Parallel Shortest Path Problem

Ibnul Jahan (iej) + Ryan Gess (rgess)

Summary:

We will parallelize the sequential shortest path problems involving Dijstra's algorithm, Bellman-Ford's algorithm, and Johnson's algorithm using OpenMP and MPI and compare the performances.

Background:

There are three distinct types of shortest path algorithms that we will be attempting to parallelize and benchmark. We will use them to solve two different graph theory problems related to determining lengths of shortest paths, but they have varying amounts of computations and information in their outputs. In order, they will be Dijikstra's, Bellman-Ford's, and Johnson's, the latter of which will build on the earlier two. The two shortest path problems we are solving are as follows:

The Single Source Shortest Path problem (SSSP) - This problem attempts to find the path of shortest weight between one specified node (typically denoted s) and every other node in a graph. For notation's sake, the number of edges in a graph is typically denoted as m and the number of vertices in a graph is denoted as n. We will solve this problem using two different algorithms: Dijikstra's and Bellman-Ford's.

- Dijikstra's algorithm - This algorithm works under the assumption that the supplied graph does has only positive edge weights. It has a runtime complexity of O(mlogn) in typical implementations, and we will be looking to improve upon this through the use of parallelization.
    - Dijikstra's is a very commonly implemented algorithm, but suffers from a poor performance as the size of our graphs increase, and for this reason can benefit from parallelization. The algorithm is structured to be inherently sequential, but by unrolling the series of our initial operations we can create separate tasks to do computations with.
- Bellman-Ford's algorithm - This algorithm is able to handle graphs that have negative edge weights, under the condition that there are no negative edge cycles. It has a runtime complexity of O(mn), and we will be looking to improve upon and benchmark this as well.

    - Bellman-Ford's works by relaxing edges simultaneously, creating a spread that gets farther from the source as iterations increase. This is an optimal place for parallelization can occur, as we will have tasks increase rapidly in most graphs, and more and more edges and vertices need to be expanded at once.

The All Pairs Shortest Path problem (APSP) - This problem expands upon the SSSP, and attempts to find the path of shortest weight between every single pair of vertices. Naively, this can be done by running an SSSP solver, like Bellman-Ford's, on every single vertex. There are other algorithms, in this case Johnson's, that work to greatly reduce the time complexity and repetitiveness of this naive approach.

- Johnson's Algorithm - This algorithm is able to handle graphs of negative edge weights by making a call to Bellman-Ford's, and then making subsequent calls to Dijikstra's after altering the graph. This makes this algorithm directly built on top of the prior two, and gives it a complex runtime of O(nmlogn).
    - This will be a great algorithm to parallelize, as we will be able to benefit from the parallelization of the previous algorithms mentioned. Different implementations of each can both be attempted, leading to a great place to showcase the benefits of different parallelization schemes.

The Challenge:

This problem is challenging because in order to determine the minimum cost in a graph, a large number of vertices need to be visited, growing as the number of edges and vertices increases. Each algorithm solves the problem differently so each will provide a unique challenge in parallelization. In general, however, we can expect a very large amount of communication depending on how our vertices are spread apart, as the computations are relatively simple in comparison.

Dijikstra's

- There is a lot of sequential work built into the algorithm, so finding the optimal areas to parallelize will be difficult. It works by having a collective priority queue to determine which edge to interpret, and parallelizing around this collection of information will be difficult.
- There will be a large amount of global data in this problem, as regardless of our task separation every task will need to be aware of currently visited nodes and this global priority queue.

- Because of the inherent sequentiality of the algorithm, task creation will be a difficult problem to overcome. Finding set pieces of work for different threads to do will be hard, as will scheduling them to execute.

Bellman-Ford's

- This algorithm is reliant on relaxing the amount of the graph we are computing over through our iterations. Because of this, the number of tasks grows rapidly as we continue our iterations, and scheduling these different tasks will prove to be complex.
- In an MPI implementation, there will be a significant amount of communication increase between iterations. This will lead to a significant change in the overhead, leading to a potential for poor performance scaling.

Johnson's

- This algorithm is a combination of Bellman-Ford and Dijikstra's algorithms so in parallelizing both Bellman-Ford and Dijikstra's it will be interesting to see what our bottle-neck is when they are combined and what extra complications may be faced.
- There is clearer work separation for this algorithm, but the tasks are very large (possible whole runs of Dijikstra's), so speedup will likely be bottlenecked by this.

One collective challenge for parallelizing graph problems in general is the separation of vertices into different groups. If a single processor is only responsible for a subset of vertices, then a significant amount of communication will need to occur, particularly in an MPI implementation.

In addition to exploring the strengths and weaknesses of each algorithm, we also hope to see how much we can speed up each algorithm from its sequential version. This will culminate well in our implementation of Johnson's where we can mix and match approaches to try to get our maximal speedup.

Resources:

We plan to use the GHC cluster to run our code and perform our speedup analysis. We will code our solution from scratch using C/C++, OpenMP, and MPI. We will reference pseudo code of the algorithms online (Wikipedia) and we are somewhat familiar with these algorithms from courses we have taken already.

Goals and Deliverables:

Our goals are to implement the three algorithms in both OpenMP and MPI. We will use these three implementations to compare performance in speedup and overall runtimes as we vary both problem size and the number of resources available to us.

We plan to achieve a significant speedup compared to the sequential versions that we initially implement. For our implementation of Dijikstra's, we are hoping and expected to achieve a modest speedup of 4x, as there is still a significant amount of sequential work we will have to overcome. For Bellman-Ford, we are hoping and expecting to achieve a speedup of 8x, as we will be able to more clearly create tasks and will work to dynamically schedule the increasing loads of work.

A large portion of this project will be about analyzing our performance analysis in these different platforms as we vary the number of resources we use. We plan to observe which algorithms scaled the best as input size and thread count increases, and see which implementation of the algorithms will scale better as well. We will look to see if the overhead of communication in MPI will be a big barrier as well, because we expect difficulty in large amounts of communication in graphs with large vertices.

At the poster session, we expect to largely be presenting graphs of performance analysis and comparing how the different setups performed against one another. It would be possible for us to do a demo on a live graph, but it would likely lead to an uneventful pause as we wait for the programs to run. The poster session would be better served by us spending time on our analysis and scaling.

If we are able to make significant progress early on towards these initial goals, we also are considering attempting to port some (or all) of these algorithms on CUDA, and seeing the speedup we are able to achieve on that platform as well. The first one we would approach would be Bellman-Fords, and if we are only able to achieve one algorithm to do in CUDA, we could also benchmark test our Johnson's algorithm with the new CUDA version we have created.

Platform Choice:

We expect to build our project using C++ or C and OpenMP and MPI on GHC and later the Latedays clusters. We believe that these systems are suitable for our workload because there is a decent amount of independent work and eventually the Latedays clusters will allow us to test on many more nodes.

Schedule:

April 12	- Implement Sequential version of each algorithm (Dijstra's, Bellman-Ford's, and Johnson's) and benchmark - Set up git repository - Research and make a plan to parallelize each algorithm
April 19	- Parallelize each sequential algorithm using OpenMP and benchmark
April 26 (Project Checkpoint)	- Parallelize each sequential algorithm using MPI and benchmark - Write project checkpoint report
May 3	- Implement in CUDA or continue optimizing OpenMP or MPI code
May 10 (Report Due)	- Write report - Create presentation