Parallel Single Source Shortest Paths

Vincent Kang, David Zeng

July 10, 2018

1 Summary

We implemented the delta-stepping algorithm using OpenMP which computes single source shortest paths.

2 Background

2.1 Algorithm

Delta-stepping is similar to Dijkstra's but allows for parallel computation through the use of buckets instead of a priority queue. Like Dijkstra's algorithm, the algorithm uses tentative distances from the source to all the vertices, initially set to ∞ except for the source which is set to 0. In Dijkstra's the vertex, v_{min} with the smallest tentative distance in the priority queue is relaxed. Relaxing a node means that the the tentative distances of all neighbor nodes, w, are updated to $\min\{\text{tent}(v_{min}) + \text{dist}(v_{min}, w), \text{tent}(w)\}$.

The delta-stepping algorithm uses a looser ordering, replacing the priority queue with an array of buckets and a parameter Δ. Bucket i will contain all vertices v such that v is queued and $i \cdot \Delta \leq \text{tent(v)} < (i+1)\Delta$. Instead of relaxing a single vertex, the delta-stepping algorithm relaxes all the vertices in the first non empty bucket. The goal of this relaxation step is to completely process all vertices in the bucket. However, since relaxing all neighbors of vertices in the bucket might cause new vertices to be added to the same bucket, the algorithm first split vertices into light edges and heavy edges. Light edges are edges with weight at most Δ. Heavy edges are edges with weight greater than Δ. The algorithm repeatedly relaxes light edges until the bucket empties,

keeping track of all vertices processed. The algorithm then relaxes heavy edges, which are large enough to guarantee that any relaxed vertices can only go into a higher bucket. This completes the processing of a bucket. The algorithm continues this process until there are no longer non empty buckets. When this happens, the tentative distance array contains true distances from the source. Although not part of the algorithm, the actual shortest path can be easily recovered from the distance array, just as in Dijkstra's. The pseudocode from the delta-stepping algorithm paper is below in figure 1. [1]

```
foreach v ∈ V do                                   -- Initialize node data structures
    heavy(v) := {(v,w) ∈ E : c(v,w) > Δ}           -- Find heavy edges
    light (v) := {(v,w) ∈ E : c(v,w) ≤ Δ}          -- Find light edges
    tent  (v) := ∞                                 -- Unreached
relax(s, 0); i := 0                                -- Source node at distance 0
while ¬isEmpty(B) do                               -- Some queued nodes left
    S := ∅                                         -- No nodes deleted for this bucket yet
    while B[i] ≠ ∅ do                              -- New phase
        Req := {(w, tent(v) + c(v,w)) : v ∈ B[i] ∧ (v,w) ∈ light(v)}
        S := S ∪ B[i]; B[i] := ∅                   -- Remember deleted nodes
        foreach (v,x) ∈ Req do relax(v, x)         -- This may reinsert nodes
    od
    Req := {(w, tent(v) + c(v,w)) : v ∈ S  ∧ (v,w) ∈ heavy(v)}
    foreach (v,x) ∈ Req do relax(v, x)             -- Relax previously deferred edges
    i := i + 1                                     -- Next bucket

Procedure relax(v, x)                              -- Shorter path to v?
    if x < tent(v) then                            -- Yes: decrease-key respectively insert
        B[⌊tent(v)/Δ⌋] := B[⌊tent(v)/Δ⌋] \ {v}      -- Remove if present
        B[⌊x /Δ⌋] := B[⌊x /Δ⌋] ∪ {v}               -- Insert into new bucket
        tent(v) := x
```

Figure 1: Pseudocode for Delta-stepping algorithm [1]

2.2 Key Components of Algorithm

There are several key algorithm components that need to be implemented.

The first component is the interface for finding heavy and light edges of each vertex.

The second key component has to do how relaxation works. A central pattern of Delta-stepping is that the algorithm will first gather all light-edge or heavy-edge neighbors of a set of vertices. Then, it will relax all of those neighbors. Because this occurs on two separate steps, some data structure needs to be used to store those neighbors while they are being collected. Specifically, the neighbor set (**Req** in the psuedocode), represents a set of neighbors visited along light-edges or heavy-edges. The deleted vertex set (**S** in the psuedocode) represents a set of vertices that were

in the bucket and then removed after being processed.

