CMP202 Data Structures and Algorithms 2 Assessment

Ethan Hastie

The Problem

- My application creates an interactive Mandelbrot set using parallelisation methods, allowing you to zoom into two specified coordinates.
- My first aim is to demonstrate how the implementation of a CPU farming pattern benefits the computation of the Mandelbrot.
- My second aim is to demonstrate how multithreaded rendering impacts my application.
- The Mandelbrot set is particularly good problem as it can be solved using fork-join parallelism or using the farming pattern. It requires a significant level of computation so it benefits from implementation of parallel methods.

Structure

- The Mandelbrot set was parallelised using the CPU farming pattern (where each line became a task: more tasks than CPUs!)

i. Each task was added to a queue container, where a specified number of worker threads are able to be allocated tasks to allow them to compute the Mandelbrot set.

ii. Once a thread has been allocated a task from the queue, another thread will take a task from the queue, allowing the previous thread to run their task.

iii. Once they have computed a line of the Mandelbrot, it will take another task from the queue or exit if there are no more tasks available.

Structure

- So, once the threads have run out of tasks or stopped computing the Mandelbrot…

i. A channel will take the Mandelbrot image and carry this to the render task.

ii. The channel was implemented using mutexes and condition variables.

- The second task implements SFML multithreaded drawing to render the Mandelbrot image to the display.

i. The main thread deals with window creation and event handling such as zooming and closing the window. So, this separates them into two separate threads, with rendering and events dealt with separately.

ii. Once the Mandelbrot set has been rendered to the display, users can zoom back and forth between two sets of coordinates.

iii. Rendering can only be done on one thread at a time due to its OpenGL context so it was safer to test on one thread.

Threads

- For our first task, the farming pattern here for the Mandelbrot set makes use of a task queue available to all of the worker threads.

i. When we are adding tasks to the queue, this process is protected by a mutex.

ii. When the worker threads are assigned tasks from the queue, this process is protected by a mutex.

- Worker threads will check to see if the task queue is empty and if so then this unlocks the queue mutex and will allow them to stop computing the set.

Threads

- Once the farm has been completed or there are no more tasks, a channel is used to signal the render task to render the Mandelbrot image to display.
- Rendering can only be used by one thread at a time, so I created a single thread function that will render the Mandelbrot to the display.

i. This was implemented so that when a user switches between coordinates, the render threads do not overwrite each other for control of the display and crash the application.

ii. A window also cannot be active in more than one thread at one time, so my render task is single threaded.

iii. If a user zoomed into a new set of coordinates, the current render task is not terminated. A new render thread is not created and so it continues to use that same thread to render it to the display.

Presentation – Performance Evaluation

- I conducted four tests on my application using different thread groups and measured them across different areas.

i. Test 1 - 1 render thread and 1 worker thread

ii. Test 2 - 1 render thread and 2 worker threads

iii. Test 3 – 1 render thread and 4 worker threads

iv. Test 4 – 1 render thread and 8 worker threads

- Each test measured the application over 101 iterations to allow for a view of its typical behavior as well as to make it easier to measure the mean.
- The MAX_ITERATIONS value was kept as 500, and the size dimensions of the Mandelbrot are 1920 x 1200.

i. The task based approach allows for 1200 tasks.

Presentation – Performance Evaluation

- Measurements included:

i. The time for each slice to be computed,

ii. The farm's overall time taken (including setting up tasks and executing them),

iii. The time taken for the Mandelbrot to render to the display (if worker threads for the Mandelbrot had an impact on its performance)

iv. The program's overall time taken.

