Jacobi’s Algorithm

Mason Pratz
Thomas Jones-Moore

Implementation of Jacobi's algorithm using concurrency allows us to further understand multithreading and recursive behavior with threads. This implementation of Jacobi's required our own version of a pthread_barrier with the use of primitive semaphores. Experimenting with a different amount of threads was in this experiment to understand the benefits of threads. Results of this experiment yielded rudimentary data that we were mostly able to predict from the beginning.

Jacobi's algorithm was implemented by using custom structures, pthread, a recursive function, and barriers. The first issue we had to tackle was setting up the two 'matrices', current_matrix and next_matrix. These two objects keep track of the data values for each iteration respectively. At first there were big memory problems because the 'current_matrix' was being thrown away (and free'd) and then was recreated with malloc. This was unnecessary, and our poor handling resulted in a memory leak. After rethinking our implementation, we came to the conclusion that instead of throwing away the old 'current_matrix', we could just wipe the values and reuse it. This not only fixed the leak, but also significantly decreased the overall program time.

The code went through two major changes. First, a misinterpretation that stemmed from how the threshold worked. Originally, we thought that each single data value's delta change in the new matrix that was under the threshold would "lock", preventing further changes. We soon realized that this was incorrect. Our code was not seeming to terminate, even on the smaller threshold experiment. This is where the first big overhaul happened. We took out everything that involved 'lock' and ran the program. This resulted in our program properly running, but we still weren't getting the desired times we wanted.

Then we completely overhauled the code. To support multithreading in the future, at first we had two main structures in the matrix: row_objects and cell_objects. We originally constructed the matrix to be filled with row_objects because we could assign specific rows to each thread to work on. We realized this was slow, and scrapped all the row_objects and scrapped these for one single object to hold the matrices: matrix_object. Now each thread will work on a row determined by the order they were created in and increment by the total thread number.

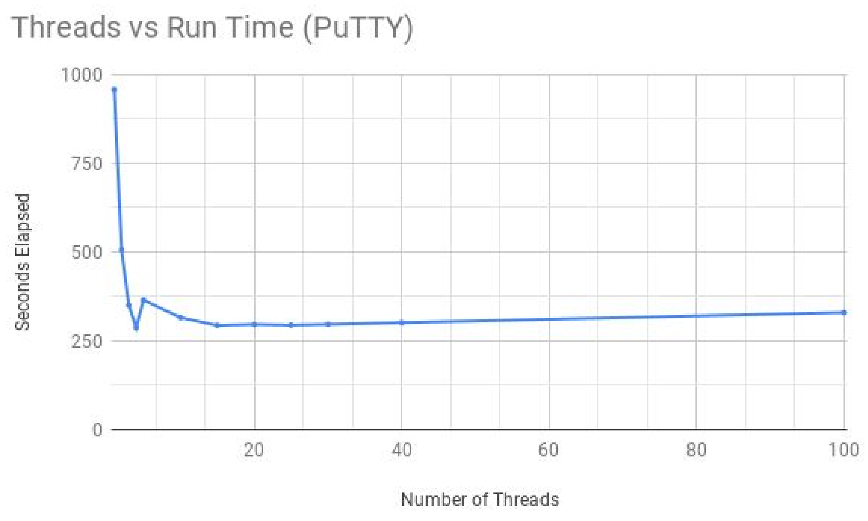
After fixing the fundamental issues and plugging the leaks, we fixed our timer. Originally we were able to properly record time when there was just 1 thread. After adding multithreading, the time was incorrect since it measured cpu clock, and we had multiple threads. To counter this, we added two global timespec structures. These would then be called at the beginning and end of our program by the very first thread only.

To split up the work we would do the core of the work together, then take home ideas and concerns to think about. If a big idea came to mind, we wouldn't be scared to write up the code at home and push to the communal gitlab repository. We also utilized texting as a form of communication allowing us to bounce ideas off of each other at ease.

A challenge we faced was synchronizing threads and creating our own custom barrier. We first made sure the code worked with the given pthread_barrier function. Using semaphores, we then made our own barrier, and replaced. The first barrier was put at the end of the work() function to collect all the threads. When released, the very first thread would check if we are done iterating. If not, then the second barrier would collect all the threads again, before recursively calling work().

In order to check the results, we conducted two main sets of tests. The first set included using PuTTY on a laptop to connect to a linux machine on WWU's cluster and test from there. The other setup was a i5-equipped windows 8 desktop running the code from its own terminal. The data was tested with 100, 40, 30, 25, 20, 15, 10, 5, 4, 3, 2, and 1 thread for both setups. Since a variable number of users were connected to WWU's cluster at different times, we based the majority of our conclusion on the data from the desktop.

The results yielded from this experiment were mostly expected, minus a couple of interesting points. We assumed from the start that the more threads we throw at it, the faster the program should run. From 1 to 20 threads this trend follows except for when using 5 threads. For some reason, using 5 threads caused the program to run for 366 seconds, versus the 4 threaded run time of 287 seconds and 10 threaded run time of 316 seconds. When more than 20 threads are given to the program, the elapsed time slowly increases. We can see this when looking at 20 threads vs 100 threads on the graph below.



The bulk of our program is the work() function. This function gets called recursively until the program is terminated. It performs the delta calculation on each iteration. This is also where our two barrier synchronization method is used. The run time for our program with 1 thread is still a little slower than Clauson's implementation. Ours ran for about 16 minutes, while Clauson's ran for around 7 minutes. We believe the reason for this difference is due to individual cell allocation and possible usage of pointer swapping.

The conclusion of our test data seems reasonable with what we predicted before starting the assignment. We believe we have still not reached the limits of the machine, and there are a small assortment of tweaks we could make to speed up the run time. In conclusion, concurrency is very powerful but it does have its limits. The performance speed vs thread count do not follow a linear curve, and instead follow more of an exponential curve (to a certain point). Multithreading Jacobi's algorithm provides insight to the limits of concurrency, machine restrictions, and big data handling.