The final components are the the tentative distance array and the bucket storage.

2.3 Work Load and Dependencies

We break the work load of delta-stepping into five main parts: precomputing heavy and light edges (1), gathering light edge neighbors (2), relaxing light edge neighbors (3), gathering heavy edge neighbors (4), relaxing heavy edge neighbors (5). Step (1) is done at the beginning. Then, for each bucket, we need to do steps (2) and (3) repeatedly until the bucket empties, and then we perform steps (4) and (5). All five components can be parallelized. For parts (2) and (4) we can parallelize by mapping blocks of vertices in a bucket/set to threads. We can parallelize parts (3) and (5) by mapping vertices to be relaxed to threads. However, the parallelization of (3) and (5) is somewhat tricky due to contention for the same data structures.

2.4 Modifications

We make several modifications to the original delta-stepping for better performance in practice. These modifications do not impact correctness of the algorithm.

For buckets, we do not remove vertices from a bucket. Instead, we leave it there and just do a check when reading from the bucket that determines whether it should have been removed.

For the neighbor set and the deleted vertex set, we allow for vertices to be added to the set multiple times.

3 Approach

3.1 Key Data Structures

Graph The graph was represented with the compressed sparse row format that was used in assignment 3 and 4. In addition, we stored the in edges and weights for each of the edges. We chose to use this format because we only needed to read from the graph and its minimal use of space.

Neighbor Set To process neighbors of all vertices in a bucket and generate the neighbor set in parallel, we wanted a synchronization free data structure for each thread to write to. We initialized a per-thread array representing the neighbor set for each thread. The choice of an array over a set representing data structure was that in practice, the benefit of only storing each vertex once was far outweighed by additional overhead. In a later section, we discuss one attempt at an efficient set-like data structure that did not yield performance benefits.

Deleted Vertex Set The deleted vertex set comprised a small enough portion of the runtime that there was little benefit to adding parallelism for generating the set. Thus, we used a single global array that kept track all deleted vertices. Our choice of using an array over a more complicated data structure was for similar reasons as the neighbor set.

Buckets We implement Buckets using a fixed size list of reusable arrays. A key insight is that if the maximum edge weight in the graph is M, then only M/Δ distinct buckets are ever in use at a time. Thus, we can reuse buckets by simply inserting vertices into bucket $i \bmod (M/\Delta)$. We initially implemented in bucket with a self-resizing unbounded array. This yielded good performance in the single threaded setting, but created synchronization issues in the multi threaded setting. Since we wanted to avoid the use of locks, we stuck to fixed sized arrays, which unfortunately, greatly increased the memory requirement. In addition, our implementation depended on the fact that M/Δ would not be too large. Since the only operations on buckets are inserts, iterations, and clears, we exposed a stack-like interface to our algorithm. Finally, because our buckets do not support removals, whenever we iterate through a bucket, we check the tentative distance array to determine whether it should still be in the bucket, possible ignoring that vertex.

For the multi-threaded case, we kept our implementation lock-free by using atomic operations. Specifically, on each insert, we needed to increment the number of items and save the new vertex at the un-incremented index. This fit the pattern of one of OpenMP's pragma atomic capture so we used that. Since even in the parallel code, we did not always need atomic inserts, we gave the option for both atomic and non-atomic inserts.

Tentative Distance Array We implemented the tentative distance array as a simple array. For our parallel code, we made it an array of atomic integers, taking advantage of the atomic library

in C++.

In the multithreaded case, we needed to access to an atomic minimum to update the tentative distances. We implemented this using compare_exchange_weak (roughly compare and swap) from the C++ atomic library, using the standard pattern of using a while-loop that continues until the compare and swap succeeds or the minimum has changed.

3.2 Important Optimizations

Selective Parallelism Since the size of the buckets being operated on changes during the runtime of the algorithm, we discovered it was important to only use parallelism when the problem size justified it. Specifically, when relaxing heavy-edge neighbors of the deleted vertex set in parallel, we only used parallelism when when there were at least 100 vertices. When there were less than 100 vertices, performance improved due to savings on thread overhead and atomic operations.

