Parallel Implementation of Dijkstra’s SSSP Algorithm by Non-Recursive Reduction

Final Project Writeup

Nolan Orloff

CPSC 5600

Problem Space Exploration

Dijkstra’s Single Source Shortest Path algorithm, often abbreviated to Dijkstra’s SSSP or Dijkstra’s algorithm, is a graph search algorithm to find the shortest paths between a source vertex and all other vertices in a graph. The graph is weighted or unweighted, non-directed and possibly cyclic. Implementation of Dijkstra’s algorithm can give one of two results. The first is a table of destination vertices and their costs where the cost is the sum of weights of edges in the path from the source vertex to the destination vertex. The second is a set of paths that form a shortest-path tree rooted at the source vertex. The shortest-path tree is a spanning tree of the graph with each branch being the shortest path from the root vertex to another vertex. Though producing the shortest-path tree requires extra work on top of the cost table. In most implementations of Dijkstra’s algorithm, it is trivial to implement one result type if the other is already implemented. Dijkstra’s algorithm has many applications. It is most often used for creating optimal paths for laying pipe or electricity lines or for plotting a driving route between two points.

Dijkstra’s algorithm is a notoriously hard problem to parallelize. This is not because there are few possible implementations of parallel Dijkstra’s algorithm, but because there are few implementations that do not add an unreasonable amount of work. Many papers have been published that propose solutions that avoid as much overhead as possible; and I referenced two for my project. The first is *More Parallelism in Dijkstra’s Single-Source Shortest Path Algorithm* by Kainer, Träff published in March 2019. This paper proposes a solution that works in phases. Work within each phase is done in parallel, but phases are executed serially. This solution can run using a small root of *n* phases (*n* being the number of vertices in the input graph) and similar work and latency to Δ-stepping implementations. The main goal of the solution in the paper was to run with work O(*n*); and the proposed solution could not guarantee this. The second source I referenced is a side deck *An Implementation of Parallelizing Dijkstra’s Algorithm* by Zilong Ye for a course for University at Buffalo. The solution presented in the slides is very similar to a standard serial solution for the algorithm. Many threads work in parallel on a shared graph to find the globally-lowest-cost unvisited vertex in the graph at each step. Threads do this by finding their local minimum-cost unvisited vertex and broadcasting it to other threads. This solution has significant parallelization overhead; and work and latency are in the same efficiency class as the serial solution. There is some speedup, but it is quickly dwarfed by parallelization overhead. This is especially true when threads are on different compute nodes.

Problem Statement

For my final project, I implemented a parallel solution to Dijkstra’s SSSP using the non-recursive tally reduction library. One risk that I did not expect is that the programming paradigm enforced by this library would make implementing Dijkstra’s algorithm impossible. The reduce class requires the input dataset to be a list of objects that threads work on in pieces. The input dataset cannot be a single shared object. Both of the referenced solutions used a single graph object that threads share access to; and this structure is an integral part of how the solutions achieve work and latency efficiency classes similar to the serial solution. Overcoming the risk presented by using the tally reduction class in my solution required significant extra work. Because of this, my solution has no target work or latency efficiency classes. Also to limit the complexity of my solution, I limited weights to only positive numbers and assumed that all edges in the input graph were directed. These limitations can be trivially relaxed.

Approach

After reading the reference sources, I wrote a serial implementation of Dijkstra's algorithm as a baseline to compare my parallel implementation against. The serial solution uses a list of frontier edges. Each edge in the frontier list start at a visited node and ends at an unvisited node. The frontier list is sorted so that the first element has the lowest weight. Each iteration follows the following instructions:

- The lowest weight edge is popped off the list and a path that ends in the edge is found.
- The last vertex in the new path is added to the list of visited nodes.
- Add all edges that start at the last vertex to the frontier.
- If the new path is the lowest-cost path to its last vertex, add the new path to the shortest-path tree and delete higher-cost paths with the same last vertex.
- Repeat until the frontier is empty.

One highlight of this implementation is that the input and output data types are the same as in the parallel implementation. The input data type is the custom class edge. This class represents an edge in the graph and has a start vertex, end vertex and weight. The output data type is a list of the custom class path. This class holds a list of edges that represent a valid path in the graph. Using the same data types in the serial and parallel implementations means that their runtimes can be more easily compared because their basic operation is the same. The basic operation in both the serial and parallel implementations is comparing two edges.

The work and latency cost for my serial implementation of Dijkstra's SSSP is below.

