Parallelizing SSSP using Delta-Stepping

Daniel Li & Enrico Green

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

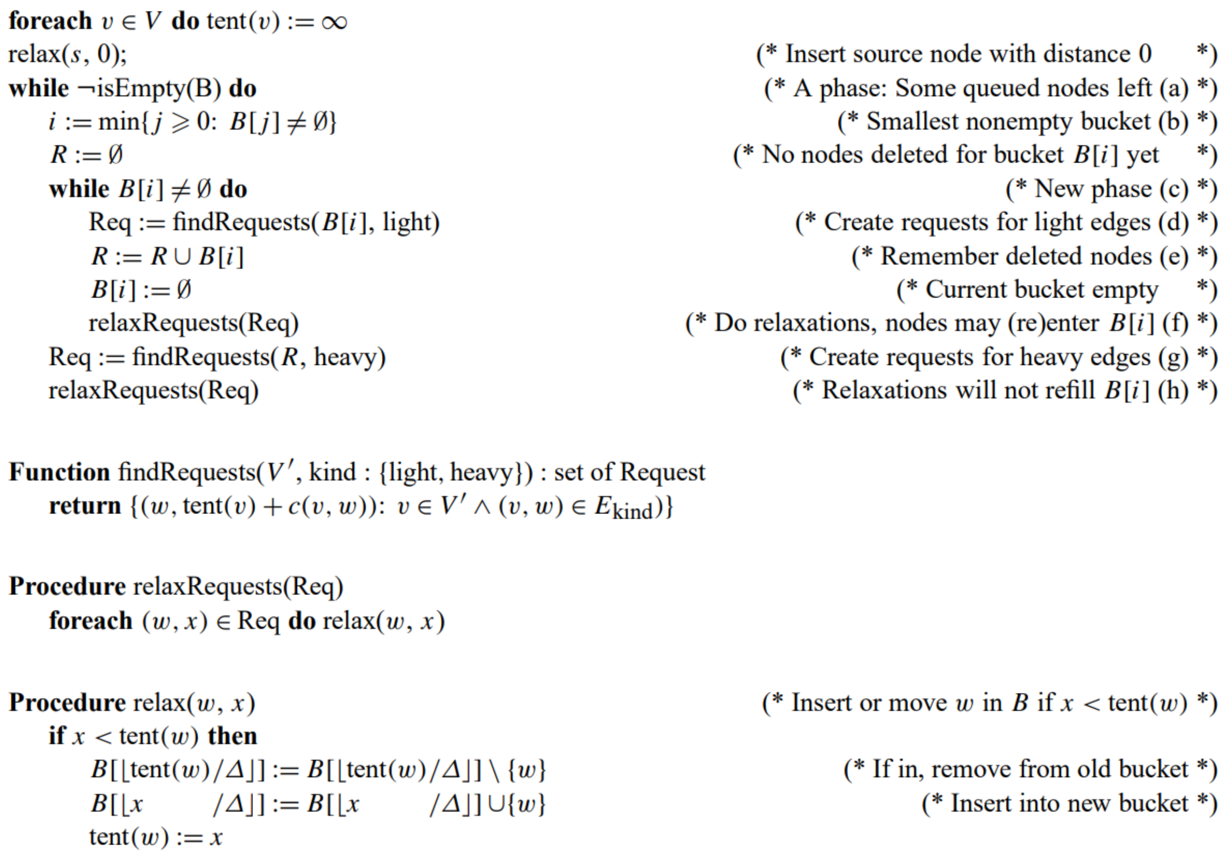
Summary

In this project, we implemented the Delta-Stepping algorithm using OpenMP on the CPU and CUDA on the GPU and compared the performance of the two implementations. We also implemented several classical algorithms used to solve the same problem to serve as benchmarks for the performance of our Delta-Stepping algorithms.

Background

The Single-Source Shortest Path (SSSP) problem is a graph theory problem which takes a graph as an input and outputs the shortest distance from a single "source" node to every other node in the graph. There are two well-known classical algorithms for solving this problem: Dijkstra's algorithm and the Bellman-Ford algorithm. Dijkstra's algorithm is a label-setting algorithm in which correctness depends on the vertices being processed in order of increasing distance from the source. Hence, it is inherently quite sequential, as processing multiple vertices at the same time may result in an incorrect output. The Bellman-Ford algorithm, on the other hand, is a label-correcting algorithm which admits a significant amount of parallelism. However, this parallelism comes at the cost of doing a large amount of redundant work – Bellman-Ford requires O(|V||E|) operations, while Dijkstra's algorithm only requires O(|E| log(|V|).

