Jeremy Spiro, Ravi Jain, rjain8@u.rochester.edu, CSC 254, MW 10:25-11:40, Prof. Michael Scott

In this project, we were tasked with parallelizing a sequential program using code provided by Michael Scott. We compared the Delta stepping algorithm and Dijkstra's algorithm on a graph of x vertices as an exercise of evaluating the possibilities when utilizing concurrency and multi-threading. Delta stepping involves an array of buckets where any given bucket holds the vertices whose currently best known paths have the lengths $i \times \Delta \leq l < (i+1) \times \Delta$ (at the beginning, only the source is present in the array, in bucket 0).

Our implementation involved the expansion of Scott's code for the use of multiple threads. We established that, regardless of the number of threads, there should be five locks in order to run concurrent delta-stepping efficiently. These barriers were:

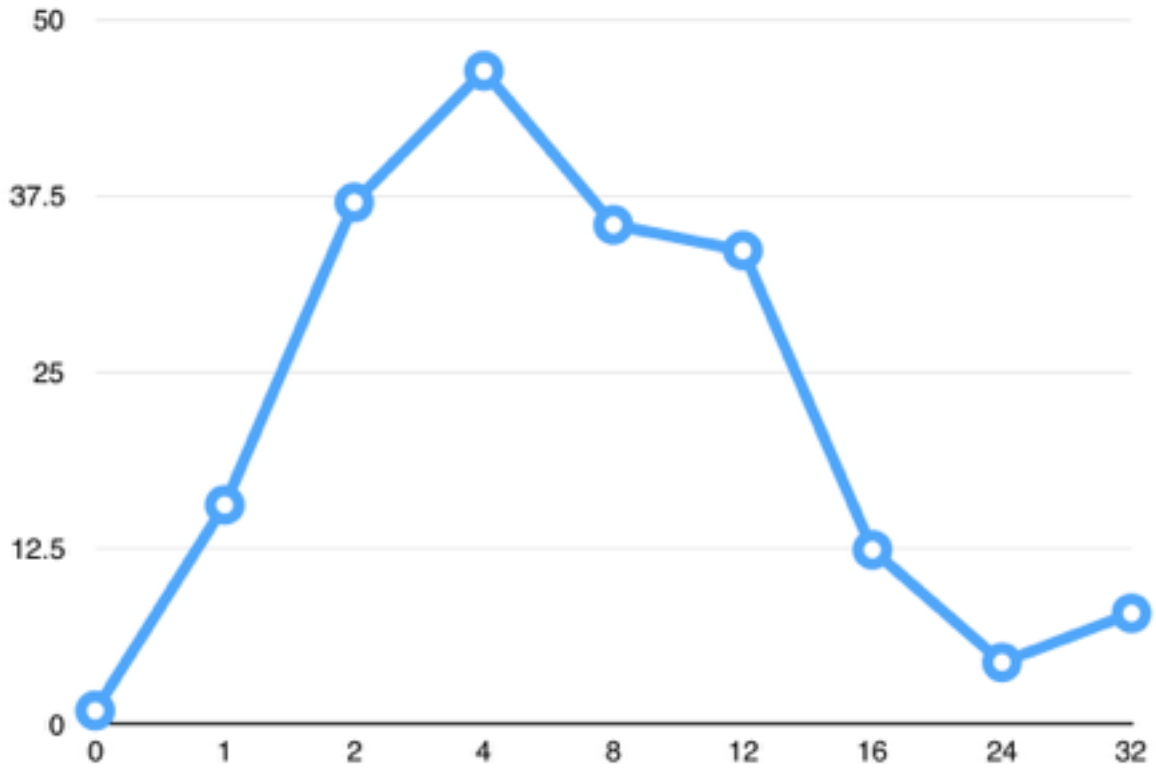
1) light relaxations
2) messages sent to threads concerning light relaxations
3) heavy relaxations
4) messages sent to threads concerning heavy relaxations
5) a final barrier that ensures that all threads have completed their tasks

The first four barriers were implemented using the cyclic barrier class in Java; these were also implemented with a runner to specify conditions for each respective barrier. The final barrier that ensured the completion of all threads was implemented with the join subroutine for threads. This subroutine acts as a lock that prevents the threads from moving forward until all threads have been joined together.

Initially, we were concerned about preserving threads, so we implemented our algorithm with a specific thread class in mind. As the project went along, we discovered that this was not absolutely necessary, but our code was too deeply integrated to change the structure of our code. Additionally, we had a list of booleans to signify whether or not a thread had completed all of its tasks. It appeared to be rather inefficient, but no other viable option presented itself. We predicted that this may have a negative effect on the program as the number of vertices existed.

In the end, our parallel program was still slower than Djikstra’s algorithm, even when running on multiple cores with several threads. We suspect that most of this is due to bottlenecks in our code, especially around barriers ending sections where a minority of threads have to handle the majority of the work.

Even though our results weren’t ideal, as the number of threads increased the runtime of the program started going down, though there was a jump initially where a small number of threads would result in higher execution times (probably due to the impact of bottlenecks being more severe at those points). While the peak is at 4 threads, the runtime begins to decrease once 8 threads is reached, and continues to drop after that as more threads are added.


50
37.5
25
12.5
0
0
1
2
4
8
12
16
24
32

x-axis: number of threads (0-djikstra)
y-axis: time of execution in seconds

x-axis: number of threads (0=dijkstra)
y-axis: times slower the parallel version was than dijkstra (ex. if y=15, parallel program was 15x slower than djikstra)