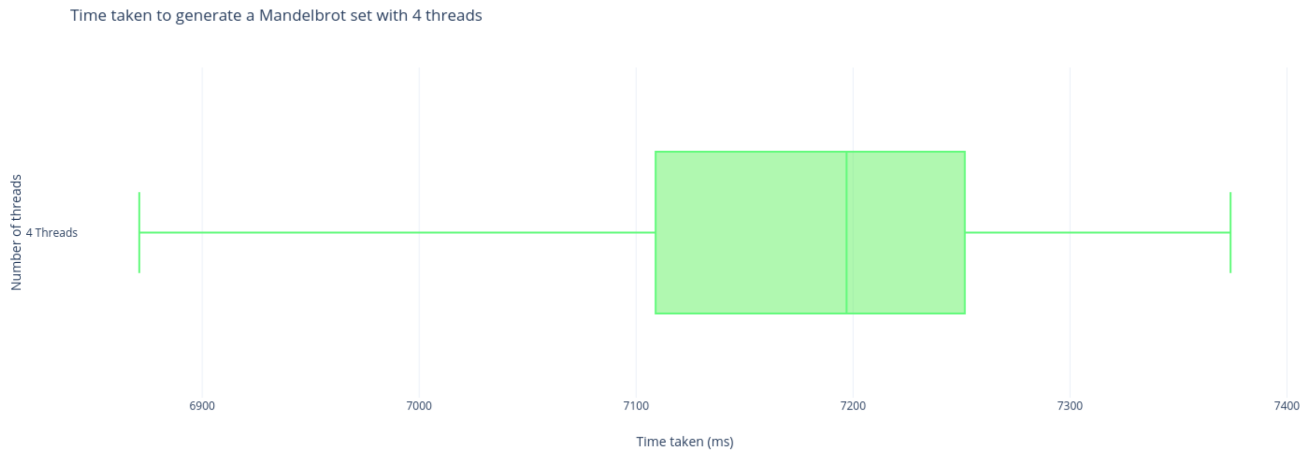
# 2 Thread Timings

- 7580 ms
- 9638 ms
- 8147 ms
- 7564 ms
- 7600 ms
- 7845 ms
- 7828 ms
- 7669 ms
- 7608 ms



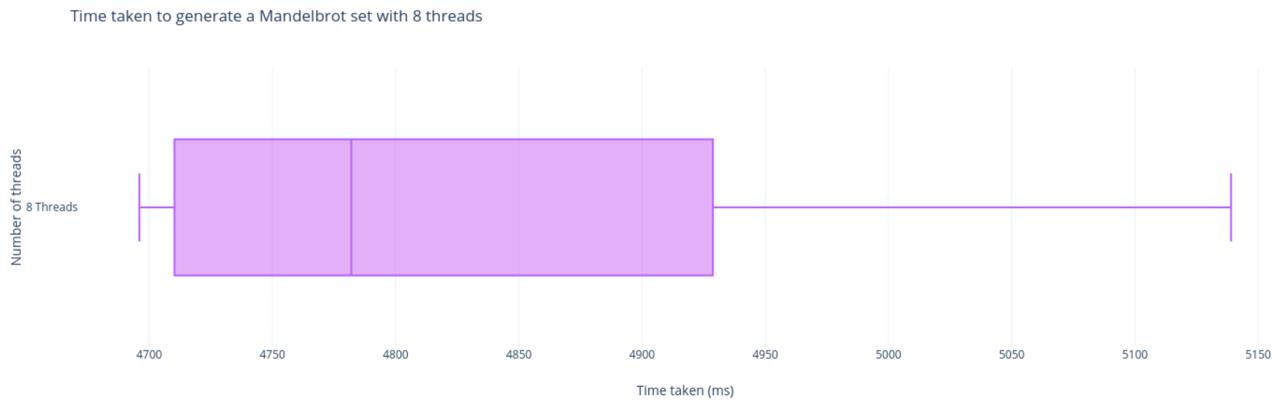
# 4 Thread Timings

- 7147 ms
- 7194 ms
- 7197 ms
- 7231 ms
- 7374 ms
- 6995 ms
- 7313 ms
- 6871 ms
- 7200 ms



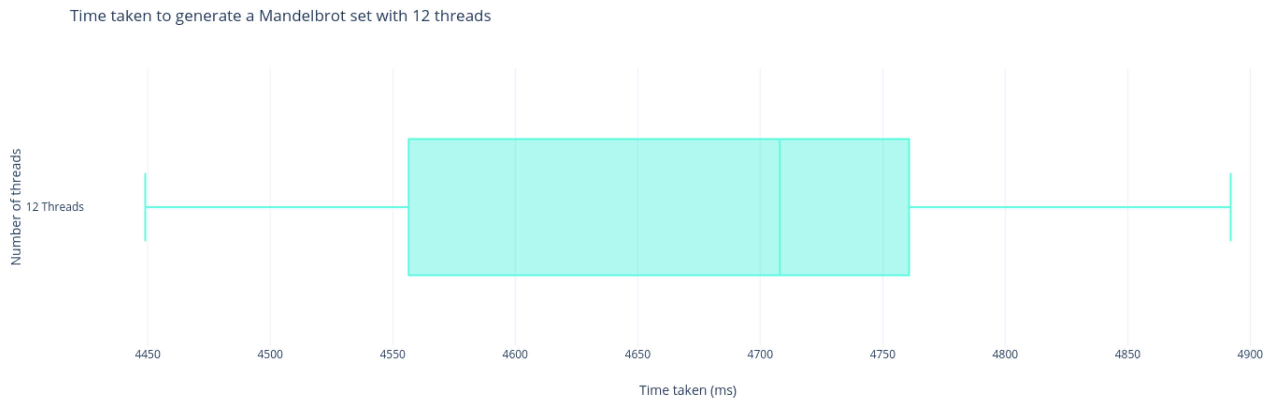
# 8 Thread Timings

- 4806 ms
- 4696 ms
- 4782 ms
- 4759 ms
- 4903 ms
- 5006 ms
- 4708 ms
- 4711 ms
- 5139 ms



