Parallel shortest path

Mina Goranovic, Philippe Guyard, Mark Daychman

3rd of June 2023

1 Introduction

SSSP is a class of algorithms that aims to solve the single source shortest path problem. Precisely, given a weighted graph $G = (V, E)$ and a source node $s \in V$, we want to find the shortest distances from this source node s to every other node $v \in V$. Algorithms for pathfinding have broad applications, from GPS routing to AI game navigation. Common algorithms include Dijkstra's and A* search, which determine distances very efficiently, but are inherently sequential, limiting their potential on parallel computing architectures.

In this project, we implemented the Delta-stepping algorithm, which is a variant of Dijkstra's that is more suitable for parallel processing. Despite being more complex, delta-stepping demonstrates significant speedup on parallel systems without compromising on the accuracy of the shortest path. Our implementation of the algorithm is heavily based on the following paper [1].

The project structure is outlined below with more in-depth explanation of each step in the following sections.

2 Delta-stepping algorithm

SSSP algorithms very often use the following structures:

1. Nodes are divided into 'visited' and 'unknown'. The source node s is always visited
2. For all 'visited' nodes, the shortest distance from s has been computed. For 'unknown' nodes, we keep a 'tentative' distance *tent*, which may or may not be the shortest one (defaults to ∞)
3. While there are 'unknown' nodes, we keep them in some sort of queue, and then handle elements in the queue in some pre-specified order, improving their tentative distance every time.

For example, a common implementation of Dijkstra uses a priority queue, sorting unknown nodes by their tentative distance.

The Delta-Stepping algorithm follows a similar paradigm. It accepts a hyper-parameter δ, and uses an array of 'buckets' as a queue. Each bucket is a set of nodes, and the bucket at position i is defiend as follows:

$$B[i] = \text{All nodes such that } i\delta \leq tent(v) < (i+1)\delta.$$

When going over our queue, it handles the nodes bucket by bucket, performing an operation similar to Dijkstra's algorithm for every node in the bucket. From here, a natural sub-division of the edges E of our graph arises. Indeed, we define two classes of edges:

1. 'light' edges have a weights strictly less than δ.
2. 'heavy' edges have a weight at least δ.

We need to treat 'light' edges separately, as an outgoing 'light' edge from a node in bucket i might end up in a tentative distance that still belongs to bucket i. This also allows us to parallelize handling light edges, which gives Delta-stepping an edge over Dijkstra on highly-parallel machines.

[1] "Δ-stepping: a parallelizable shortest path algorithm", Journal of Algorithms 2003

3 Implementation

3.1 Code outline

All our code for this project is contained in this Github repository. To compile and run the project you need to have CMake installed. To run the project, use the command.

```
cmake .; make .; ./main
```

The project is divided as follows:

1. **benchmarker.hpp:** This is a small custom tool we wrote to measure the performances of small snippets of code and display the summary at the end. We used it extensively to find bottlenecks in our implementation.
2. **buckets.hpp:** We abstracted away the implementation of a 'bucket list' interface from the Delta-Stepping solver. This was especially useful in the early stages of the project, when comparing sequential Delta-Stepping to Dijkstra. In short, using the SimpleBucketList class in the sequential solve yields similar results to the $O(N^2 + M)$ implementation of Dijstra. On the other hand, the PrioritizedBucketList class behaves similarly to a priority queue, so using it in the sequential solve yields similar results to the priority queue implementation of Dijkstra with $O((N + M)logN)$ complexity.
3. **dijkstra.hpp:** A classic Dijistra sequential implementation to test our algorithm against.
4. **generators.hpp:** A tool we made to generate arbitrary graphs of different kind.
5. **graph.hpp:** All the auxiellary classes that are used in the delta-stepping algorithm are defined here.
6. **main.cpp:** Testing showcase with extensive logging from the benchmarker. This file also checks our implementations for correctness.
7. **solvers.hpp** The main DeltaSteppingSolver. The different methods from it will be described below. This file also contains a DeltaSteppingSolverSorted, which will be explained in 3.2

Our DeltaSteppingSolver has many methods, following the different algorithms described in the original delta-stepping paper:

