Broday Walker

# A Single-Source Shortest Path Algorithm - Dijkstra's Algorithm in Parallel

The Single-Source Shortest Path (SSSP) problem involves analyzing a graph G(V, E) to find the shortest paths from a starting vertex S to all other vertices in the graph. The solution to this problem has many applications in everyday life. For example, an application such as Google Maps makes use of graphs in the form of towns (vertices) and roads (edges) to provide navigation instructions. To supply the customer with the fastest or most efficient route from the user's start to their destination, an efficient algorithm is needed.

There are several SSSP algorithms for computing shortest paths. On an unweighted graph, this can be accomplished with a simple Breadth-First Search (BFS) in O(V + E) time. BFS traverses a graph by discovering and visiting neighboring vertices of the starting vertex and neighbors of neighbors until all vertices have been visited. In other words, BFS discovers neighboring vertices in layers. The connections between layers are effectively equal in weight. In most real-world scenarios, such as the one mentioned above, this is not the case: all paths are not equal and a path through many nodes can be shorter than a path through fewer nodes. For these problems, the graph is composed of vertices with weighted edges, necessitating the use of an algorithm such as Dijkstra's Algorithm.

Dijkstra's Algorithm runs in O((V + E) log V) time and works for both directed and undirected weighted graphs with two key restrictions: all edges must have non-negative weight and the graph must be connected. For the purposes of this project, a graph is considered to be connected if for any two vertices x and y in the graph, there exists a path whose endpoints are x and y. Dijkstra's Algorithm is an example of a greedy algorithm, where the current best choice is made at each step in the algorithm. This is accomplished through the use of a technique called relaxation. For each vertex V in the graph, the neighboring vertices in the adjacency list are analyzed. To relax an edge means to test whether the best known path from the start node S to a vertex U (which is adjacent to V) is to go from S to V, then to take the path from V to U. If this distance is shorter than the currently known best path, the path is updated. This process continues until all vertices have been processed, resulting in the shortest path from S to all other vertices.

In this project, Dijkstra's Algorithm will be analyzed in multiple ways. First, a serial implementation of the algorithm will be benchmarked to test for accuracy as well as to provide a baseline from which to judge the parallel implementations. Second, a parallel implementation of Dijkstra's Algorithm will be implemented in CUDA C++ without the use of shared memory. The third and final iteration of the Dijkstra's Algorithm will be written in CUDA C++ using shared memory. All implementations of Dijkstra's Algorithm will be executed and timed on Midwestern State University's Turing cluster and Texas Advanced Computing Center's (TACC) Maverick2 cluster. Within TACC's Maverick2 cluster, the parallel implementations will be executed and timed using NVIDIA's GTX 1080 Ti as well as the Tesla V100. The results for each implementation will be graphed and discussed.

References

1. https://courses.cs.washington.edu/courses/cse373/01sp/Lect24_2up.pdf
2. https://algs4.cs.princeton.edu/44sp/
3. https://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf
4. http://www.cs.rpi.edu/~goldberg/14-GT/08-block.pdf
5. Halim, S. & Halim F. (2013). *Competitive Programming 3*.