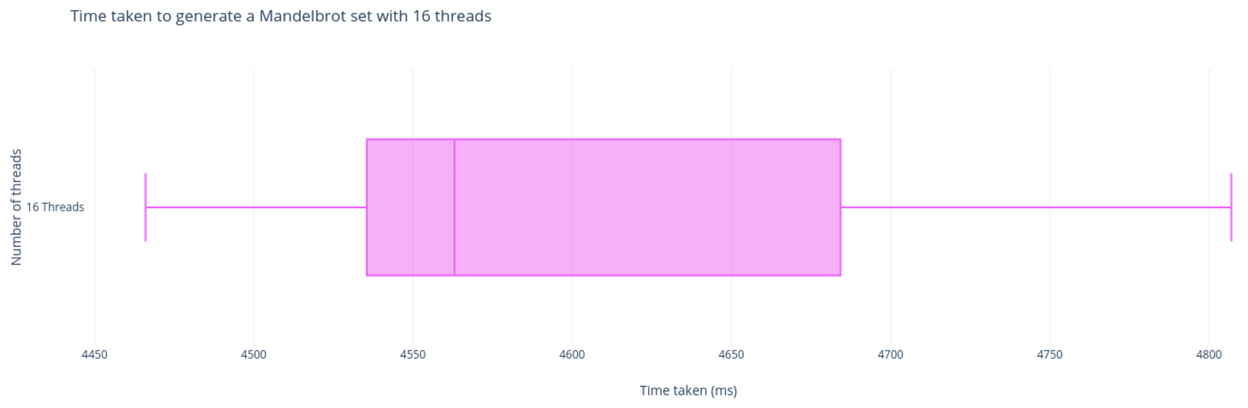
# 12 Thread Timings

- 4711 ms
- 4449 ms
- 4727 ms
- 4647 ms
- 4537 ms
- 4892 ms
- 4563 ms
- 4862 ms
- 4708 ms



# 16 Thread Timings

- 4605 ms
- 4531 ms
- 4537 ms
- 4766 ms
- 4657 ms
- 4466 ms
- 4563 ms
- 4555 ms
- 4807 ms



# How much faster is using threads? (on average)

- Average without multithreading: 19238.11 ms
- 1 thread: 12097.22 ms (37.12%)
- 2 threads: 7942.11 ms (58.72%)
- 4 threads: 7169.11 ms (62.74%)
- 8 threads: 4834.44 ms (74.87%)
- 12 threads: 4677.33 ms (75.69%)
- 16 threads: 4609.67 ms (76.04%)

# Results

- Multithreading the program has a significant impact on the speed at which it is run, but only up to a certain point due to CPU constraints
- In conclusion, parallelising the program using CPU multithreading affords a significant time advantage over generating it from a single function.

# So how did I go about parallelising the Mandelbrot Set?

# Requirements

## 01

Running at least three threads, with at least two different thread functions

## 02

Sharing resources safely between threads

## 03

Signalling between threads

# Thread functions

- The Mandelbrot was computed using an array of threads (threads[i])
- A thread that wrote the current time out to a text file
- Arguably also the main thread, where I called write_tga()

```
auto* threads = new std::thread[threadNum]; // array of threads for computing

// populate the array
for (int i = 0; i < threadNum; ++i) {
    threads[i] = std::thread(compute, left, right, top, bottom, (0 + chunkSize * i), chunkSize + chunkSize * i, colour);
}
std::thread timeWriteThread(write_time); // write the current time
```

# Sharing Resources

- Used Mutexes and Atomic Variables
- Mutex:
  - Locks a condition variable that prevents the program from continuing
- Atomic Integer:
  - Keeps track of the number of threads run by incrementing when a thread is finished

```
        }
    }        std::atomic_int runThreadsCount
}            atomic int that keeps count of the number of threads that have been
std::cout << runThreadsCount.fetch_add(1) + 1 << std::endl;

countLock.lock();
cv.notify_one();
countLock.unlock();
```

```
std::unique_lock<std::mutex> lck(countLock);
while (runThreadsCount != threadNum) {
    cv.wait(lck);
}
```

# Signalling between threads

- Condition Variable:
  - Used to prevent program rushing ahead
  - In tandem with mutex

```
countLock.lock();
cv.notify_one();
countLock.unlock();
```

```
// join the threads in the array
for (int i = 0; i < threadNum; ++i) {
    threads[i].join();
}
```

```
std::mutex countLock; // mutex for lo
std::atomic<int> runThreadsCount(0);
std::condition_variable cv; // condit
```

# Thank you for listening