Concurrent Implementation of Jacobi Iteration

Connor Freitas & Theo Floor
Submitted to Aran Clauson, 6/1/16

As the number of processors in modern computers increases, programmers become better able to leverage concurrent programming to solve scientific and mathematical problems. Our goal in this project was to use a parallelized Jacobi iteration algorithm to compute the solution to Laplace's equation for a large (2048x2048) matrix to within a reasonably small maximum change per iteration, $\varepsilon = 0.0001$. Jacobi iteration is one of several algorithms implementing Laplace's equation for calculating the behavior of heat, electricity, gravitational, and other potentials across a two-dimensional surface of fixed perimeter. This sort of simulation is useful across a multitude of scientific disciplines, and when used in conjunction with parallelized programming, can produce efficient modeling results from large bodies of data. Jacobi is practical for parallelization due to its relatively straightforward structure. We used a hyperthreaded, multi-core CPU to experimentally find the theoretical speed-up and portion of our implementation which was parallelizable. We wrote our code in C, utilizing the pthread and semaphore libraries to leverage our machine's ability to perform concurrent calculations. Here we also discuss results from our testing, as well as potential opportunities to further increase our concurrent program's efficiency.

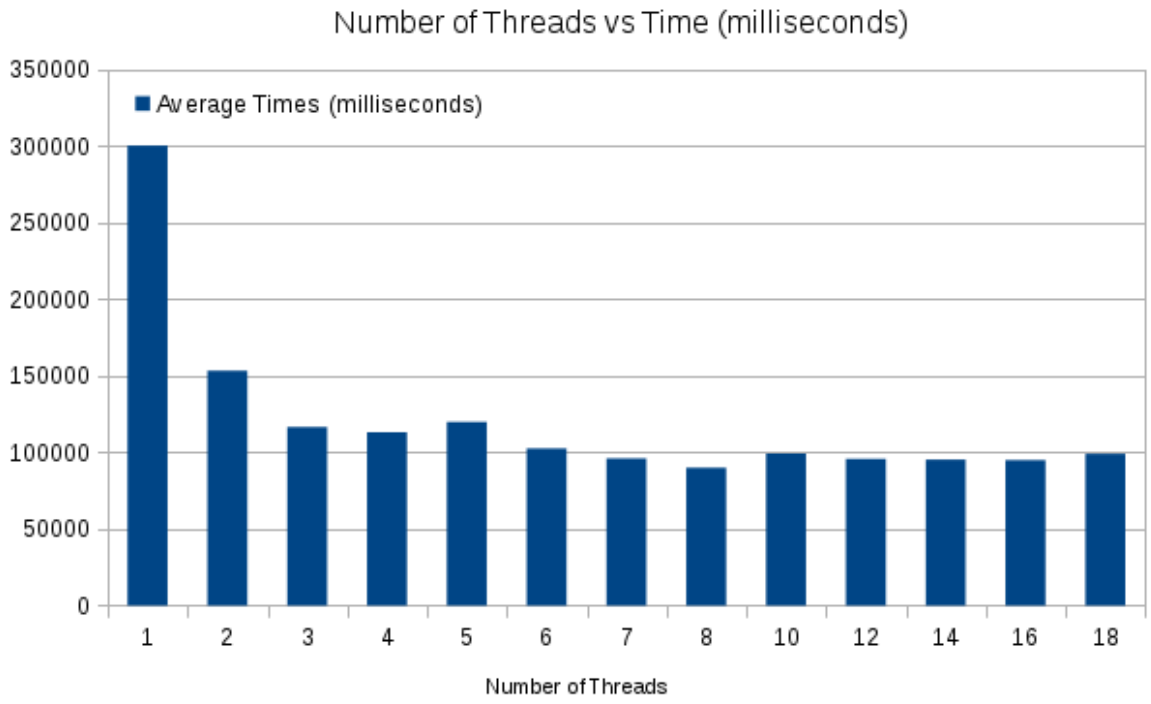
The logic of our parallel implementation revolved around dividing up the input matrix efficiently and providing a way to guarantee that each thread computes values based on the same generation of the matrix. This is analogous to computing the potential of the two-dimensional surface changing over time. We chose to use the C language because of its efficiency in reading and writing memory and in handling threads. The convenience of the pthread and semaphore libraries were also motivating factors. Our implementation of Jacobi's algorithm, **jacobi.c**, involves a main function to allocate and initialize the conditions for each thread, a helper function (executed by each thread) to compute iterations of Jacobi's algorithm, and a barrier function to ensure that all of the threads calculate exactly one generation of the algorithm at a time. Jacobi.c takes optional (positive integer) arguments to specify the number of threads to use. Each thread completes the following routine to calculate the value of its portion of the matrix for the current generation. All threads iterate through these steps until the maximum change from one generation to the next is less than the desired value of ε:

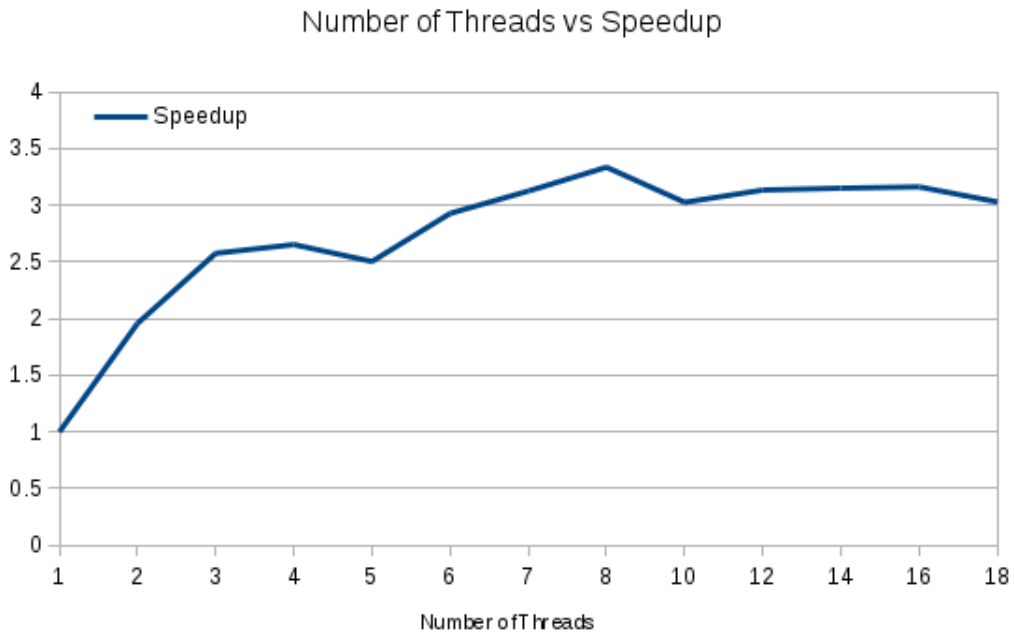
a) calculate new values of all cells in the thread's assigned portion of the matrix
b) if the thread's maximum difference over all cells between two generations is sufficiently small, notify all other threads
c) swap the addresses of the matrices corresponding to the current calculated generation and the previous generation
d) wait in a logical barrier for all threads to complete the current generation

Our barrier function was critical to keeping threads synchronized. We used an array of semaphores to capture each thread until all threads had reached the barrier. Upon entering the barrier, a thread acquires a mutex to check the number of threads already in the barrier, then releases the mutex and waits on its corresponding index in the shared semaphore array. If a thread is the last to enter the barrier, it checks all threads' states to determine whether to terminate the algorithm. It then releases all threads still waiting on the array of semaphores and continues. After termination, the final generation's matrix may be output to the console (or another file) or discarded.

To test the performance of jacobi.c, we ran it multiple times with set numbers of threads. The

machine we used to run our tests had 8 hyperthreaded logical CPU cores on 4 physical cores. Using the Linux “time” program (found in /usr/bin/time), we recorded the execution time of each run of the program. In order to gather reliable performance data, we ran 5 trials of each number of threads, 1 through 8, plus an additional 3 trials each for 10, 12, 14, 16, and 18 threads. These test runs were all performed on the same machine, after a reboot, back-to-back over a period of approximately 2 hours. To compile jacobi.c, we used the standard gcc C compiler with no optimization features active.





Top: average run times of jacobi.c over 5 trials for increasing numbers of threads
Bottom: average speedup of run times for jacobi.c

Our results indicate that jacobi.c experiences the most dramatic speed-up when incrementing the number of threads from 1 to 4. Presumably because of the efficiency of reading and writing floating point numbers across multiple cores, we saw a slight rise in execution time when increasing the number of threads from 4 to 5, with a steady decrease until the overall best running time, achieved with 8 threads. With more than 8 threads, we can infer that the overhead of additional thread creation outweighs any gains that might have been had by having more workers. Our test data indicate that there is an almost exactly linear speed-up from 1 thread to 2, and from 2 to 3. According to Amdahl's Law, approximately 70% of jacobi.c is parallelizable:

$$\text{Theoretical speedup} = \frac{T_1}{T_n} = \frac{300384\,ms}{90020\,ms} = 3.34$$

$$\text{Parallelizable portion of code } p = 1 - \frac{1}{speedup} = 1 - \frac{1}{3.34} = 0.70$$

Since the majority of the work in jacobi.c is simply matrix calculation, one would assume that a significant portion of the program is parallelizable. Our calculations of Amdahl's Law above seem to confirm this. Non-parallelized portions of jacobi.c include thread creation, thread initialization, barrier logic (a critical section of the code), and the initial reading of the input data. For large datasets, concurrency certainly provides ample gains over simple serial execution. It would seem that jacobi.c experiences some kind of optimum in performance when using 4 threads. We can reasonably assume that this is because the machines on which the program was run have 4 physical cores, and so are ideally suited to performing up to 4 synchronous tasks. Similar to the loss of efficiency experienced by some concurrent programs when moving from 8 to 9 threads, we observed a slight decrease in efficiency when moving from 4 threads to 5. We speculate that one of the limitations of this particular processor architecture for a concurrent program such as jacobi.c could be the reduced number of available floating point registers. If the program were to operate solely on integers, it might experience a much greater speed-up, since many processors have more dedicated integer registers. Similarly, it could be the case that two hyperthreaded logical cores share floating point resources, whereas even two hyperthreaded logical cores on the same physical core could have separate integer resources. However, it is certainly true that there are worthwhile gains in execution time on the part of jacobi.c and other concurrent programs when programmers can intelligently make use of multithreaded machines.