In Dijkstra’s algorithm, a priority queue is used to determine which node should be explored next. The Delta-Stepping algorithm instead uses buckets of width Δ. All nodes within a bucket are processed at the same time, and hence Delta-Stepping can admit a significant amount of parallelism on the right graphs (and with the right Δ value). Furthermore, by exploring nodes based on the partial ordering imposed by the buckets, the Delta-Stepping algorithm is able to avoid much of the redundant work done by the Bellman-Ford algorithm. By tuning the Δ parameter, a programmer can change the balance between parallelism and work-efficiency in order to maximize performance.



```
foreach $v \in V$ do tent($v$) := $\infty$
relax($s$, 0);                                    (* Insert source node with distance 0 *)
while ¬isEmpty(B) do                              (* A phase: Some queued nodes left (a) *)
    $i := \min\{j \geqslant 0\colon B[j] \neq \emptyset\}$       (* Smallest nonempty bucket (b) *)
    $R := \emptyset$                                (* No nodes deleted for bucket $B[i]$ yet *)
    while $B[i] \neq \emptyset$ do                     (* New phase (c) *)
        Req := findRequests($B[i]$, light)          (* Create requests for light edges (d) *)
        $R := R \cup B[i]$                          (* Remember deleted nodes (e) *)
        $B[i] := \emptyset$                          (* Current bucket empty *)
        relaxRequests(Req)                         (* Do relaxations, nodes may (re)enter $B[i]$ (f) *)
    Req := findRequests($R$, heavy)                (* Create requests for heavy edges (g) *)
    relaxRequests(Req)                             (* Relaxations will not refill $B[i]$ (h) *)

Function findRequests($V'$, kind : {light, heavy}) : set of Request
    return $\{(w, \text{tent}(v) + c(v, w))\colon\ v \in V' \wedge (v, w) \in E_{\text{kind}})\}$

Procedure relaxRequests(Req)
    foreach $(w, x) \in$ Req do relax($w$, $x$)

Procedure relax($w$, $x$)                          (* Insert or move $w$ in $B$ if $x <$ tent($w$) *)
    if $x <$ tent($w$) then
        $B[\lfloor \text{tent}(w)/\Delta \rfloor] := B[\lfloor \text{tent}(w)/\Delta \rfloor] \setminus \{w\}$   (* If in, remove from old bucket *)
        $B[\lfloor x/\Delta \rfloor] := B[\lfloor x/\Delta \rfloor] \cup \{w\}$                 (* Insert into new bucket *)
        tent($w$) := $x$
```

Figure 1: Pseudocode of Delta-Stepping Algorithm (Meyer and Sanders, 1998)

Graph algorithms are notoriously difficult to parallelize effectively due to the fact that many graph algorithms have a relatively low arithmetic intensity and are highly iterative in nature. Because of this, performance may vary wildly depending on the input graph. For example, running a parallel SSSP algorithm on a tree will offer a 2x speedup in the absolute best case (because the algorithm can discover 2 nodes in each iteration instead of 1), while running the same algorithm on a graph with a high edge density will produce much better results. Previous research has shown that Δ-stepping has no known worst-case bounds for general graphs (Dong et. al, 2021). For this project, we focused primarily on improving performance for graphs with "many" edges, as this offers the greatest opportunity for parallelism across both edges and nodes. However, to demonstrate the effect of different input graphs on parallel graph algorithms, we will also include some results from running our algorithms on graphs from which we expect poor performance.

Approach

There were three places in which we attempted to introduce parallelism to the Delta-Stepping algorithm: edge preprocessing, finding requests, and relaxing requests. Delta-Stepping requires a significant amount of preprocessing, as each edge must be classified as "light" or "heavy" (based on whether its weight is less than Δ) and grouped by source vertex. By having different threads process edges incident on different vertices in parallel, we can achieve a significant speedup in this step.