3.3 Optimization that did not work

Synchronization Free Data Structures In an attempt to remove the fine grained synchronization on inserting into buckets for heavy edges, we had each thread become responsible for inserting into certain buckets. To implement this, when relaxing edges, each thread would have separate storage for each thread placing relaxed vertices into a thread if the thread was responsible for the bucket the vertex is going to be placed in. Then each thread inserts the vertices into buckets its responsible for by going across all the threads' vertices to be inserted for the current thread. This change ended up not giving a speedup. The reason for this lack of speedup was because of workload imbalance. After measuring the number of vertices a thread was responsible for inserting, we found that there would be often one or two threads with 2-3x more vertices than other threads. For example, on one iteration of running on the wiki graph we have thread 12 and 11 responsible for inserting 340,289 and 343,774 vertices respectively while the other 14 threads averaged inserting 172,363 vertices. However, workload balancing creates a lot of extra work because it would first require calculating the number of vertices placed into each of the buckets then splitting them evenly among threads. In addition, depending on the number of buckets, there might not be enough granularity to properly balance workload. Therefore, we decided to stick with the finer

granularity synchronization on the bucket inserts.

Set-like Data Structures As mentioned in an earlier sections, certain elements of the algorithm, such as the deleted vertex set, and the neighbor set are represented as sets in the original algorithm, but implemented as simple arrays. The main downside of this was that vertices may have been added to a set multiple times. One way to solve the problem would be the augment the vertex array with a vertex bitvector, that determine set membership. Thus, we can check the bitvector before adding a vertex to the array. Despite the relatively low overhead of this approach, it did not provide any speedup. After some analysis, we found that for the real world graphs we were testing on, vertices being added to a set multiple times was not a frequent enough occurrence to be concerning.

3.4 Use of existing code

We used and modified the graph structure and preprocessing code from assignment 3 & 4 to represent and process graphs from inputs.

4 Results

4.1 Measurement units

We measured performance with wall clock time in milliseconds and derived speedup from the measured times.

4.2 Benchmark Graph Data

We tested our algorithm with multiple graphs, both real-world network datasets and generated graphs. For the real-world graphs we used a Wikipedia network of top categories [2] and a Gowalla social network graph [3]. Specifically, the Wikipedia network is a webgraph of hyperlinks between top Wikipedia pages, and there are 1,791,489 vertices and 28,511,807 edges in the graph. The Gowalla graph is a friendship graph from the Gowalla social networking website, and it contains 196,591 vertices and 950,327 edges.

For our generated graphs there is a random structure graph and a ring structure graph. The random structure graph contains 200,000 nodes and each of the nodes have 20 neighbors sampled from a uniform distribution. We omit self edges so there is also a possibility of having only 19 neighbors. The random graph we benchmarked on has 200,000 vertices and 3,999,976 edges. The ring structure graph contains 10,000,000 vertices and 110,000,000 edges. It was generated by having each vertex have edges to the 10 closest vertices by id (mod 10,000,000) and a random neighbor.

Since each of the graphs did not have weights, we generated weights for each of the graphs. In order to test how the algorithm performed on different weight distributions, the weight generation either followed uniform(1,1000) distribution or a geometric distribution with p = $\frac{1}{10}$.

4.3 Performance Measurements

We ran our performance tests on a 16-core Amazon EC2 instance (m5.4xlarge). The processor used is a Intel(R) Xeon(R) Platinum 8175M CPU. For the performance data, uniform and geometric refer to the distribution of the edge weights. We compare our parallel implementation with the sequential version of delta-stepping with $\Delta = 1$ because this sequential version outperformed our implementation of Dijkstra's algorithm. All of the times are measured in milliseconds. For the parallel benchmarks we ran the algorithm with $\Delta = 2$. We ran the algorithm once on each graph to compute the shortest paths from a single vertex.

