20sdc2

PageRank Project Proposal

For my final project, I would like to follow the Stanford CS315b assignments 3 and 4 implementation of PageRank and the parallelization of PageRank.

PageRank is the original formula for calculating search results used by Google. The general premise is to devise a website's relative importance based on its connectivity to other websites. This connectivity score is based both on the number of websites which link to a website, as well as the relative importance of *those* websites.

This leads to a necessarily iterative approach, where we must start with the assumption that all websites have the same level of importance, then run through all the connections from site to site calculating new importance levels, then repeating. The levels of importance should converge towards the theoretical "true" level of importance, though we may never actually reach that value.

Each iteration can be parallelized, however, by breaking up how the calculations are done. There are two primary modes described in the Stanford assignment: node and edge parallelization. Each thread can take a specific set of nodes to run calculations on, or each thread can take a set of edges to run calculations on.

I will implement a single-threaded version of this code, a naively node-parallelized version (each thread receives a random set of nodes), an improved node-parallelized version (each thread receives the same number of nodes), a naively edge-parallelized version (each thread receives a random set of edges), and an improved edge-parallelized version (each thread receives the same number of edges).

I will implement a locking schema where each target node's weight needs to be locked before it can be updated. If time permits, I will also create a version where each thread maintains a node-rank delta-set which tracks that thread's changes to each node, which can then be aggregated (reduced) by the main process before the next iteration is called.

Another optimization I will try is how the graph is stored. I will consider an array of edges as well as an adjacency matrix for storing these edges.

The optimizations for arrays versus adjacency matrices and for locks versus reducing local copies will likely be determined during one of the earliest iterations of my code (in the sequential version and the first parallelized version). While this has a small chance of leading to an incorrect solution about which optimization is faster, the optimizations greatly affect the design choice of all versions of the program, and it would be unrealistic to have to code all sets of versions for each iteration of my code given the amount of time we have for this project, and it is likely true that the results of optimization testing will hold relatively true for all cases (so the likelihood of a false signal from an early iteration is low).