Parallelization of the PageRank Algorithm

Williams College – Computer Science 338: Parallel Processing – Fall 2019

Background

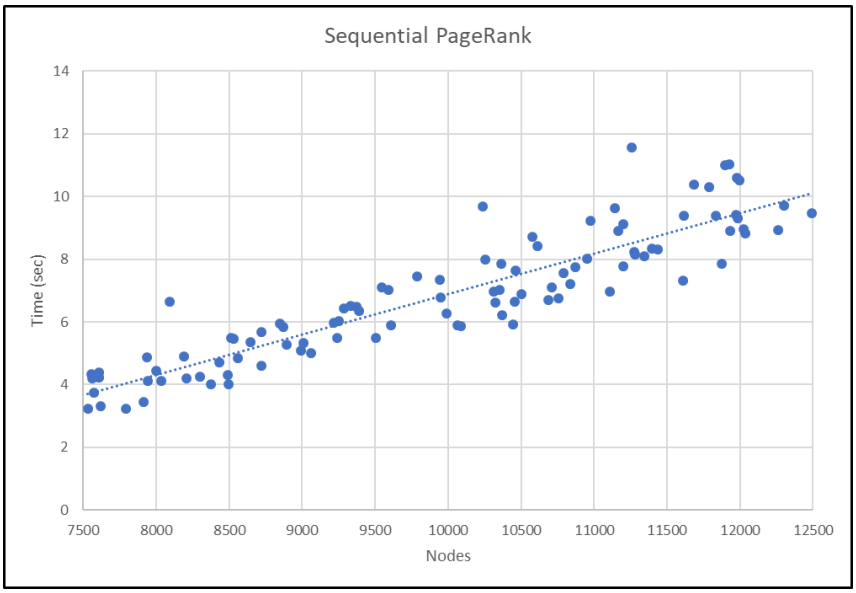
PageRank is a well-studied algorithm in Computer Science used to generate ordinal node rankings in a graph based on connectivity. The algorithm was developed as a research project in 1996 at Stanford by Larry Page, who later expanded on this work with Sergey Brin. The resulting product eventually became the driving force in Page and Brin's founding of Google, where PageRank was used as the original algorithm for ordering web search results– with the intuition that more highly connected webpages were more relevant than other webpages. The algorithm also accounts for the importance of the webpages connecting to a given webpage– to avoid search result manipulation. This leads to an iterative or generational computation model where we can estimate the "true" PageRank results by giving each node a default importance value, and then use the edges to "pump" importance along outgoing edges into receiving nodes. The error from the "true" result decreases dramatically with each iteration, and execution stops once an arbitrary error invariant threshold (which is "good enough") is reached.

Goals Achieved

The goal of this project was to parallelize the PageRank algorithm in C. I began with a sequential version of the algorithm. I proceeded by creating four parallelized versions of the code. Next, I found the best of those four versions, and implement various optimizations to that versions. Finally, I tested these results in a multi-threaded environment against the sequential version of the algorithm. I also created a program to generate graphs of various sizes and connectivity-levels to test these algorithms on. All my PageRank algorithm graphs are represented by an array of nodes and an adjacency matrix of edge connections. My algorithm also uses a dampening factor which is used to simulate "random" web-surfing. The intuition is that not every web access is sequential; there is always some amount of "random" browsing on the web. My algorithm, in line with standard PageRank implementations, uses a sequential-access ratio of .85, with a random-access ratio of .15.

Sequential PageRank

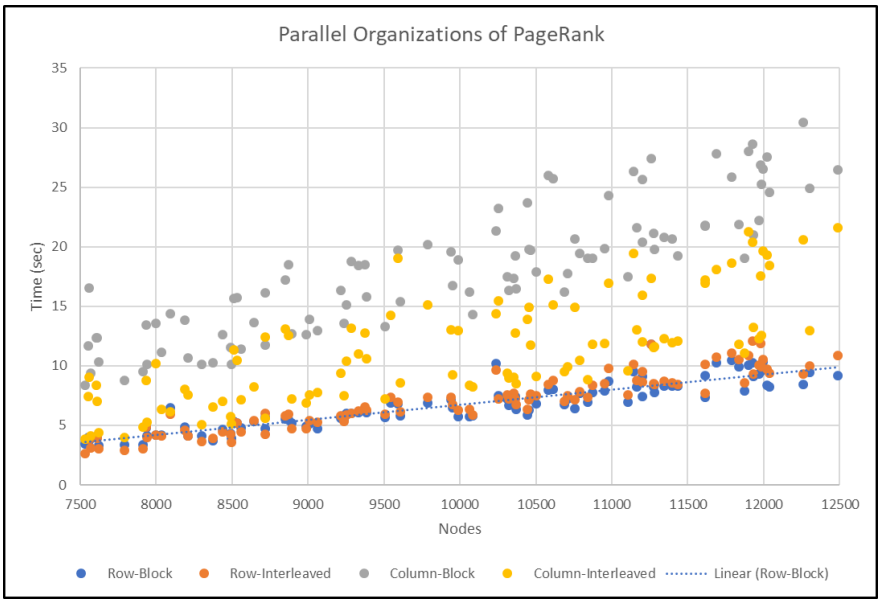
The first portion of the project was to implement a sequential version of the algorithm. The algorithm begins by reading in all the nodes and edges from an input file and creating the data



structures necessary to run the algorithm. The sequential version does not divide any work, so it simply iterates over the adjacency matrix node-by-node to calculate the updated weight of a node, based on the random-access ratio, the original weight of its incoming-neighbors, and the number of outgoing-neighbors each of those incoming-neighbors has. A graph charting the results of this algorithm is included here. The average execution time over these tests was 6.844 seconds and rose steadily as the number of nodes and edges in the graph increased.