Two of the Delta-Stepping algorithm's helper functions (findRequests and relaxRequests) also admit opportunities for parallelism. FindRequests iterates through all the vertices in a bucket and adds all adjacent vertices connected by edges of a given type (either light or heavy) to a set of requests. While this set is a shared data structure, we were able to find workarounds to parallelize this step in both our OpenMP and CUDA implementations. On the other hand, relaxRequests takes the set of vertices discovered by findRequests and iterates through each one, updating tentative distances and moving vertices between buckets. This was parallelized pretty easily using OpenMP, but we were unable to do so effectively using CUDA due to the fact that buckets needed to be stored on the CPU in order to ensure correct control flow for the algorithm. However, we were still able to perform a reduction on the requests to filter out only the requests which cause a relaxation, taking advantage of the parallel computing power of the GPU.

In order to implement the Delta-Stepping algorithm, we needed to make use of several shared data structures – buckets (modeled as a C++ set), an array of tentative distances, and sets of ongoing requests are all accessible by multiple threads in the pseudocode outlined above. Meyer and Sanders discuss the use of lock-free buffers where processing units use “randomized dart-throwing” to atomically insert values into empty locations in the buffer, retrying repeatedly until there is no collision. We elected to use fine-grained mutexes to protect the buckets and tentative distances, and resorted to several enhancements outlined in the “approach” section below to avoid data races and contention for the sets of requests.

OpenMP Algorithm

Our initial OpenMP implementation of Delta-Stepping was quite faithful to the pseudocode from Meyer and Sanders. However, we quickly found that this didn’t offer a significant speedup over a sequential implementation of Delta-Stepping – while the relaxation phase was significantly faster than the sequential version, the request-finding phase saw almost no speedup due to contention over the shared request set.

In order to achieve a significant speedup in the request-finding phase, we decided to convert the set of nodes in the bucket into a vector before doing any processing. Since OpenMP’s parallel-for construct doesn’t support C++ sets, we had previously been using tasks for this parallelization, but since each task corresponded to only one node, this was creating a lot of contention for the shared set (as each thread would have to acquire a mutex to the set each time it processed a node). By swapping to the parallel-for construct, we were able to store the newly generated requests locally, thus removing the need for threads to contend for access to the shared set.

We also considered “reducing” the local sets to a shared set at the end of the request-finding phase. However, we found that this would entail a significant amount of memory movement and unnecessary serialization, as we would need to use a mutex to avoid race conditions while copying the entries of the local sets to the shared set. Instead, we modified our relaxation phase to make use of the local request sets generated in the finding phase directly.

After going through the process of figuring out the above optimizations, we think that OpenMP is a very effective framework for implementing graph algorithms. This is because using shared memory is very convenient for graph algorithms and because OpenMP offers very natural control over the control flow of the program. In delta-stepping, for example, the inner while-loop must repeat until the current bucket is empty. In CUDA or MPI, this would require a significant amount of communication (between the CPU and GPU in the case of CUDA, or between processors in the case of MPI) in order to determine when the loop can be exited. Using OpenMP, however, performing this check and leaving the loop is trivial. OpenMP suffers somewhat from not offering the same performance at scale as MPI or CUDA (modern GPUs have thousands of CUDA cores, and supercomputers may use MPI to communicate between hundreds of processors), but since graph algorithms require traversing the graph sequentially and hence offer limited parallelizability, this downside is less pronounced when working with graphs than when working on other problems.

CUDA Algorithm

Using CUDA greatly increases the maximum number of available parallel processing units at the cost of preventing fine-grained control techniques, such as mutex locks. In the Delta-Stepping algorithm, there is contention in finding the next empty bucket, computing the set of requests to process, updating the relaxed vertex distances, and moving vertices between buckets. To adapt the algorithm to work with CUDA, we would like each thread to produce its own relaxation requests without serializing on some central data structure, but we will need to communicate request distances for the same vertex so we know which one to update. Hence, our strategy for the implementation was to slightly increase total work to maximize the parallelizability of finding requests.

First, the host finds the set of vertices in the current bucket to iterate. This list is passed to the GPU. The device generates the set of relaxation requests in parallel, with each thread reading the data of one edge. Each thread does this by performing a binary search through the vertices to determine the corresponding vertex and edge. Then, each thread computes the new distance to the destination vertex of the request. We use a parallel sort to find the minimal distances for each destination vertex, which gives us the set of vertices and new distances to those vertices. The relaxations which decrease the tentative distance are collected (using a prefix sum) and passed to the host, which moves vertices to their new buckets. We find the next non-empty bucket and loop.