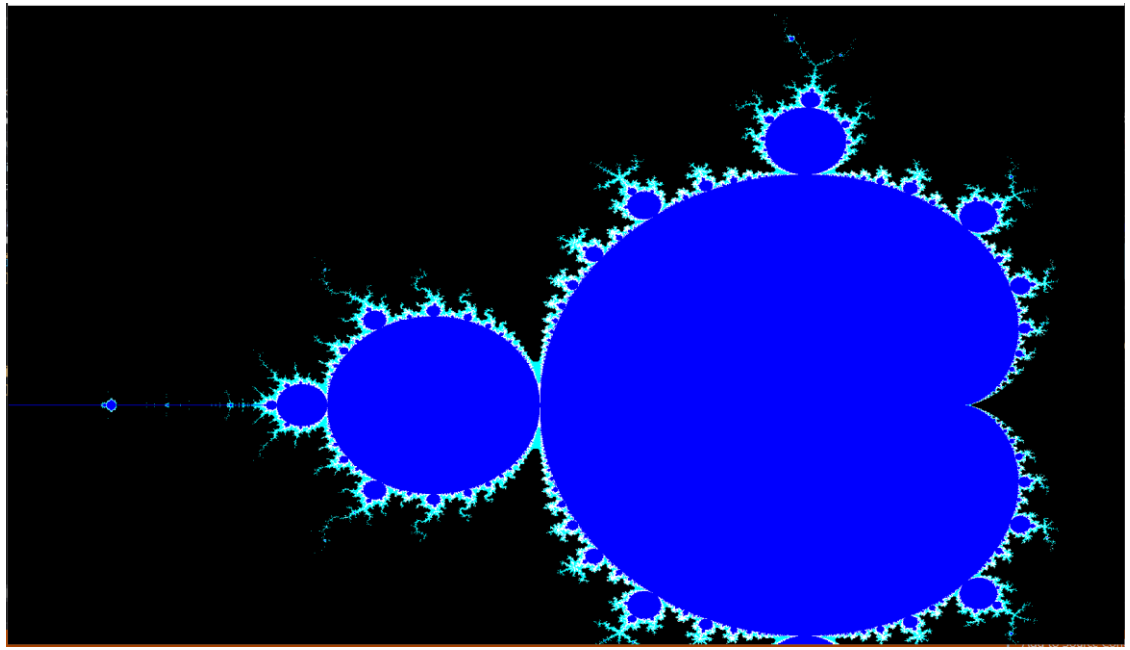
Performance Evaluation - Specification

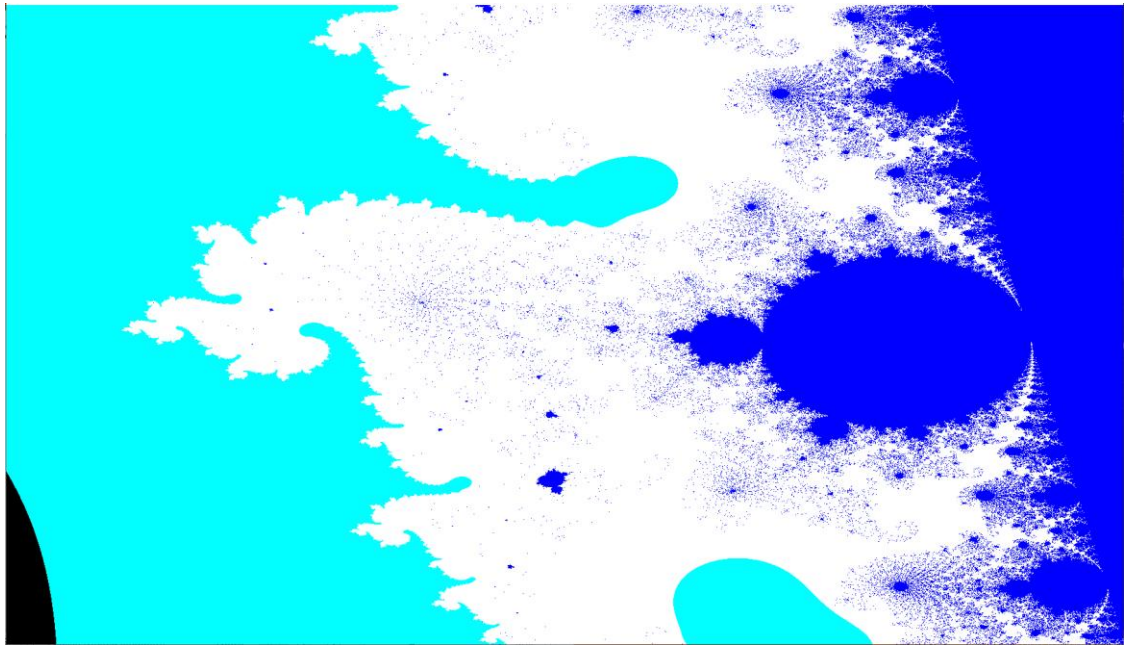
- I implemented my application on the CPU
- Processor: Intel(R) Core™ i7-8565U CPU @ 1.80GHz 1.99 GHz

