# An Analysis of Multi-threading via Jacobi Iteration

By: Nat Choeypant and Jason Lacy

**Introduction:**

Efficiency is quickly becoming one of the most desirable traits in technological programs due to the ever-increasing demand for bigger and bigger data. Over the last two years alone, we have generated about 90% of the total data that has ever existed on the planet earth. A mind-boggling 2.5 quintillion bytes of data are created each day, and this number is only getting larger (Forbes.com).

In this age of Big Data, users do not care how we retrieve their Facebook status updates; they care only care that its reliable, secure, and accessible. They don’t care how many softball games the programmer missed trying to finish his SQL queries if Snapchat decides to crash mid-selfie because an overworked thread couldn’t handle the pressure. We, as programmers, are here to solve problems. We must go beyond the call of duty when creating a program that technologically illiterate users will utilize daily. With the ever-decreasing attention spans of today’s youth, algorithm speed and data efficiency are almost as important as getting the correct output. These are the reasons why we need to learn how to write programs that function quickly and correctly. Cost-effective multi-threading gives us the tools we need to increase the speeds at which we compute. We need to think past what is thought of and continue to innovate in every area imaginable.

**Implementation:**

For our Jacobi implementation, we iterated through the inner cell starting from the top inner row to the last inner row. For each iteration, our program assigned each inner cell’s value by calculating the average of adjacent cells’ values. The program kept iterating until every inner cell had reached a threshold of ε = 0.00001. We split the work for each thread by splitting the matrix into sub-matrices by horizontal lines. The number of sub-matrices to be iterated is equal to the number of threads. Our program assigned each thread to each sub-matrix for the thread to iterate through.

Our program used pthread\_barrier\_t as our barrier to synchronize our threads. We initialized the barrier to the number of our threads, so that our barrier can block the thread at the barrier until every thread has reached the barrier. We assigned the barrier before iterating through matrix N, another barrier before estimating updating matrix M, and another barrier after updating matrix M to synchronize all the threads before the next iteration.

**Experiment:**

In our experiment, we disregarded the time it takes to read input file to produce two 1024x1024 matrices. We used *clock\_gettime* to get timestamps before creating threads and after all the threads finished Jacobi iteration, then we calculated the amount of time by finding the difference in two timestamps.

We measured the amount of time for each number of thread it took to finish Jacobi iteration. We tested each number of thread four times, then used the average running time for each number of thread as our data. Each of us tested our program via 405 lab computers twice, one computer running nothing but the program and another one running other applications while testing the program. For some reasons we could not compiled thread program via MakeFile, so we compiled the program with gcc compiler in our shell with command: *gcc -o thread thread.c -lpthread*.

**Results:**

Blah

**Conclusion:**

Blah