This strategy admits a high amount of parallelism, and performs the computations and data transfer in $O(n_\Delta + (R \log R) / p)$ time, where n_Δ is the number of nodes in the bucket, and R is the number of requests generated in this step. We introduced a factor of log R in total work compared to the OpenMP implementation, but this results in a practical speedup for many cases. This algorithm performs worse asymptotically, whereas OpenMP can use efficient data structures and communication strategies to perform better.

One of the main challenges with CUDA was performing the updates on the buckets at the end of each iteration. One way to find the minimum non-empty bucket is to have processors scan through the buckets in parallel with stride and track the first local non-empty bucket found, then use a reduction to find the minimum. However, since CUDA cannot adequately represent the bucket set structures, we must pass to the CPU to perform the bucket updates. Thus, the bucket scan suffers from reduced parallelization. In cases where delta is proportional to the maximum edge weight, the number of buckets is constant, so this step is not a significant portion of the running time.

We attempted to add in OpenMP to parallelize the bucket updates in the last step. With some testing, we found that the additional speedup is minimal. This may be due to the overhead of spawning a gang of threads to process requests in addition to the lock contention on buckets caused by multiple threads moving nodes between buckets. In the OpenMP implementation, the use of local requests covers some of the cost, and the requests do not only contain relaxations, so this might explain the difference in benefit in multithreading.

Results

In order to measure performance for this project, we created a Python script which creates a representation of a graph (encoded as the number of vertices and a list of edges) in a file. This script was modified to create a variety of different graph types, such as trees, cycles, complete graphs, regular graphs, as well as completely "random" graphs. As mentioned above in our background section, we will focus on the results from the graphs with a relatively high edge density (that is, the complete, regular, and random graphs), as they benefit the most from parallelism, but we think that including some data from other graphs is valuable as well.

Graph name	Description
complete-1k	Complete graph with 1000 vertices
random-20k	20,000 vertex graph with 50,000 edges
local-100k	100,000 vertex graph with average degree 33. Ordered vertices, with edges to nearby vertices in front and random weights proportional to difference in position. Simulates road network.
tree	Random tree with 100,000 vertices
cycle	Cycle with 100,000 vertices
reg-X-Yp	Graph with X vertices, with Y% of edges filled.

Figure 2: Graph Descriptions

All of our testing and implementation was done on the GHC machines with an 8-core Intel i7-9700 CPU and an NVIDIA GeForce RTX 2080. Note that we did not include initialization time in our data, which means that neither the time spent sorting the edges between light and heavy (which can be done as the graph is being loaded from a file) nor the time required to move a graph from the CPU to a GPU is being considered. In a real-world scenario, the performance of the CUDA solution would likely be significantly restricted by the bandwidth between the CPU and GPU.

In order to test performance, we implemented Dijkstra's algorithm, the Bellman-Ford algorithm, and a sequential version of the Delta-Stepping algorithm to serve as benchmarks. The sequential algorithm follows the pseudocode presented in the original paper with some changes to data structures. We calculated speedup primarily based on the performance of the sequential delta-stepping implementation, but we think that being able to compare performance to Dijkstra's and Bellman-Ford is valuable as well. As such, we included performance for these two algorithms in the chart below, while the remaining charts/graphs will focus on the speedup from parallelizing Delta-Stepping.

	Dijkstra	Bellman-Ford	OpenMP Δ	CUDA Δ
complete-1k	0.1819	1.5167	0.0175	0.0056
random-20k	0.0023	3.1203	0.0002	0.0003
local-100k	0.0169	(timeout)	0.0006	0.0038
tree	0.2354	(timeout)	0.1222	0.0204
cycle	0.1564	(timeout)	0.6688	7.0821

Figure 3: Wide variety of tests with Dijkstra's and Bellman-Ford

On well-connected graphs with high and low degree, Delta-stepping appears to have a xsignificant speed increase compared to sequential dijkstra. Bellman-Ford scales with the number of vertices and edges and quickly becomes too slow to run using OpenMP. We notice that for the cycle graph, which contains 100,000 vertices in a cycle, the performance of CUDA degrades severely. Our profiling shows that almost all of the time is spent in the "find" step, where we only ever have one edge in an iteration, but need to initialize a block on the GPU to handle that single edge. There could be some unforeseen bottleneck which causes this to have significantly worse performance than local-100k, despite having the same number of vertices.

