Short Report:

Dijstra’s algorithm and Breadth First Search (BFS) are both SSSP (Single Source Shortest Path) algorithms that solve the problem of finding the shortest path from a starting node to all other nodes. Sort of like google maps and what not, finding the shortest and most efficient path from a starting location to the destination. This report will focus on comparing their theoretical and real world performances.

To understand their performances and why we’re comparing them, we must first know what each algorithm does. Both algorithms have the same general idea, and it’s reflected in how they function. They start from the starting node, and explore the graph, mapping it out node by node, but while doing so, they always go to the node that is closest to the starting node. In more detail, the first couple steps would be to set the distances with a value and add the starting source node to a queue. Afterwards, they’ll perform iterations in which they’d take out the node with the shortest distance from the source node from the queue. This process will repeat until the queue becomes empty.

Once that’s been done, the algorithms would check on all neighboring nodes of the extracted node and note the new distance they’ve achieved. If the new distance is better than the old one, the algorithms would then update the distance of the new node and push it into the queue. The algorithm then performs another set of iterations until the queue becomes empty.

It’s important to note that Dijstra’s algorithm is virtually identical to BFS, the only real difference being that Dijstra’s algorithm is a weighted graph, while BFS is unweighted, resulting in the fact that Dijkstra’s algorithm uses a priority queue data structure in order to keep track of unvisited nodes while BFS uses a regular queue data structure. This begs the question of how each graph would compare to each other with such a difference after they’ve been parallelized.

To parallelize Dijkstra’s algorithm, a cluster must be first made around a source vertex. Each core will handle a subgroup of the vertices in and out of the cluster. When there’s a vertex that’s not in the cluster, vertices in the subgroup that are not in the cluster will have their distance from other vertices not in the cluster calculated. The vertex with the shortest path as the local closest vertex will be the one selected. Afterwards, a parallel prefix is used to grab the closest vertex from all the local closest vertices from each core.

Parallelizing BFS is a similar ordeal. In a similar method to the one applied to parallelize Dijkstra’s algorithm, the parallelized version of the algorithm will process every vertice of a single level at the same time, once that’s been done, the process can repeat with a parallel loop that processes every level 2 vertice at the same time and so forth, thus ironically, the parallelized algorithm will have a sequential code serve as the main loop of the algorithm from level 0.

Onto comparisons, for our Dijkstra’s algorithm, it was found that compared to a sequential version, the startup time for the parallelized algorithm caused it to fall behind quite significantly when inputted with a low amount of data. However, as the amount of data scales up, the gap closes, and on average, with around 500 vertices, the parallelized algorithm begins to outperform it’s sequential counterpart. The exact numbers are as follows,

Data input: 500 vertices

Sequential algorithm: .004034 second(s)

Parallel Dijkstra: .003797 seconds(s)

Data input: 1000 vertices

Sequential algorithm: .023626 second(s)

Parallel Dijkstra: .014205 seconds(s)

From there on out, the parallelized algorithm will begin to outwork the sequential algorithm, and with the ability to create more threads, the parallelized algorithm has the ability to delay it’s starting runtime even further in order to scale up better at the cost of processing power.

BFS runs a similar story.

Data input: 500 vertices

Sequential algorithm: .012037 second(s)

Parallel BFS: .008248 seconds(s)

Data input: 1000 vertices

Sequential algorithm: .021161 second(s)

Parallel BFS: .020124 seconds(s)

To conclude, a sequential implementation has a lower initial runtime due to not having to worry about the overhead that comes with multiple thread creations and what not, but fails to scale with ever increasing data inputs. The parallelized algorithm can theoretically scale up to a certain max depending on the amount of processors and cores available to it, as it can create threads up to the hardware maximum. As a result, a parallelized implementation would be ill suited for small scale tasks in the real world, and would be used to its greatest potential in software and applications such as a global positioning system and what not. A parallelized BFS implementation is much the same way, however, it does compare more favorably against it’s sequential counterpart when it comes to smaller data inputs.