- For an input graph with *v* vertices and *p* paths
- The basic operation is a comparison between two edges
- The cost is not dependent on the quality of input
- The main processing loop is run $O(v^2)$ times. Inside the main loop:
    - The frontier is sorted in log(*v*) comparisons
    - The list of paths is sequentially searched twice
    - One row of the graph is sequentially searched
    - This results in O(log(*v*)) + O(2*p*) + O(*v*)
    - Because *v* is always at least double *p*, the entire loop is O(*v*)
- This means that the entire algorithm is $O(v^3)$

This amount of work is terrible for any implementation of Dijkstra's algorithm. In the future I hope the optimize the amount of work to within the expected efficiency class for the problem.

Parallel Implementation

My parallel implementation for Dijkstra's algorithm takes as an input a list of edges that form a graph. Each tally object holds a local graph formed from edges it has seen and a shortest-path tree for its graph. On instantiation, a tally object initializes an empty graph of size *v* x *v* where *v* is the number of vertices. The object also initializes a new list of paths and adds the identity path to the list. The identity path is the path containing one edge from the start node to the end node with weight 0. This path is used to find any other paths in the graph. To accumulate a new edge, the tally object adds the new edge to its local graph and looks for new paths that end at the new edge. To combine with another object, the tally object accumulates all edges in the other object's graph that it has not seen before.

Looking for new paths is the heart of my parallel implementation. It performs work that grows with the number of paths, not the number of edges by assuming that after an edge is added to the graph, new paths must contain that edge and if there is not a path that ends at the new edge, there are no new paths that contain the new edge. These assumptions hold because of the requirements that there is always only one source vertex, edges in this implementation are directed and there is a maximum of one edge between any two vertices. Because these requirements are true, the shortest path from vertex 0 to vertex n+1 must be made from the shortest path from vertex 0 to vertex n and the edge from vertex n to vertex n+1. If there is no path from vertex 0 to vertex n, there cannot be a path from vertex 0 to vertex n+1.

One other major characteristic of my parallel solution is that tally objects can find more than one locally-shortest path to the same vertex. Graphs can have more than one path from the source vertex to a single destination, but only one path is the shortest. The order in which a tally object see new edges cannot be controlled, so the object may find one path to a vertex, then find another shorter path later. When this happens, the original path and all paths containing that path must be updated to use the new path. Because of the logic in the previous paragraph, this updating work can be done in cost that grows with the number of paths, not the number of edges.

The biggest difference between my serial and parallel implementations is that in the parallel implementation, only the final tally object has complete knowledge of the graph. All other tally objects only see a subset of the edges in the graph. Because the serial implementation has complete knowledge from the beginning, it can always pick the lowest cost edge on the frontier to traverse. This means that there is very little backtracking or redoing of work. In the parallel solution, because only one tally object has complete knowledge and that object only has complete knowledge at the last combination step, there is a great deal of backtracking and redoing work. This is where all of the parallelization overhead comes from and is why a fast parallel implementation is hard to develop.

The work cost for my parallel implementation of Dijkstra's SSSP is below.

- For an input graph with *v* vertices and *p* paths
- The basic operation is a comparison between two edges
- The cost is not dependent on the quality of input
- The reduction is structured as a binary tree of interior nodes, so the cost will be equal to log(*v*) * the work of the loop ran by interior nodes.
- The work of one interior node is always the work of updatePaths() * the number of accumulated edges
    - updatePaths() iterates over the paths list a maximum of four times and iterates over edges in all paths a maximum of once
    - This means that the maximum number of basic operations per call to updatePaths() is $O(5P)$
- The maximum number of edges that can be accumulated at once is $\frac{v}{2}$
- The maximum amount of work per interior node is $O(5P * \frac{v}{2})$
- The maximum amount of work for the algorithm is $O(5P\log{(v)})$

The latency cost for my implementation is below.

- The latency of the algorithm is consumed by the latency of the first thread because it stays alive for the entire run of the algorithm.
- The first node is alive for one Schwartz-tight loop and log(P) strides where P is the number of threads used.
- This means the first node is alive for log(P) + 1 timesteps
- Each timestep takes $\frac{v*s}{P}$ sub-steps where:
    - v is the number of vertices in the input dataset
    - s is the current size of the stride
    - P is the number of threads used
- This means that the overall latency cost is $O(\log(P)\frac{vs}{P})$

One interesting note about these costs is that even though the parallel implementation does significantly more work than the serial implementation, the parallel implementation has lower latency and will finish faster.

Future Work

One major improvement for future work on this project is optimizing the serial implementation to provide a better benchmark to compare the parallel implementation to. The parallel implementation can also be improved. One way to reduce work cost in the parallel implementation is to do the reduction in phases. Tally objects in each phase create a graph that represents a contiguous section of the overall graph. This change would improve work by reducing the number of edges to iterate over and by reducing the local maximum path length.