	Dijkstra	Sequential Δ	OpenMP Δ	CUDA Δ
reg-5k-5p	0.6364	0.3631	0.0512	0.0111
reg-5k-10p	1.2118	0.6897	0.1017	0.0167
reg-5k-25p	2.3251	1.3122	0.1825	0.0272
reg-2.5k-5p	0.1533	0.0898	0.0149	0.0058
reg-5k-5p	0.6364	0.3631	0.0512	0.0111
reg-10k-5p	2.6545	1.2807	0.2149	0.0297

Figure 4(a): Regular graphs with varying vertex counts and edge densities

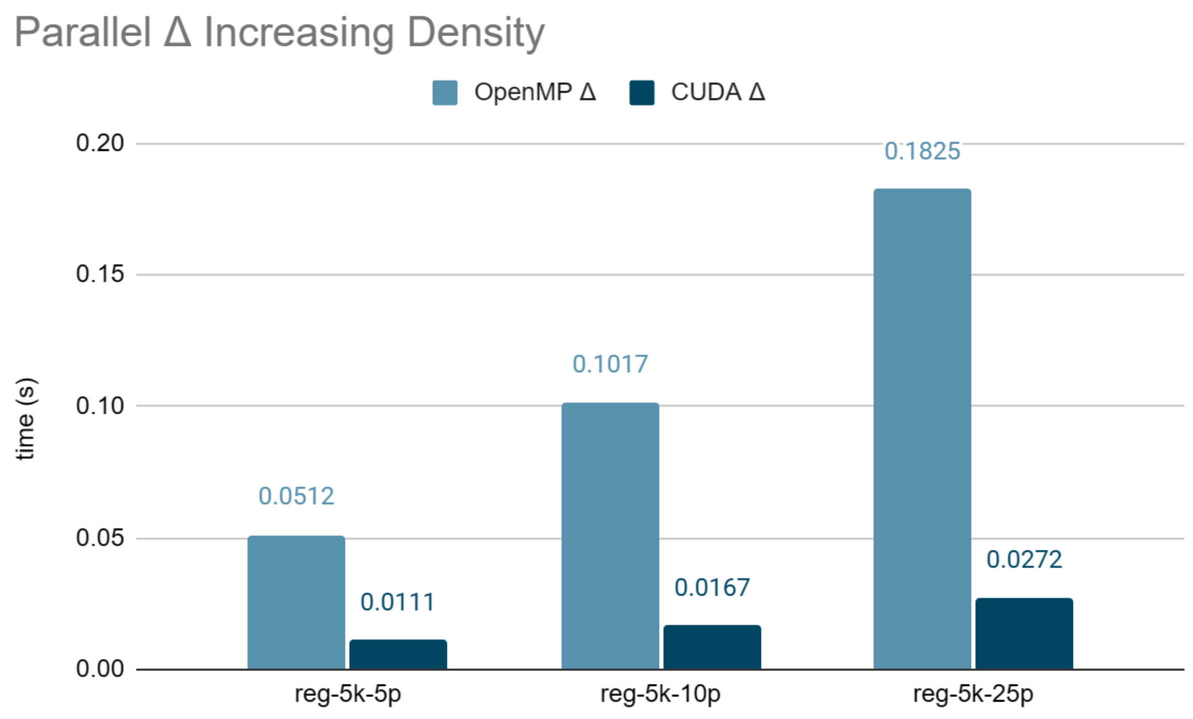


Figure 4(b): Runtime of parallel Delta-Stepping increases with density of edges. The number of edges is proportional to the density, 5%, 10%, and 25%. Running time increases slightly less than proportional to number of edges.