1. We first implemented `solve` method, which is a sequential implementation of the delta-stepping algorithm.
2. After successfully implementing the sequential algorithm, we replaced the most costly step of the algorithm - `find_requests` - by its parallel version - `find_requests_parallel`, following the parallelization described in section 4 of the paper. Note that the paper also mentions a possible parallelization of the `relax_request` method, which we did not implement as we observed that the total execution time of `relax_requests` was much smaller than that of `find_requests`, so we concentrated on optimizing that instead. The 'improved' version of the solving algorithm that makes use of `find_requests_parallel` is called `solve_parallel_simple`.
3. Later, we also implemented `solve_parallel_omp`. This is the same as `solve_parallel_simple`, but it uses the famous OMP library to parallelize computatiosn. This was useful to verify that we did not make implementation mistakes in our scheduling or in general with using std::thread in C++.
4. After this first simple parallelization was done, we implemented the algorithms described in later sections of the paper, which made use of a pre-computation of 'shortcuts'. Unfortunately, we could not make this implementation work faster than `solve_parallel_simple`. Nevertheless, you can find the code in `solve_shortcuts_simple`, `solve_shortcuts_parallel` and `solve_shortcuts_omp`.

The appendix of the paper also contains a algorithm to generate an optimal bucket size. We have also implemented this method under the name `find_delta`, but are currently experiencing some technical problems that we hope will soon be resolved.

3.2 Implementation idea: semisorting graph edges

We noticed that parallel versions of delta-stepping, especially the one with shortcuts, traversed outgoing edges from every node many times, filtering for heavy and light edges every time. We then thought of semi-sorting the edges of G into two edge lists:

1. A light list, which would contain only light outgoing edges for every vertex

2. A heavy list, which would contain only heavy outgoing edges for every vertex

We then make use of this knowledge in DeltaSteppingSolverSorted. While the semi-sorting itself was slow for our graph sizes, it is easy to see that assymptocically its cost should become negligeable compared to overall dijkstra cost (because it can be done in $O(|E|)$, and easily parallelized, as evidenced by our implementation in graph.hpp, `semisort_delta_parallel`). When making use of semi-sorting knowledge, `DeltaSteppingSolverSorted::find_requests_parallel` was about 2 times faster than `DeltaSteppingSolver::find_requests_parallel`. We did not have time to test that on shortcuts, but we also expect a massive improvement there.

4 Testing and Results

To test the performance of our algorithms, we would measure their execution times against the sequential counterpart, as well as baseline Dijkstra algorithm. We would also vary the types of graphs we test our algorithm on. Here are the types of graphs we used:

1. **Random uniform graph:** Each pair of vertices in the graph has a fixed probability to have an edge between them.
2. **Random uniform connected graph:** We achieved this by checking if the randomly generated graph is already connected. If not, we add random edges until it is.
3. **Random sparse graph:** It takes four arguments, N, p, p_sparse and max_degree. The maximum degree of p_sparse*100% of the nodes is restricted to a certain value (max_degree).
4. **Random dense graph:** It takes four arguments, N, p, p_dense and min_degree. The minimum degree of p_sparse*100% of the nodes is set to a certain value (min_degree).
5. **City graphs:** It takes six arguments, N, p, p_sparse, p_dense, max_degree and min_degree. We create a city with p_sparse*100% of industrial (dense) areas and p_dense*100% of commercial (dense) nodes. The residential area is generated like in the random graph generator.

The results for each algorithm are summarised below.

Type of Graph	10k nodes	20k nodes	30k nodes
Random Graph			
Dijkstra Algorithm	409	1464	4113
Sequential Delta-Step Algorithm (`solve`)	424	2800	29000
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	196	718	1847
Sequential with shortcuts	531	2053	7429
Parallel with shortcuts	429	1752	8006
Random Connected Graph			
Dijkstra Algorithm	400	1579	5903
Sequential Delta-Step Algorithm (`solve`)	569	1377	6268
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	371	698	2409
Sequential with shortcuts	635	2161	7181
Parallel with shortcuts	599	1988	7091
Sparse Connected Graph			
Dijkstra Algorithm	262	1044	3279
Sequential Delta-Step Algorithm (`solve`)	302	2414	15000 -
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	160	561	1634
Sequential with shortcuts	422	1833	5095
Parallel with shortcuts	333	1487	4399
Dense Connected Graph			
Dijkstra Algorithm	398	1785	4355
Sequential Delta-Step Algorithm (`solve`)	321	1476	10000
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	146	690	1795
Sequential with shortcuts	461	3449	5787
Parallel with shortcuts	419	2150	5558
Random City Graph			
Dijkstra Algorithm	505	1945	12412
Sequential Delta-Step Algorithm (`solve`)	544	21000	83000

