Ryan Wendling
322 Concurrency Project
Spring 2016 Clauson

Jacobi Method and Concurrency

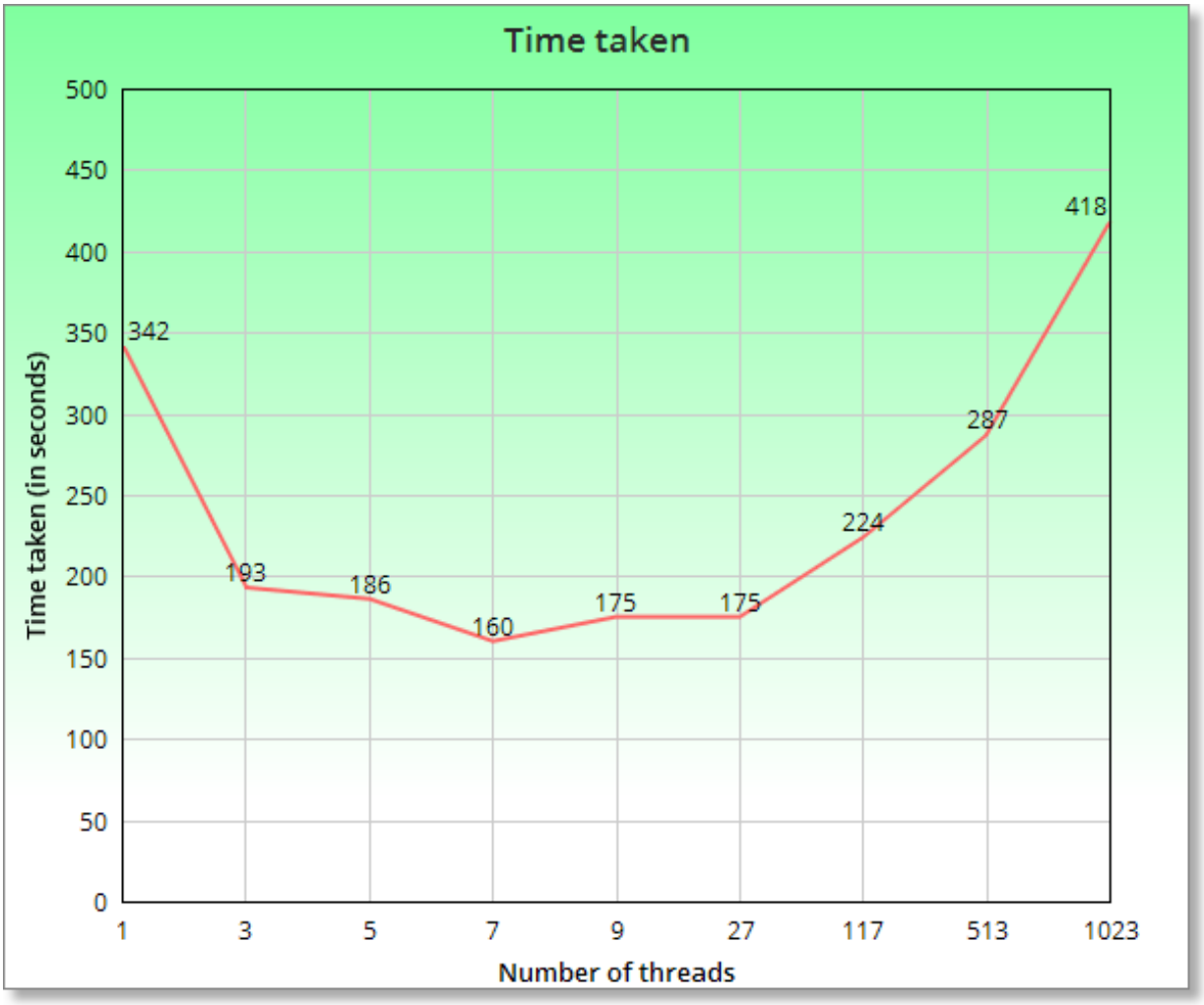
The Jacobi method is important because it gives us the ability to logically find the values of unknown cells, given a limited number of values. This is done through repeated iterations until the new numbers seem stable. The Jacobi method has real world applications like for estimating the temperature on any part of a metal plate if it is being heated in some area along the perimeter. Additionally, to help implement the Jacobi method and other important mathematical algorithms, we can use concurrency. Concurrency speeds up our computation times by getting multiple processors involved to each work on a different part of the problem. A useful feature, because no one wants to stare at a screen all day waiting for absurdly large problems to be computed.