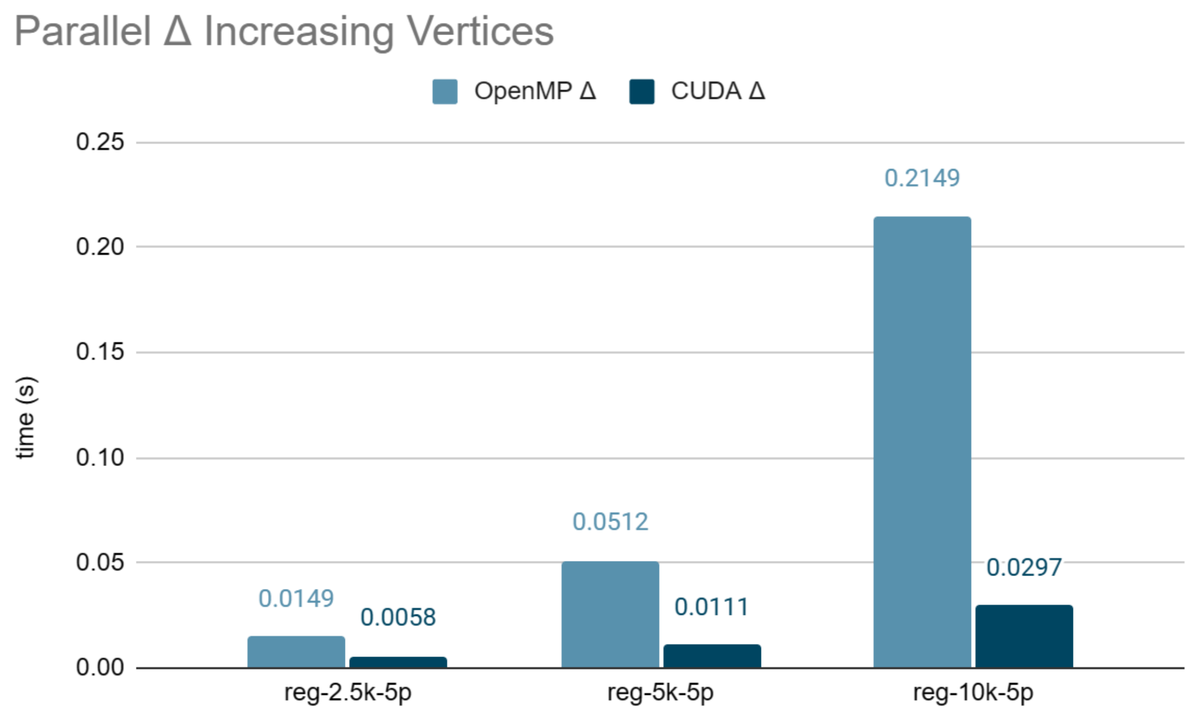


Figure 4(c): Runtime of parallel Delta-Stepping increasing with number of vertices. The number of edges increases by a factor of 4x between each pair.

The runtime of Dijkstra and sequential Delta-stepping both increase approximately in accordance with the asymptotic runtime, O(|E| log|V|). The parallel algorithms, OpenMP, and CUDA, scale slightly below the expected ratio. This may be due to the overhead of using OpenMP and CUDA, such as spawning threads, copying memory, and synchronization.

While the Δ parameter is a value corresponding to the "width" of buckets, we found it easier to think about the number of different buckets a destination vertex can "land" in when relaxing a request. Since different input graphs may have different edge weight distributions, choosing a constant value of Δ to use doesn't make much sense. Buckets in our implementation were reused after being emptied, and this value was equal to the total number of buckets in memory at any given time. Hence, we referred to this heuristic as the number of buckets.

# Buckets	Sequential Δ	OpenMP Δ	Speedup	CUDA Δ	Speedup
1	1.6495	0.2658	6.20x	0.0267	61.8x
5	0.6162	0.1263	4.87x	0.0155	39.7x
10	0.4816	0.1070	4.50x	0.0152	31.7x
20	0.4057	0.0853	4.76x	0.0123	33.0x
40	0.3325	0.0577	5.76x	0.0109	30.5x
80	0.3631	0.0512	7.09x	0.0111	32.7x
160	0.3424	0.0493	6.95x	0.0119	28.8x
320	0.3385	0.0502	6.74x	0.0142	23.8x
1000	0.3400	0.0487	6.98x	0.0173	19.7x