Type of Graph	10k nodes	20k nodes	30k nodes
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	310	1129	6000
Sequential with shortcuts	730	3209	44000
Parallel with shortcuts	616	2934	16000

Table 1: Average time (in ms) each algorithm took to resolve each graph type of each size for **16 threads**

Type of Graph	10k nodes	20k nodes	30k nodes
Random Graph			
Dijkstra Algorithm	436	1788	3888
Sequential Delta-Step Algorithm (`solve`)	522	22000	15000
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	216	1665	1982
Sequential with shortcuts	706	2909	5690
Parallel with shortcuts	549	2352	5235
Random Connected Graph			
Dijkstra Algorithm	416	1791	3540
Sequential Delta-Step Algorithm (`solve`)	1713	11000	166000
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	209	856	108000
Sequential with shortcuts	627	2479	13000
Parallel with shortcuts	591	2287	6183
Sparse Connected Graph			
Dijkstra Algorithm	282	1132	2631
Sequential Delta-Step Algorithm (`solve`)	1549	11000	53000
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	167	670	1371
Sequential with shortcuts	472	1953	4445
Parallel with shortcuts	373	1645	4206
Dense Connected Graph			
Dijkstra Algorithm	437	2261	8302
Sequential Delta-Step Algorithm (`solve`)	1369	32000	32000
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	192	1099	2841
Sequential with shortcuts	604	4663	10000
Parallel with shortcuts	462	3114	7527
Random City Graph			
Dijkstra Algorithm	567	3153	7945
Sequential Delta-Step Algorithm (`solve`)	613	14000	177000
Parallel Delta-Step Algorithm (`solve_parallel_simple`)	288	1205	2137
Sequential with shortcuts	752	4264	6878
Parallel with shortcuts	673	2858	6636

Table 2: Average time (in ms) each algorithm took to resolve each graph type of each size for **8 threads**

Note that the parameters used for the tests are following:

- p = 0.2
- p_sparse = 0.03 (and for random_city 0.1)
- p_dense = 0.01 (and for random_city 0.2)
- max_degree = 0.3N
- min_degree = 0.8N

All the tests were run on MacBook Air chip M1, with 8-core CPU.

5 Conclusion

Based on the tests in the tables above, we remarked:

- For all graph types and sizes, the **Parallel Delta-Step Algorithm** (solve_parallel_simple) consistently performs faster compared to other algorithms from the table.
- The **Dijkstra Algorithm** generally performs better than the Sequential Delta-Step Algorithm (solve) and Sequential with shortcuts, but it is not as fast as the Parallel Delta-Step Algorithm or Parallel with shortcuts. Its performance also varies depending on the graph's density.
- The **Sequential Delta-Step Algorithm** (solve) exhibits a significant increase in computation time as the graph size increases. This is most noticeable for the Random Connected Graph and Random City Graph types.
- The **Sequential with shortcuts and Parallel with shortcuts** algorithms generally perform better than the Dijkstra Algorithm and Sequential Delta-Step Algorithm (solve), but they are not as fast as the Parallel Delta-Step Algorithm (solve_parallel_simple). Their performance appears to be fairly stable across different graph densities, but they still struggle with increasing graph size, especially in the Random Connected Graph type.

As mentioned before, we also have implementations that make use of the OpenMP library for both the simple parallelization and shortcuts. In both cases, these OMP solves were added to check that we implemented scheduling correctly (using the omp library as benchmark for parallelized computing). As expected, OMP solves were faster than our own implementations (probably due to superior scheduling and making use of all available threads), but not much faster (usually a 10-20% speed increase).

The obtained results confirmed our hypothesis that a parallelizable approach would offer substantial performance advantages over traditional, sequential algorithms such as Dijkstra's, especially when dealing with large graphs. The Parallel Delta-Step Algorithm consistently outperformed other methods across various graph types and sizes, affirming the potential of parallel processing in pathfinding tasks. Although the use of shortcuts provided some performance benefits, they did not surpass the efficiency achieved by parallel processing. When transitioning from 8 to 16 cores, we observed that the performance generally improves for most algorithms with parallel algorithms benefiting more than sequential ones. The results affirm the value of the Delta-stepping algorithm for solving SSSP problems, especially due to its parallelizable potential.