Parallelizing the Single-Source Shortest Path Problem using Delta-Stepping

Daniel Li & Enrico Green

Website: https://github.com/EnricoGreenStudent/15618-Project/

Summary

We plan on parallelizing the single-source shortest path (SSSP) problem using the delta-stepping algorithm. We will implement several classical sequential SSSP algorithms, as well as parallel versions of the delta-stepping algorithm using both OpenMP and CUDA. We will then compare our various implementations in order to analyze how graph algorithms are parallelized across CPUs and GPUs.

Background

Single-source shortest path (SSSP) is a graph theory problem, which consists of finding the shortest path to each vertex in a graph from a given starting vertex. A solution to the SSSP produces a spanning tree rooted at the starting vertex, such that the distances to each vertex is minimal. Various classical algorithms exist to solve this problem in the sequential case, the most well-known of which include Dijkstra's algorithm for non-negative weighted graphs and Bellman-Ford for the general case.

Shortest path algorithms see application in calculating minimum distance between points on a road network (which can be made faster using heuristics), or routing through a network with a focus on minimum latency.

The Challenge

As mentioned in the Background section, the "traditional" sequential algorithms for solving SSSP are Dijkstra's Algorithm and the Bellman-Ford algorithm. Dijkstra's algorithm is a label-setting algorithm in which correctness depends heavily on the vertices being processed in the right order. Hence, it is inherently sequential, as processing multiple vertices at the same time may result in an incorrect output. Bellman-Ford, on the other hand, is a label-correcting algorithm which can be easily parallelized. However, this parallelism comes at the cost of a large amount of redundant work. In order to admit parallelism without sacrificing a large amount of work efficiency, the Delta-stepping algorithm uses a parameter (delta) in order to determine which vertices should be processed next. One of the main challenges in implementing delta-stepping will be figuring out how to choose a "good" delta which admits a good balance between parallelism and work efficiency.

Another challenge will be minimizing data movement and synchronization. Graph algorithms are somewhat notorious for having low arithmetic intensity. We suspect this will be especially noticeable in our CUDA implementation, as communication to/from the GPU is bound by the bandwidth of the PCIe connection to the GPU. Minimizing unnecessary communication and synchronization will be key in maximizing the speedup offered by parallelization.

Resources

For this project, we plan on using the GHC machines for both development and benchmarking. We don't expect to use any starter code, though we may source some graphs for testing and benchmarking from elsewhere. Some papers and other resources we may use for reference are listed below:

The delta-stepping algorithm is described in the 1998 paper by Meyer and Sanders

- https://www.sciencedirect.com/science/article/pii/S0196677403000762

An MPI implementation is found in the Boost Graph Library

- https://www.boost.org/doc/libs/1_46_1/boost/graph/distributed/delta_stepping_shortest_paths.hpp

Optimizations to the delta-stepping algorithms are detailed in some other papers

- https://dl.acm.org/doi/10.1145/3368826.3377909
- https://dl.acm.org/doi/10.1145/3409964.3461782

Example of delta-stepping algorithm steps over time

- https://cs.iupui.edu/~fgsong/LearnHPC/sssp/deltaStep.html

Goals and Deliverables

Plan to achieve:

- Implement Delta-Stepping Algorithm using both OpenMP and CUDA, with significant speedups over sequential versions. Note that due to the relatively low arithmetic intensity of graph algorithms, a linear speedup is not realistic.
- Analyze how different values of delta affect the performance of delta-stepping
- Analyze performance of implementations and the speedup achieved
- Use the above analysis to compare how graph algorithms are parallelized differently on CPUs and GPUs

Hope to achieve

- Compare performance of implementations with Boost algorithm or other existing implementations of parallel SSSP
- OR attempt to utilize graph-centric DSL to simplify code and further raise performance

Analysis/Deliverables

- Deliver working implementation of delta-stepping on test input graphs
- Learn how properties of graph influence speedup, see speedup in example graphs
- Non-interactive demo of delta-stepping (Hope to achieve)

Platform Choice

We plan on using the GHC machines for development and benchmarking. Because we gained some familiarity working with C++ in previous assignments, we plan on using C++ for our implementations. We feel that the OpenMP and CUDA programming models are a good choice for this project, as they will allow us to contrast how parallelizing graph algorithms differs on CPUs and GPUs.

Schedule

Week (Dates)	To-Do
Week 1: Tr 11/16 - We 11/22	Create Testing & Benchmarking tools Implement Sequential Dijkstra's and Bellman-Ford algorithms
Week 2 (Thanksgiving Break): Tr 11/23 - Su 11/26	Implement Sequential Delta-Stepping Algorithm
Week 3: Mo 11/27 - Su 12/03	Implement OpenMP Delta-Stepping Algorithm Write Milestone Report (Due 12/03)
Week 4: Mo 12/04 - Su 12/10	Implement CUDA Delta-Stepping Algorithm
Week 5 (Finals Week): Mo 12/11 - Tr 12/14	Look into tuning delta parameter (if time available) Prepare Final Project Report (Due 12/14) Prepare for Poster Session (12/15)