Processor: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz

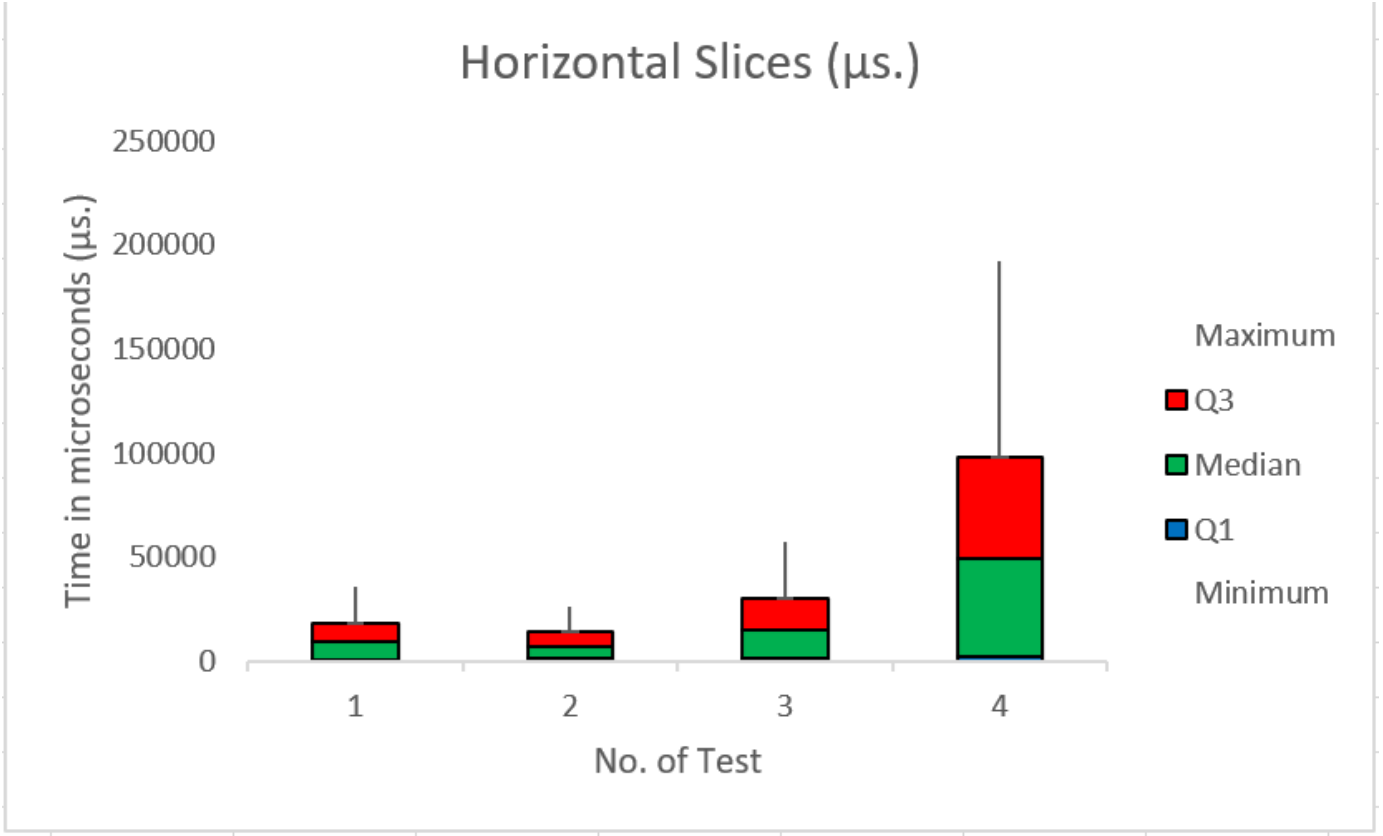
Processor: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz (8 CPUs), ~2.0GHz