Parallel Organizations of PageRank

I created four parallel organizations of the PageRank algorithm. The differences in organization relate to how nodes and connections and divided amongst threads. The "row-block" organization divides the graph by receiving-nodes and gives each thread a sequential chunk of nodes from the graph to operate on each generation. The "row-interleaved" organization divides the graph by receiving-nodes, but splits the nodes in an interleaved way instead of into chunks (e.g. Thread 0 may receive nodes 0, 8, 16, …, etc. instead of receiving nodes 0, 1, 2, …, etc.). The "column-block" organization divides the graph by outgoing-nodes and gives each thread a sequential chunk of nodes from the graph to operate on each generation. The "column-interleaved" organization divides the graph by outgoing-nodes but splits the nodes in an interleaved way instead of into chunks. The primary result observed from these organization is that the row-order organizations are more efficient than the column-order organizations. This difference likely occurs for two reasons. First, the memory accesses in column-order do not benefit from locality, as arrays are stored in row-major order. Second, and more importantly, is the column-organization requires an increased number of locks. This is true because every thread in the column organization could be trying to update the new weight of the same node at the same time, so we need to lock nodes during every update. In the row-order organizations each thread uniquely operates on a set of receiving-nodes, so the nodes never need to be locked before updating, as there won't be any such race conditions or conflicts. A graph charting the results of these algorithms is included blow. We observe that the row-block organization has the best



overall performance. The average execution time of the: row-block organization was 6.69 seconds, row-interleaved organization was 6.97 seconds, column-block organization was 18.61 seconds, and column-interleaved organization was 11.41 seconds. They all rose steadily as the number of nodes and edges in the graph increased. These organizations all get worse as the number of threads increase, however, due to the overhead associated with the code, and the amount of locking and barriers needed to make the algorithm function.

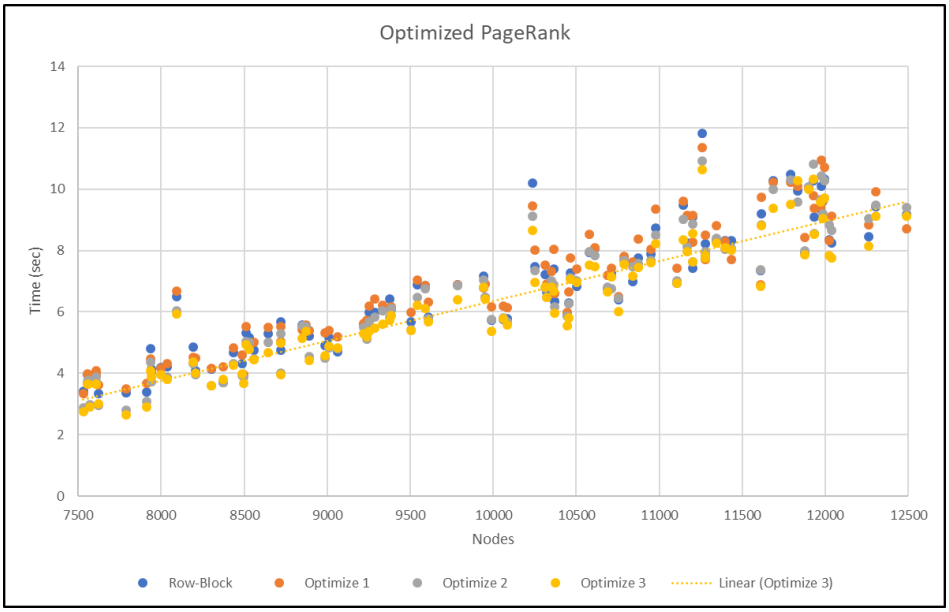
Optimizations to Row-Block PageRank

I proceeded by implementing several optimizations to the best parallel PageRank algorithm to try to improve it. Besides a few small optimizations, there were three major optimizations. The first two major optimizations dealt with pre-calculating information for threads. The first of these optimizations was to pre-calculate the node-range associated with each thread. In the original implementation, in each loop of each iteration, the thread would calculate which set of nodes it was operating on, but this range was fixed. Instead, I moved this work out of the parallel PageRank function and created an array which each thread indexes into to determine its range of nodes. The second pre-calculation optimization was to pre-calculate the "contribution" each node would supply to its neighbors. I observed that in a dense PageRank graph, we would need to calculate this value up to n^2 times, once for each potential edge in a graph. With this optimization, we pre-calculate the value once per node, which is n times, and then can access this information from a new field stored in each node struct.