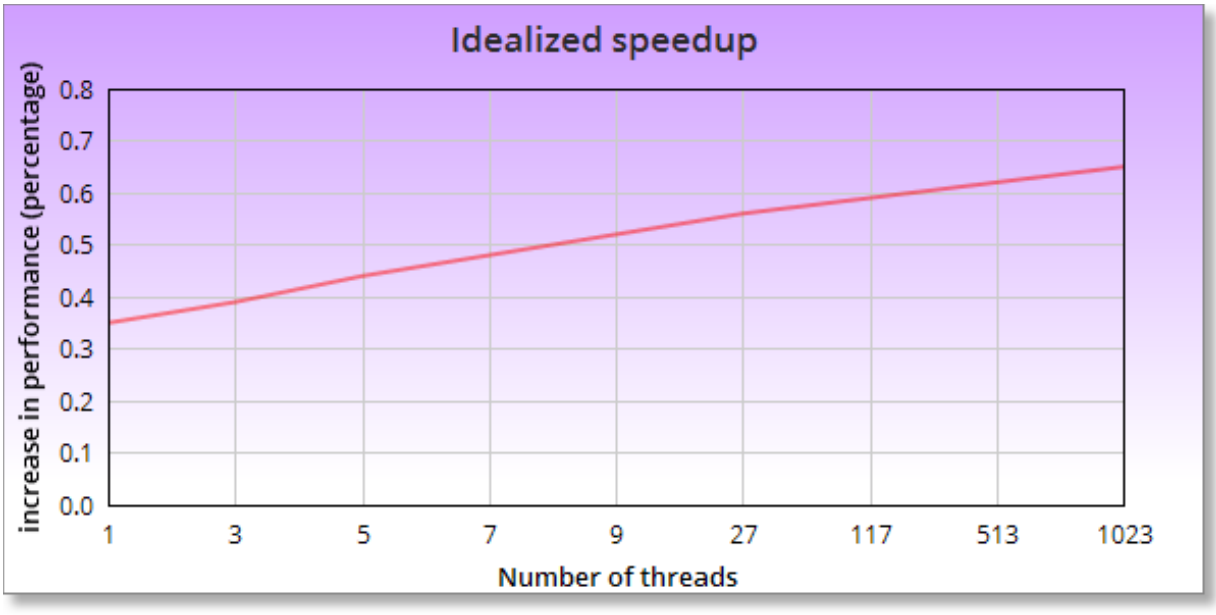
These topics- Jacobi iteration and concurrency, are the center point of this paper. With that in mind, we will delve into an instance of Jacobi iteration implemented with concurrency. To measure the success of this instance, many trials were performed with varying threads and data sizes. After the various experiments, the results were logged, and will be discussed.

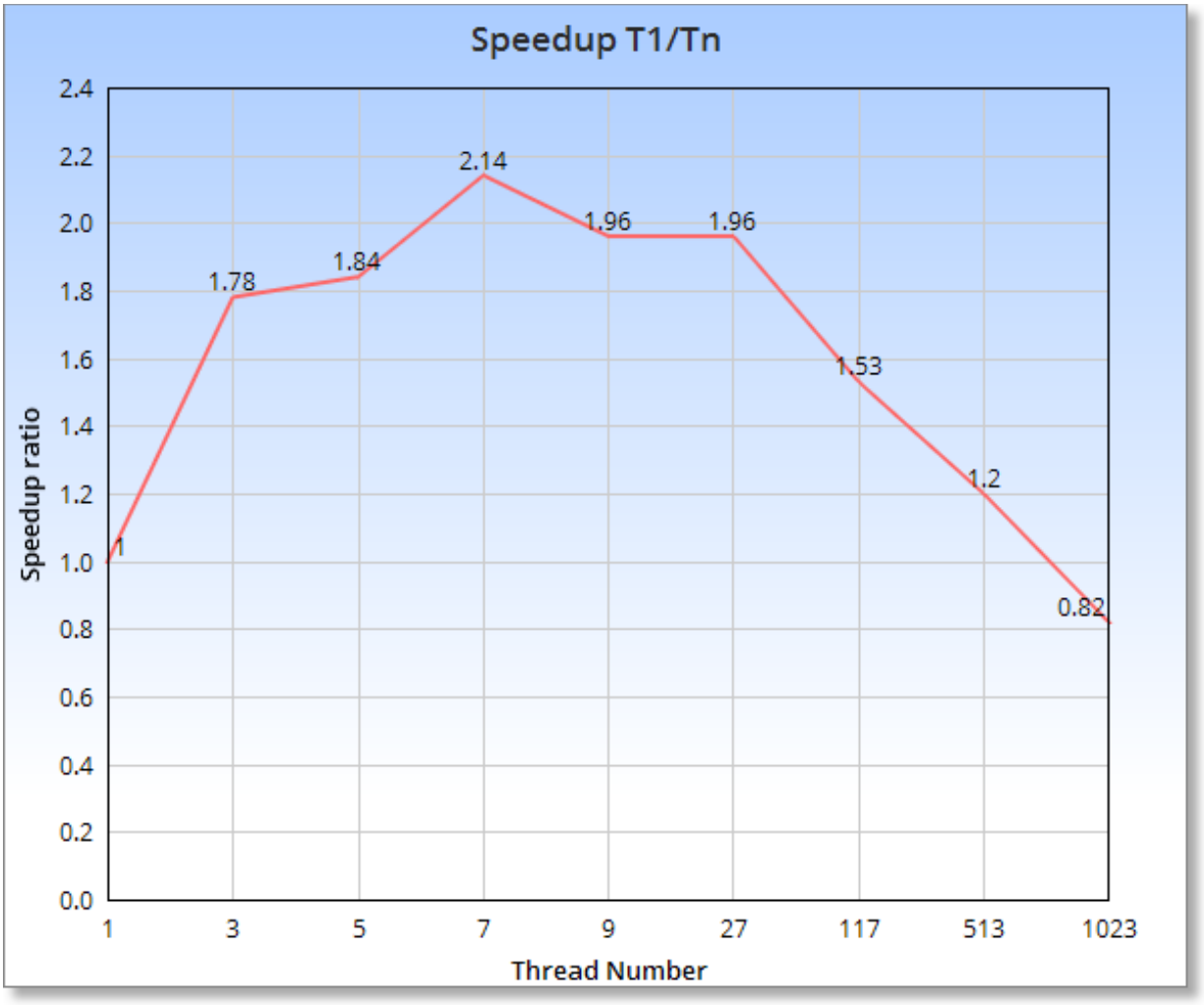
To implement the Jacobi algorithm, the Java programming language was used, along with a variable number of threads, double arrays, for loops, and dissemination barriers. For this illustration of the Jacobi method, we are given a 2048 x 2048 matrix of values, where only the perimeter values are populated. We are to make a program that bleeds in the values to the interior cells. To emulate Jacobi's many iterations, we use a while loop that keeps running so long as there is significant enough change to the values of the interior values. To keep the threads aligned, and working on the same iteration of data, we used a dissemination barrier. As we know, barriers make it so all threads have to wait until any straggler threads catch up to the same point and are then released at the same time. The benefits of this particular barrier is that it is a symmetric type, so that it does well when each thread performs a similar task and will all be ready in a similar time frame. It also scales well to any number of threads, both even or odd. With the program designed successfully and giving values similar to the output.mtx file, we set the stage for some experiments to see the speed up we can achieve.