RESULTS

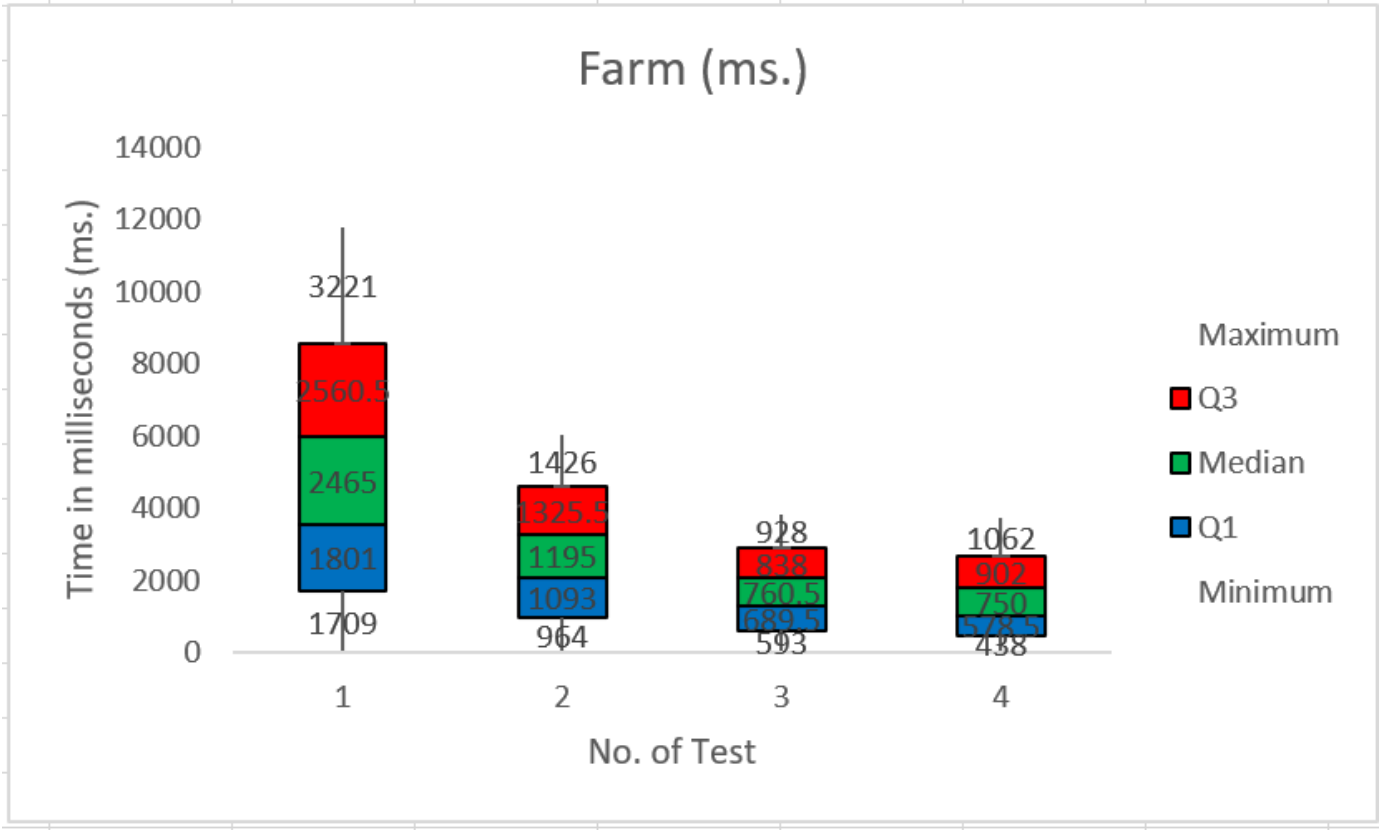




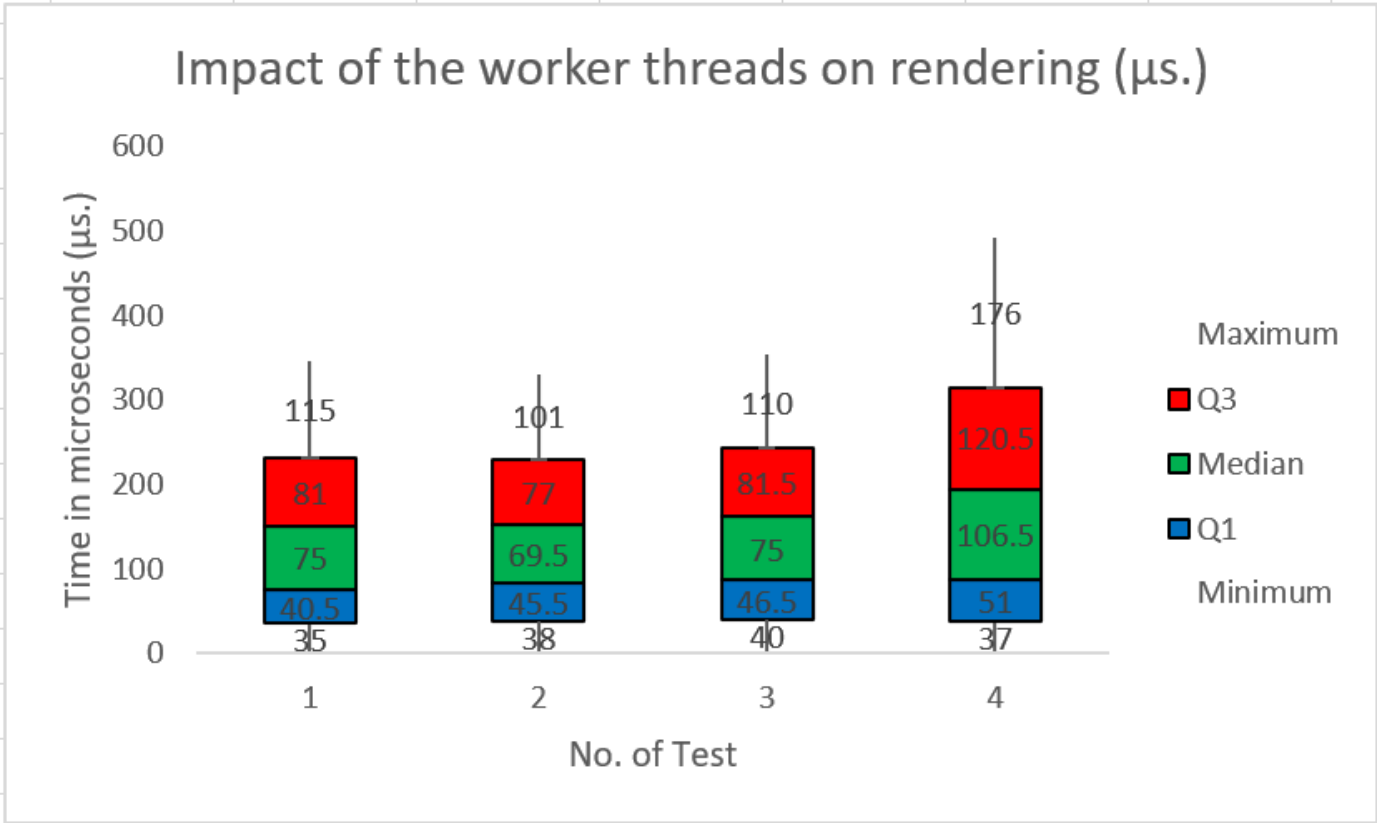
Horizontal Slices results



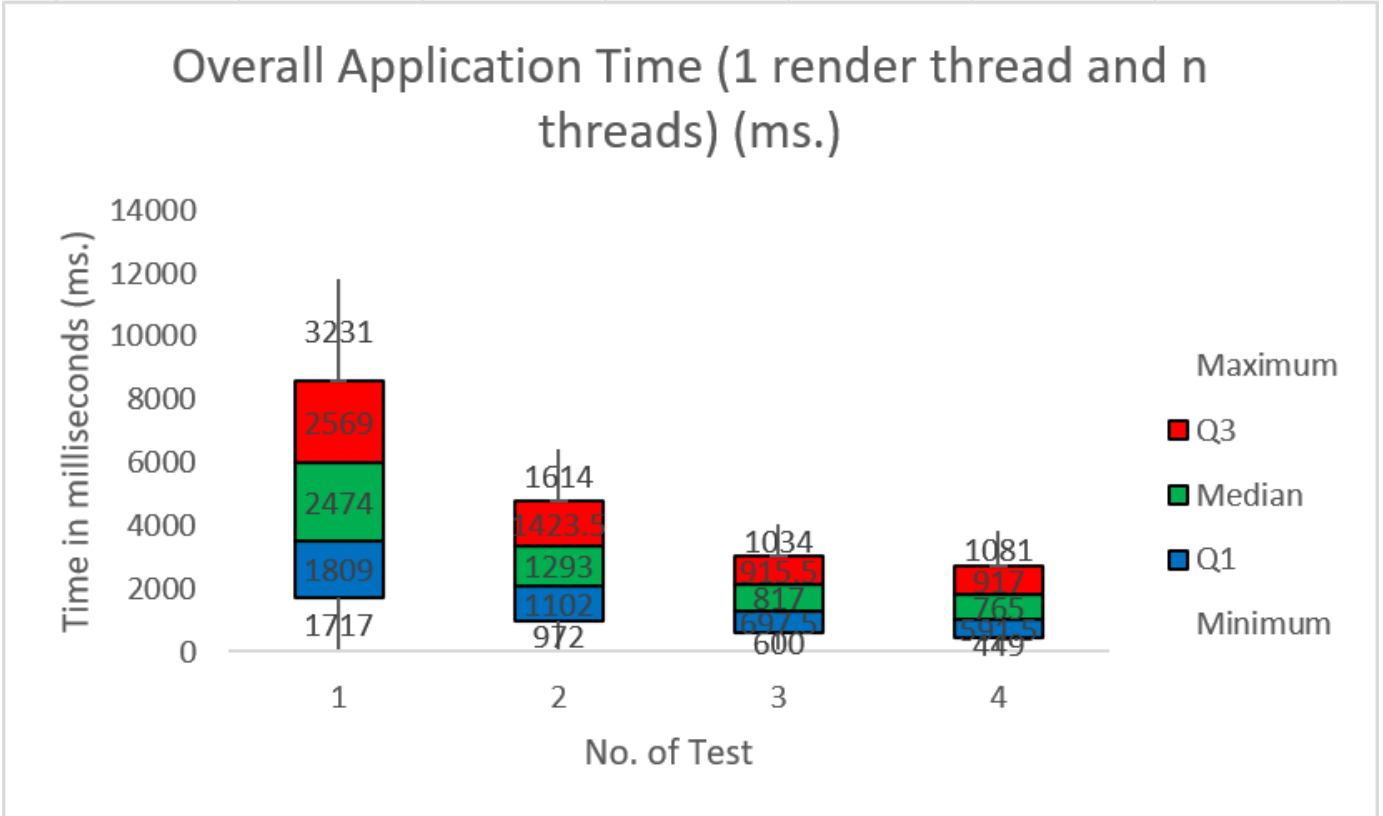
Farming Pattern results



SFML Rendering results



Overall results



Discussion

- In terms of the horizontal slices, we could see a greater variation of slice results because we had 1200 tasks being calculated, with 101 iterations (101 x 1200 = 12200 times) so it allowed us to see a greater mixture of the short and long compute times.

i. This meant some of the measurements became harder to measure as it was computing so fast (in milliseconds), which is why we converted it into microseconds to get a more accurate reading.

ii. Since it is computing a single line, we are computing significantly less than if every thread had a single section to compute. This allowed the significantly shorter slices to be computed very easily, especially those on the outer set (where it isn't touching the set – MAX_ITERATIONS)

iii. A reason why there are much longer measurements than others is because it is dealing with a line in the set, where it spends much longer iterating to determine if it's in the set.