Figure 5: Modifying Δ value on random reg-5k-5p

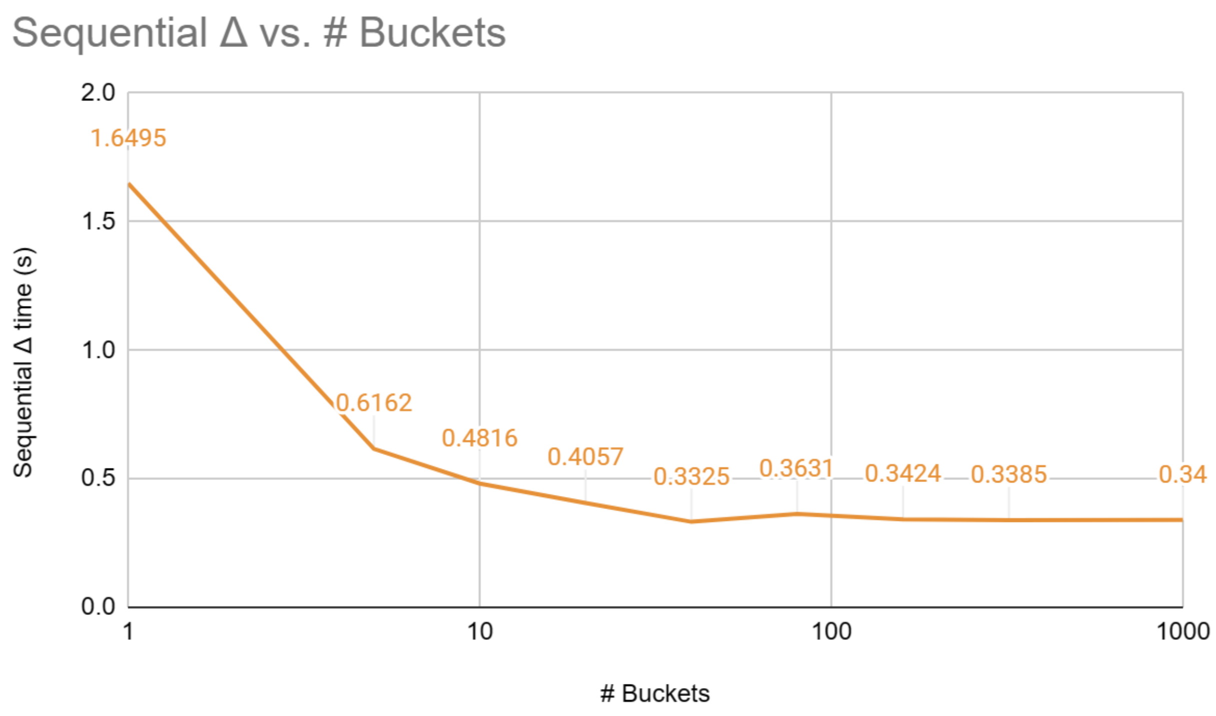


Figure 6(a): Runtime of sequential Delta-Stepping when modifying Δ value.