To measure execution time of these experiments, we use the “System.nanoTime()” call once at the beginning of the program and again at the end. This creates an internal timer, measured in nanoseconds. The program was ran at least three times for each of the following thread amounts: 1, 3, 5, 7, 9, 27, 117, 513, and 1023. The computer used to run the simulations are the school computers in

lab 162. I believe that the they have seven cores. Jgrasp was used to run the program. We were also given a 64x64 sized matrix to be used to check our output values in a more efficient manner. The experiments mostly consist of waiting around however, it is the results that give the real value.







The results achieved seemed valid and follow a logical order. When run with only one thread, it takes about 342 seconds on average. When run with three to seven threads the time is much quicker and reaches max efficiency at seven threads. From 9 threads onward, the time taken slowly increases until it becomes even slower than having one thread. This is mostly due to the increased overhead that concurrency creates along with the additional stages in the dissemination barrier checker. Above is a graph of our speedup (calculated using Amdahl's law) , a graph of our overall time for each thread count (times rounded to the nearest second), and a graph of our idealized time speedup for the parallelized portion of the program.

After sifting through the results, we can deduce the concurrency is indeed beneficial to the speedup of the program because we get the same results, in terms of the matrix output, whilst taking less time. For example, going from one to three threads produces the largest decrement in time taken by about 44 percent. A lot of the program can be computed in parallel, but there are things that are computed serially. This includes the initial variable declarations and in my program's case, the initial filling of the input array that is being read from the input.mtx file. The parallelization was mainly used

in dividing up the for loops, so each thread only worked with a fraction of the matrix's values. This included reading the data and calculating the values for the next iteration of the matrix. Also parallelized was the calculating of the max differences(between the old value to the new value) . Mathematically, the percentage of the program that can be run in parallel seems to be .65. This was deduced using Amdahl's law, as using this value for p, we can get speedup values similar to those shown on the graph that were calculated by dividing the time taken for one thread divided by the time taken by n number of threads. It is interesting that the Jacobi iteration doesn't fully follow the logic of Amdahl's law; The law of diminishing returns holds true, but once we use more than seven threads, the speedup actually gets worse. In the ideal case, the program would keep speeding up linearly for the parallelized section of the program. But in reality this was not the case, and the green graph is not an accurate representation of the actual performance.

We can conclude that Jacobi iteration is a valid mathematical process as we are now left with an entirely filled in final matrix, it as also scalable to other input data sizes and will adapt to varying perimeter values as well. Concurrency is also shown to be quite beneficial and also very processor dependent. It works best when the threads match the amount of processors, otherwise we have processors that aren't being utilized fully or are insufficient and each processor has to switch between threads and that will incur additional overhead.