How Threading can Improve Image Generation

Anthony Vardaro

Experiment

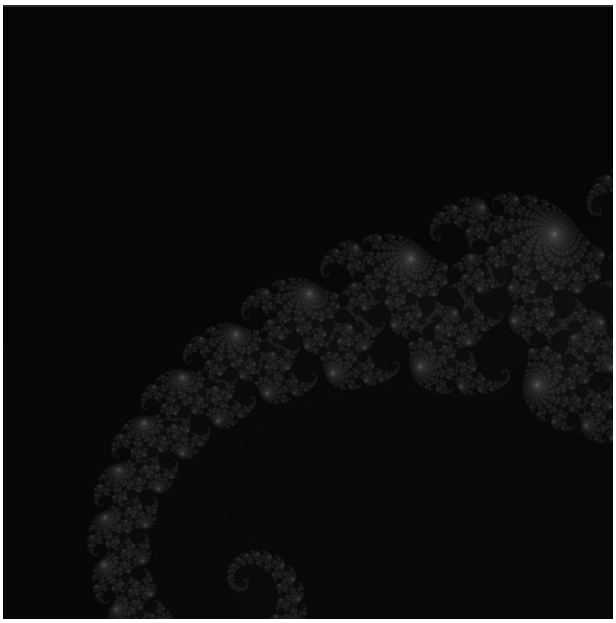
The purpose of this experiment is to observe how including threading functionality into a normally non-threaded program can impact the performance. Given a program that normally produces the image of a Mandelbrot set, we want to modify the program to divide the labor of algorithm into chunks, and observe the difference.

In this experiment, we provide the following commands to the `mandel.c`

`./mandel -x -.286932 -y .014287 -s .000055 -H 3000 -W 3000`

The original program takes about **7s** to produce the image, whereas the threaded implementation can produce the same result with **2** threads in only **5s.** A whopping **29%** improvement! If we increase the number of threads to **3**, we can produce the image in **4s**!

However, after **3** threads we begin to hit the point of diminishing returns, where increasing the number of threads will have a negligible impact on the speed. This is likely attributed to the overhead associated with spawning a lot of spreads and also linearly increasing the space complexity of the program.

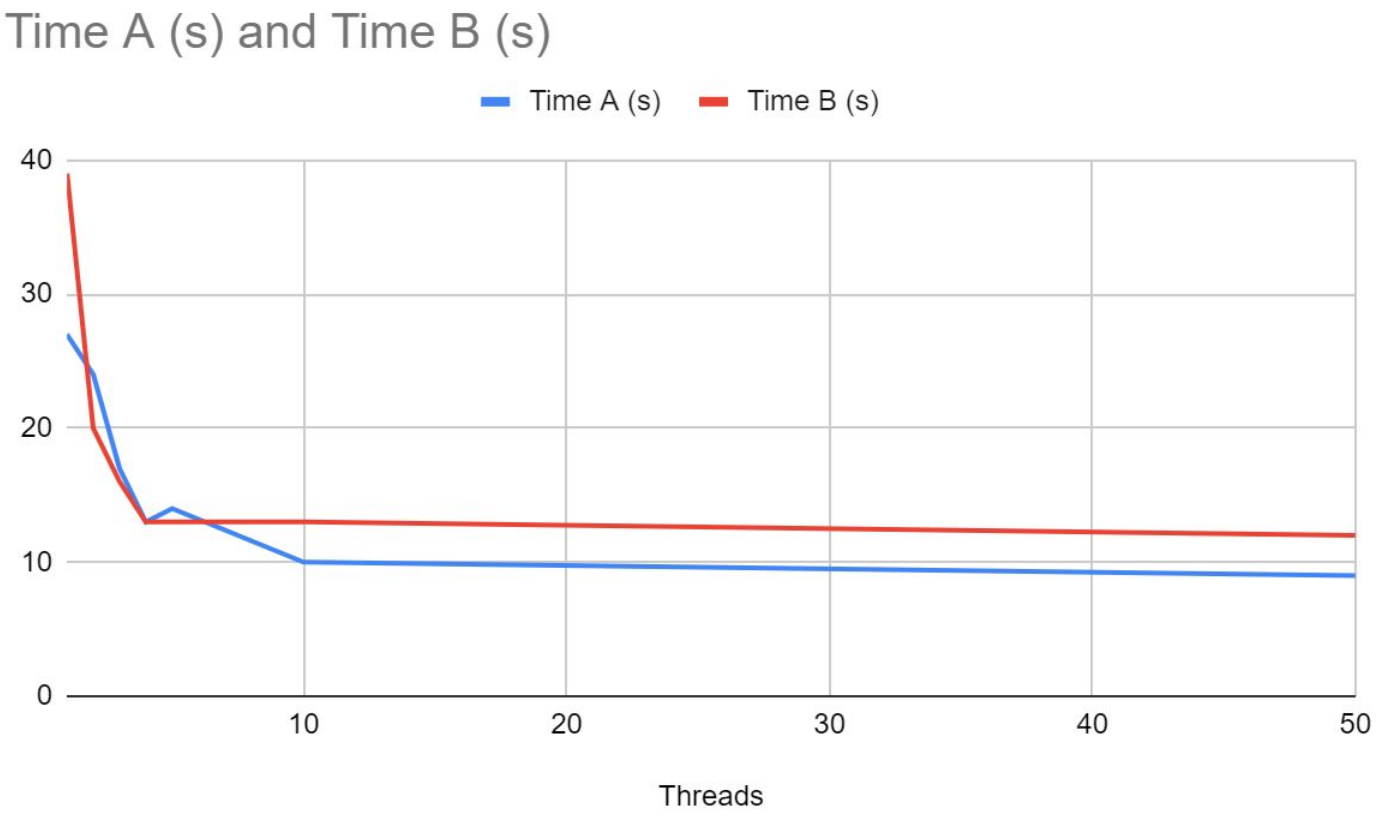


Implementation

To improve the original program, I opted to use the `pthreads` library, because I wanted to ensure that other people running the code would be able to do so with minimal configuration.

The new `mandel.c` enables the user to dynamically spawn **n** number of threads for the image generation through the command line. The program divides the work by dividing the height of the image by **n**, and allocating a whole thread to each “chunk” of the image.

Sample Data



Conclusion

As evident in our experiments, threading is powerful and performant up until you reach that point of diminishing returns. Eventually you begin to spawn so many threads that the individual impact of each thread is negligible and even something harmful. From our experiments, the optimal number of threads for this use case is roughly **4** or **5** threads total**.**