The third, and greatest individual, optimization I implemented was a total refactoring of the PageRank algorithm. The goal of this optimization was to reduce the burden and overhead of the original lock and barrier schemes. In this new version, I do as much work in parallel as possible without needing to coordinate between threads. Once all of this is done, then the threads terminate, the main process collates this work and prepares for the next parallel iterations of the program. This optimization alone yielded better outcomes than the original row-block PageRank implementation.

Finally, I created an optimized program which includes all the major optimizations outlined above.

A graph charting the results of these optimizations is included here. We observe that the first set of major optimizations performs very similarly to the original row-block PageRank. The lock and barrier refactoring optimization performed notably better than the original row-block PageRank. The final program, which included all the major optimizations performed the best, vastly outperforming the original row-block PageRank. The row-block PageRank had an average execution time of 6.69 seconds while the most optimized version had an average execution time of 6.31 seconds. More importantly, the most optimized version had a much slower growth-rate in relation to an increasing number of nodes. This suggests that the optimized version scales much better than the unoptimized version, which is important when considering an algorithm which is designed to operate at massive scales.

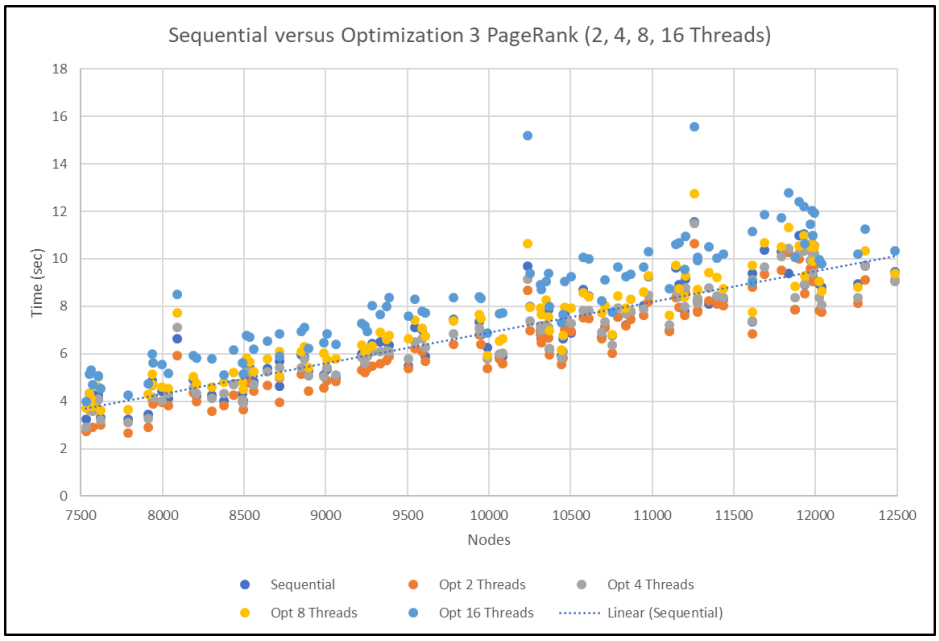


Results

Finally, I compared the results of the best optimized algorithm with the sequential algorithm. Initially, the optimized parallel PageRank outperforms the sequential version. As the number of threads increased, however, the parallel algorithm began to slow down, and underperformed relative to the sequential version. There are several possible explanations for this, but I think the most likely is that the scale of test problems I am operating on is relatively small. The graphs generated have approximately 10,000 nodes and approximately 100,000,000 edges. Given the constraints we have in the Williams CS Department these tests are rather large,[1] but relative to the scale of Web-based applications of the algorithm (e.g. Google ranking all indexed websites on the internet) the testing we could do was limited. The algorithm executed extremely quickly on these relatively large test files, but with fewer storage constraints, I believe the parallel algorithm would scale better. At the current scope, the overhead associated with created threads, dividing work, etc., outweighed the benefits of parallelism. We still observe that the parallel algorithm, for any number of threads, had a slower growth rate than the sequential version as the

[1] Large enough that Mary Bailey emailed me to say I need to remove them ASAP because they are taking up too much storage (~50 GB)

number of nodes increased; this supports my argument that the parallel program scales much better than the sequential algorithm. A graph of these results is included here.



Future Work

The biggest optimization yet to be observed in my work is changing the representation of the graph from an adjacency matrix to an edge-list. This optimization would not be intended to improve performance speeds necessarily, instead it would be used to save memory during execution (which could have an additional benefit, then, of improving speeds). The issue with adjacency matrices is that they use a lot of space, namely n^2 space, which is extremely wasteful for sparse graphs. We don't need to waste memory storing where there *aren't* edges when we're only interested in where there *are* edges. This optimization would have been interesting to explore, but during my work I chose to implement optimizations which focused on execution speeds rather than reducing memory usage.

Under different conditions, more work could also be done to create larger test files, to test the scalability of these various implementations. Unfortunately, I was constrained in my storage capabilities using the Williams College CS Lab computers during this initial work on my implementations.