Assignment 2 - Implementation Report

October 7, 2019

1 Relevant concepts

Stack vs heap allocation
When possible, we intend to use stack allocation due to its speed.
However, since the stack is significantly limited in size, we recognize that we will have to store larger objects, such as our image matrices, on the heap.

Locality
We intend to write our images row by row, and thus storing our matrices in row-major order is an obvious way to improve spatial locality and thus reduce the time required to fetch the image data.

Threads
The use of threads are a mandatory part of the assignment and has obvious improvements on speed. We use one thread for writing and d threads for Newton iterations. We are careful not to make any unnecessary computations between thread_lock and thread_unlock for the mutexes.

File streams
As mentioned in the assignment, we will favor fwrite over fprintf when writing our files. We will also attempt to minimize the number of calls to fwrite, by writing large blocks of data at a time.

GCC optimization
Utilizing compiler optimization options can yield a major improvement in runtime. This improvement may sometimes come at costs such as higher memory usage, but since our only requirement is a sufficiently fast execution, we do not need to consider such tradeoffs.

2 Intended program layout

As stated in the assignment, the task necessarily splits into two independent tasks; writing to files, and Newton iterations. These two tasks will be performed by separate tasks, and while the writing will only be performed by a single thread (due to the difficulty of writing to a single file from multiple threads), the computing can be done on multiple threads due to its independent nature. Global variables to store computation results and other relevant data will be created. These threads will largely follow the structure given in the lecture on pthreads.

2.1 Writing thread

This thread will write our image files, parsing one row at a time. When the next row in order has been computed, it will read this row, then loop over the row and write a string with RGB-values corresponding to the computed values. The same method will apply for both images.

Strings of each RGB-value needed will be precomputed, so that we can effectively memcpy each RGB-value into the output strings. We are allowed to cap our grey values to 50, so we will only need 51 different grey RGB-values that we format and store using sprintf beforehand. For degree n we will only need $n+1$ different color values, so we will simply hardcode these.

So far, we have achieved runtimes well within limits by initializing strings on the stack, to contain a full row, then for each element in a computed row, using memcpy to copy the corresponding RGB string into its correct slot in the string, and using fwrite inbetween each row. Calling fwrite on a string on the stack yielded much faster runtimes than storing the strings on the heap.

2.2 Computing threads

Each computing thread will compute rows that are multiples of its ID, and by assigning consecutive IDs, we make sure that the sets of rows per thread are disjoint. Each compute thread will scale pixel coordinates to complex numbers, then perform Newton iterations and store the results in global variables.

3 Computational optimization

We intend to optimize the program in the computational aspects of the program in addition to parallelization. This will be discussed in this section.

3.1 Precompute where possible

By reducing the expressions in the Newton iteration, we can minimize the number of floating point operations required by precomputing values. This step is obvious, but has meaningful impact on the performance, as we can more than halve the number of operations needed by simplifying.

3.2 Alternate complex power function

A naive way to compute $z^n,\ z \in \mathbb{C}$ is to simply multiply z by itself n times, but this is very inefficient. One can also convert z to polar form and simply multiply the exponent by n. However, the latter requires the use of both the argument function and then sine/cosine to get the result. For small values of n this will take longer time.

A simple way to deal with this is to only compute a power of z once, and use it to compute higher powers. We thus compute z^n in $\mathcal{O}(\log n)$.

3.3 Avoiding complex absolute value

The most critical part of taking both real and complex absolute value is the slowness of the square root function. This is simply avoided by squaring the desired inequalities and convergence criterions.