Discussion

- We also saw across all the tests that the minimum time it took to calculate a slice was around 40 microseconds.
- However, with the increase of the worker threads in the farm across the four tests it was found that the longest time to calculate a slice was 93899 microseconds.

i. This is unusual as although we are increasing the amount of worker threads computing the Mandelbrot set, the results across all the four tests should be around the same.

ii. The slices across all the tests should take around the same time as we are only helping to boost the amount of worker threads to compute the Mandelbrot.

iii. This also unusual as eight worker threads should boast the best performance in terms of the overall time of the farm, and it shouldn’t affect any times of individual slices.

Discussion

- The farming results demonstrate that as we increment the number of worker threads available, we see a decrease in the overall time of the farm.

i. In the results, there is a decrease when comparing Test 1 and Test 2. When we increment the available workers, we are not attempting to make the computation of the slices faster: we are using the divide and conquer approach to allow the tasks to be split up and be executed faster than if one thread was doing it themselves!

ii. In addition, we also see a decrease in the time between Test 2 and Test 3. The increase in the number of threads from 2 to 4 does demonstrate a smaller increase than compared to Test 1 and Test 2. The increase from 4 to 8 threads meant a small decrease in the overall time of the farm!

Discussion

- In order to accurately measure the rendering task's effect on my application, I measured the overall time it took to render it to the display in microseconds.
- Instead of starting a new thread and deactivating a window every time a user would want to zoom in, the application uses the same thread. This means that this avoids the overhead of starting a new thread and scheduling it every time a user wishes to zoom in.
- The rendering does not affect the performance of my program significantly as it was quite simple to implement.

i. It would convert the image into a texture and convert this into a sprite, allowing it to be rendered to the screen.

