TDT4200 - Task 3

Daniel Romanich

October 2018

1 Threads

1.1 Escape time algorithm

The initial results of rendering the set without any parallelization was 6100ms on average. When I implemented rendering in parallel with 8 threads, the average run time ended on around 1700ms. This means we got a speedup of around 3.6x.

1.2 Marani silver first touch

Running the Marani silver without any optimizations resulted in an average run-time of 1900ms. When I implemented the parallel border checking it had an average run-time of 1060ms. This is a speedup of around 1.8x. Given that we are only using the same parameters in the first task, a smaller start blockDim and a smaller subDiv is improving its performance; this is because a lot of the screen is covered by the set, which means a large starting square would have a very high probability of having its borders "breached". With a larger resolution the speed is slowed down (however, we can tweek the subDiv/blockDim to reduce the slowdown, which is explained in the next task).

1.3 Marani Silver

If I run the parallelized Marani Silver algorithm (with 8 threads) in conjunction with the threaded common border solution, the maximum amount of threads running at once will be 8 + 4 * 8 = 40. This means that for every thread we run the Marani Silver on, we will get 4 additional threads every time we need to check the borders. This is not very ideal, since at some point the overhead from the threads will overpower the speedup gained from parallelizing the border checking. This ends up slowing down the program. It is not directly dependent on the application parameters; this is because we will never have more than 40 threads at the time, regardless of what parameters we have. The Operating

System ends up wasting a lot of time on context switching (overhead) between the threads when they ramp up a lot.

In general, it is hard to tell how the blockDim/subDiv affects a render with random parameters. It is entirely depent on how much of the screen is covered by the set/black. This makes sense, as a picture with little of the set on it would benefit a lot from a large starting blockDim, while a picture with a lot of the set on it would benefit from a smaller starting blockDim. It is connected to the output resolution by the fact that a larger resoltuion gives more pixels with the set and with black. Therefore, a good set for a lower resolution with the same zoom/pos will work worse for a higher resolution because it will find too small squares at the time. I did a test by having some good initial values for blockDim/subDiv in res=1024, then a test when doubling blockDim/subDiv. Doubling the values resulted in a slow-down. Then I increased the resolution to 2048, using the two same blockDim/subDiv tests, and the speedup was opposite. This confirms my speculations above.

The position and zoom absolutely play a part in what subDiv and blockDim that is good (this explaination goes under how the blockDim/subDiv affects the render). This is because these parameters decides the how much of the screen is covered with the set.

It would be very hard (impossible?) to find optimal parameters for all use cases, as explained in the two examples above. The different zoom/offset values warrants different blockDim and subDim values to be optimized, and knowledge of how much of the screen is covered by the set is therefore needed to optimize the values for all parameters.

2 Producer & Consumer

2.1 Create a queue

In comparison to the recursive call solution this could be seen as a "breadth first" algorithm, while the recursive is depth first. Since we are using a queue, we will always execute all of the new jobs added at one step, before we execute the deeper steps.

2.2 Condition variable & mutex

This step is really important in order to get a proper output & no deadlocks. If we do not define a critical section to adding and removing from the queue we **WILL** at some point get a race condition. A race condition can cause many problems, including loss of data. If two threads adds to the queue at the same time, one will write over the result of another causing data to be lost. Inproper usage of mutexes/condition variables can cause deadlocks, since it is possible for every thread to be waiting for the resource. I ended up having a deadlock in my initial code draft since I was using the notify function at the wrong time (This

happened after I did the next task). This caused every thread to hang if the queue was empty and only one thread was running the marani silver function.

2.3 Worker threads

Running the marani silver in parallel with the same parameters as in task 1 the average run-time was 340ms. As mentioned the unoptimized averaged to 1900ms. This is a speedup of about 5.5x. This being faster than the approach in task 1 makes perfect sense. In task one the workload is not balanced well between the threads, because it depends on which part of the image they get. In the consumer/producer approach every thread should have about the same workload, allowing the program to run even faster (It is also faster because it does not ramp up nearly as many threads as the solution in task 1).

3 OpenMP

The reason to why the output is all messed up after implementing OpenMP is due to how the pixels are saved in the image. Saving to the frame buffer causes race conditions. The frame buffer saving is done by a pointer that is incremented for every color (4 times for each pixel), and when the incrementations are done at the same time, the different threads are writing on top of eachother. I solved this by calcualting the actual buffer offset for each thread instead of incrementing. The result from running the OpenMP with the same parameters averaged around 700ms. Compared to the multithread (queue based) average of 340ms and single of 1900ms, it is clear that it is not as efficient as the multithreaded approach. The speedup from singlethead to OpenMP is about 2.7x, while the multithread is about 2x faster than OpenMP. This also makes sense, as the OpenMP solution suffers from the same problem as the multithread apporach in the first task. It is not able to distribute the workload in an efficient manner causing it to lose some performance.