Table 1: Measurements of runtime (ms) on different graphs for different number of threads

	Wikipedia Uniform	Ring Uniform	Wikipedia Geometric	Gowalla Uniform	Random Geometric
Sequential	928.306	5739.492	1029.305	167.324	525.452
Parallel (1 Thread)	1697.911	10176.145	1248.121	326.814	927.399
Parallel (2 Threads)	950.055	5045.508	763.896	366.517	579.642
Parallel (4 Threads)	598.000	2913.814	523.692	300.740	393.841
Parallel (8 Threads)	331.721	1662.195	343.212	267.261	282.128
Parallel (16 Threads)	218.807	1018.581	287.182	168.841	247.468

Figure 2: Speedup over the sequential implementation

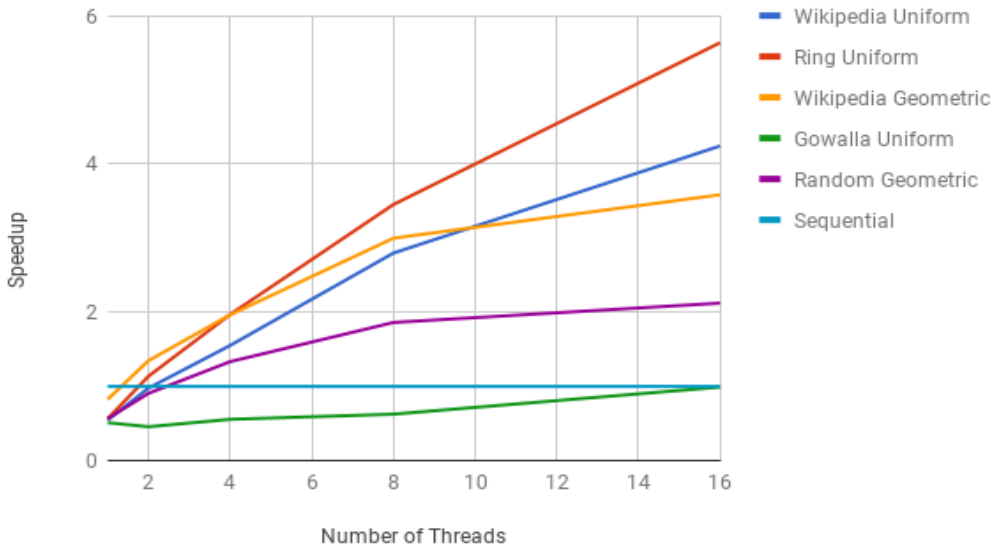


Table 2: Speedup over parallel single-threaded implementation

	Wikipedia Uniform	Ring Uniform	Wikipedia Geometric	Gowalla Uniform	Random Geometric
Parallel (2 Threads)	1.787	2.017	1.634	0.892	1.600
Parallel (4 Threads)	2.839	3.492	2.383	1.087	2.355
Parallel (8 Threads)	5.118	6.122	3.637	1.223	3.287
Parallel (16 Threads)	7.760	9.991	4.346	1.936	3.748

4.4 Analysis

Our speedup was mainly limited by synchronization overhead. We can break this into two parts. The first is the overhead of using atomic operations, even without multiple threads. In Tables 1, observe that the sequential Delta stepping (without atomics) runs significantly faster than the single-threaded parallel run time. Over the five graphs we test on, the single-threaded parallel implementation ran approximately 1.8x slower than the sequential implementation.

In addition, even when comparing the the performance of the single-threaded parallel code vs the multi-threaded parallel code, the bottleneck seems to be synchronization. Observe that in Table 3, when we break down the speedup of each portion of the query, the component requiring atomics, heavy edge relaxation, experiences a much smaller speedup than the synchronization-

free portions. Specifically, the speedup is 3.9x versus 7.2x and 6.0x. This is supported from a theoretical standpoint since atomic operations are costly, especially when there are many cores executing atomic operations simultaneously.

The performance of the algorithm varies from graph to graph and seems to work best with the uniform wikipedia and ring graph. We achieve 5.12x and 6.12x speedup with 8 threads for those two graphs. While this is below the ideal speedup, this is still above our initial goal of achieving half the ideal speedup. In addition, the choice of edge weight distribution is important since on the same graph, we see worse performance with weights that are pulled from the geometric distribution. One graph that Delta-stepping performs particularly poorly on is the Gowalla graph. It is likely that the topology of the graph is just not suited for Delta-stepping parallelism. In general, we only tested on large graphs because there was not enough parallelism to take advantage of for small graphs.