- Every time a user zoomed in or out it would recalculate the Mandelbrot image and pass a new image to the channel, which would become the new image on screen.

Discussion

- The overall time taken to finish my application is also benefited from multithreading.
- This is helped as we increase the thread workers in the farm.
- In Test 1, we only have one thread taking tasks off the queue so the farm does not benefit as it is taking much longer to divide the work to just one thread.
- In Test 2, we once again see a decrease in the overall application time, with Test 3 continuing this trend.

i. The increased workers in the farm allow the all the tasks to be completed faster overall.

ii. Comparing Test 4 to 3 shows a small decrease in the overall completion of the program.

- From the results, it is clear that the task that has the most significant impact on performance is the farm and benefits the most from multithreaded programming.

Evaluation

- My solution is one method of demonstrating the benefits of parallelisation in the computation of the Mandelbrot set.
- It allowed for decrease in time for the calculation of every slice, at the cost of having more tasks to compute.

i. The mixture of short and long compute times means that the short slices could be computed very easily.

- As we varied the number of threads, we saw a decrease in the overall time of the farm and the application. This is the part of the program where varying the number of worker threads has a dramatic effect on the performance of the program.

i. We saw this in the results for the overall application time (a decrease in time between Test 1, 2 and 3).

ii. However, when we set the number of thread workers to 8 we didn't see a significant decrease in the overall time of the farm and the application itself.

Evaluation

- This could partially be a design flaw of the farm…

i. Lock the mutex
ii. Thread checks to see if the queue is empty, if it is (break out) but if not then…
iii. Assign a task from the front of the queue
iv. Pop the front of the queue
v. Unlock the mutex
vi. Run that task

- 8 worker threads are waiting for tasks to become available
- Solution: work-stealing.

Evaluation

- One issue behind the farming pattern is that we have a deliberate bottleneck in our program.

i. The presence of the task queue which we add our tasks to.

ii. We use a mutex when threads enter the task queue so when their tasks are being assigned two or more threads do not interrupt each other and perform the same task.

- So we are sacrificing performance to allow our threads to safely access shared resources.

Evaluation

- Possible sources of error:

i. The rendering task and the event loop complicated implementation as it was much easier to have them both in the same thread. However, across all 4 tests this area wasn't detrimental to performance.

ii. When we increase the number of threads in our farm, this shouldn't have an effect on the horizontal slices across all the four tests. However, as Test 4 shows there was a typical increase in the timing of slices when the threads were set to 8.

iii. Work-stealing would allow threads to steal tasks from each other and decrease the overall time of the farm. This implementation would have been effective due to short and long compute times, so threads with short slices to calculate would be able to steal tasks from threads that are dealing with longer slices.

Evaluation

- Another possible implementation of parallelising the Mandelbrot set is fork join parallelism.
- Each thread would be responsible for calculating a section of the Mandelbrot.
- Once each thread finishes calculating their section, a barrier would be used to signal the render task that the Mandelbrot set has been completed.
- In my application, I had already had a method of checking to see if there were any more tasks to be completed (check if queue is empty) so there was no need to use a barrier.
- My omission of the barrier meant I had to find another method of signaling between threads.

i. If a user wants to zoom in our out the channel would simply update the old Mandelbrot image with a new one.

Evaluation

- SFML has better syntax compared to OpenGL. It was quite easy to understand and explain my code better because of the syntax.
- My experience isn’t in games programming > much easier to implement SFML in C++ than OpenGL or other libraries.
- Since I am generating a 2D image SFML allows me to do this since it supports 2D drawing.
- In addition, it supports multi-threaded drawing so my second task became quite simple because it was quite easy to draw to the screen.

Conclusion

- My thoughts on the farming pattern?
- So, it is clear from the results that with my current design of the application, the Mandelbrot benefits from using 4 threads and 1 render thread. This avoids having to use 8 threads to achieve almost similar results than that of 4 threads.
- This is another method of implementation that demonstrates the benefits of parallelism in the Mandelbrot set, and threading in my application.
- The render task was quite simple as a new thread wasn't created every time a user wished to use the arrow keys. Since this was primarily GPU based, it didn't have a dramatic effect on the performance of my program. It allowed for a powerful task to be executed without causing much effect on my program.
- It is recommended that rendering and the event loop be kept in the same thread to make them easier to handle and could affect performance time. Here, they were kept in separate threads. Although this did not impact the performance.