Table 3: Time spent in each query component for WikiGraph (1 vs 8 Threads)

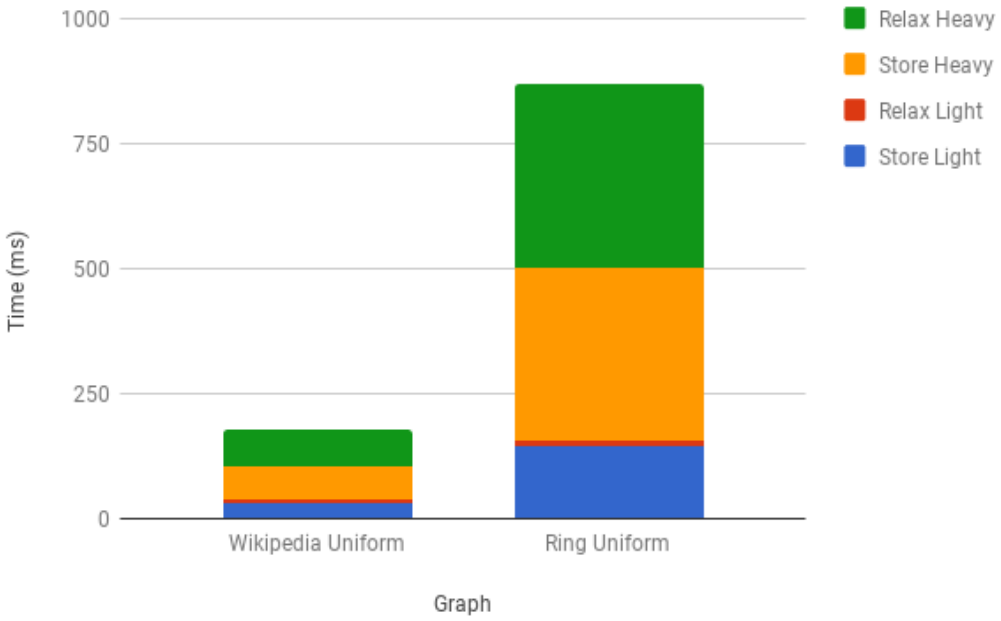
	Follow Light Edge	Relax Light Edge	Follow Heavy Edge	Relax Heavy Edge
1 Thread	212.5983	8.4274	680.9522	589.5097
8 Thread	35.4261	7.827	94.7919	150.0624
Speedup	6.0011x	1.0767x	7.1837x	3.9284x

The execution time can initially be broken up into two stages, the precomputation and the query. From the table below, we can see that a majority of time is spent in the query phase. We parallelized both of the phases but focused mainly on the query phase. The query phase can be further broken down into more stages. Gathering/storing the light edges, relaxing light edges, gathering/storing heavy edges, and relaxing heavy edges. As shown in the figure below, most of the time is spent in the store heavy and relax heavy stages, 38% and 42% respectively. An average of 17% of the time is spent on the the store light stage and 3% of the time is spent on relax light edges. This will vary from graph to graph depending on the edge distribution but the heavy stages will still dominate. While this is mostly due to our choice of Δ, where our testing indicated that lower Δ is still preferred over higher Δ despite the imbalanced in computation cost of each portion.

Table 4: Time spent in the precomputation and query stage of the algorithm

	Precomputation	SSSP Query
Wikipedia Uniform Sequential	103.489	395.648
Wikipedia Uniform Parallel (16 Threads)	37.381	181.426
Ring Uniform Sequential	728.062	5011.431
Ring Uniform Parallel (16 Threads)	147.4789	871.1025

Figure 3: Time breakdown for different stages in query phase



4.5 Choice of Machine Target

For this particular project, the choice of machine was sound because the divergent computation, and computation that operated on non-contiguous memory locations, would not have fit GPU's SIMD model well versus a multi-core processor.

5 Division of Work

Equal work was performed by both project members.

6 Code

Source code is available at https://github.com/davidkzeng/ParallelShortestPaths

7 References

1. https://www.cs.utexas.edu/~pingali/CS395T/2013fa/papers/delta-stepping.pdf

2. https://snap.stanford.edu/data/wiki-topcats.html

3. https://snap.stanford.edu/data/loc-gowalla.html