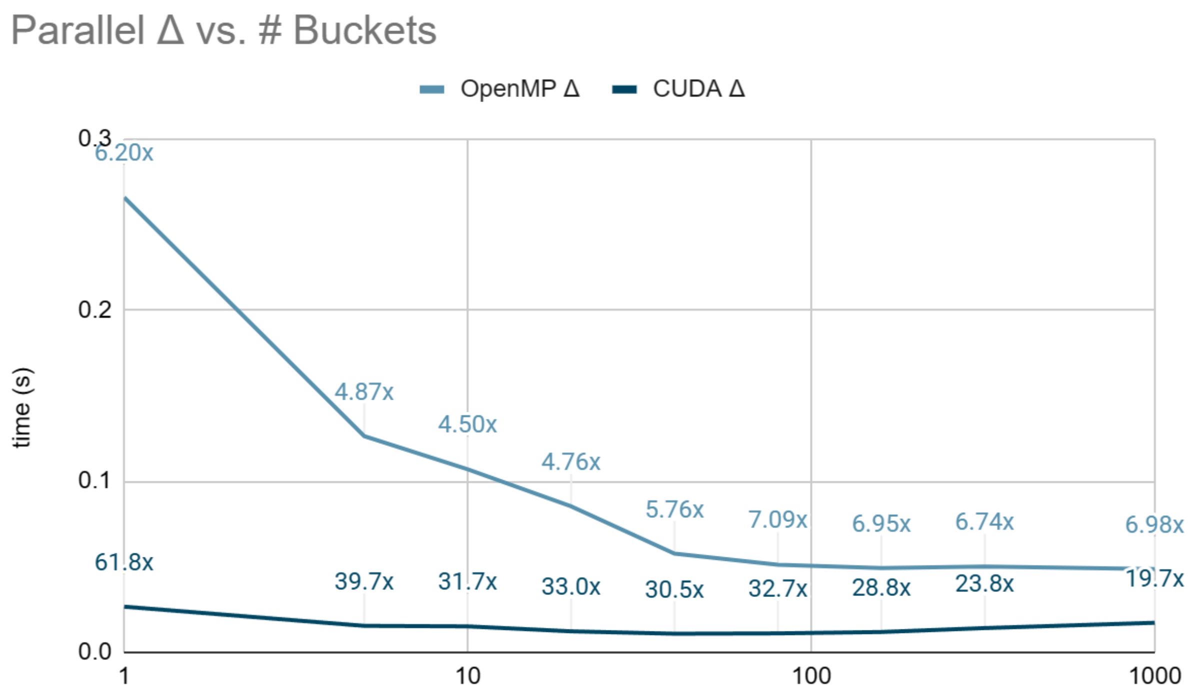


Figure 6(b): Runtime of parallel Delta-Stepping when modifying Δ value. The labels correspond to the speedup when compared to Sequential Δ.

When there is only one bucket, the algorithm is similar to Bellman-Ford. When edges are integral and delta is 1, the algorithm is similar to Dijkstra (Meyer, 1998). Fewer vertices end up in the same bucket if the number of buckets is large, reducing the available parallelism. We see that the speedup with CUDA decreases as the number of buckets increases. The speedup achieved by OpenMP also flattens out at higher bucket counts.

With profiling, we find that 80 buckets appears to be the sweet spot for both the find and relaxation steps for CUDA. The performance for OpenMP also began to plateau at around 80 buckets. Thus, we used this data to choose a default bucket count. For this graph specifically, the edge weights were generated uniformly from integer values between 1 and 1000, so it may introduce biases towards certain values of buckets. In particular, with *n* distinct values for edge weights, it never makes sense to have more than *n* buckets. With other distributions of edge weights, different values for delta may provide better performance.

It is worth mentioning that the implementation of the algorithm is slightly different in OpenMP compared to the sequential version, which leads to slight increases in performance. Hence, the speedup for OpenMP Δ is slightly higher than the speedup compared to running the OpenMP with one thread. We see the speedup of OpenMP versus the thread count in Figure 7. We recall from above that the Sequential Δ runtime is 0.3631s, so there is an additional speedup even with one thread.

# Threads	Find	Relax	Total	Speedup
16	0.0308	0.0173	0.0672	3.61x
8	0.0235	0.0246	0.0512	4.74x
4	0.0439	0.0407	0.0879	2.76x
2	0.0853	0.0626	0.1516	1.60x
1	0.1688	0.0714	0.2425	1.00x

Figure 7: Runtime of OpenMP with different number of threads. Also shows time spent in each part of the algorithm, finding requests and relaxing. We see that Find has more parallelization, as it enjoys more speedup compared to Relax as the number of threads increases.

We can see from the above table that the OpenMP implementation does not scale linearly with the number of threads. This is to be expected due to contention over the vertex/bucket locks, as well as potential workload imbalances caused by “unfilled” buckets. Since these tests were run on an 8-core CPU, we see a performance loss with 16 threads due to unnecessary context switching.

We note that Relax has poorer speedup because it needs to remove a vertex from a bucket and add it to another bucket in one atomic action, which we solved using locks. To improve the speedup, we might be able to consider the use of lock-free data structures to separate all relaxation requests by their destination vertex, and partitioning the vertex set so that each thread can maintain a local collection of buckets. This would increase contention in Find, so there is a performance tradeoff.

Conclusion

The Delta-stepping algorithm provides good parallelizability for the SSSP problem, allowing us to achieve reasonable speedup against the single-threaded implementation while being faster than Dijkstra’s algorithm for many randomly generated graphs. Both the OpenMP and CUDA approaches provide strong tools to increase speedup, but face different issues for synchronization. The main difficulty of parallelizing comes from relaxing an edge and moving vertices between buckets. We were able to tackle this challenge by using mutex locks for OpenMP and passing results to the host for CUDA. We were able to design a suitable algorithm for CUDA which slightly increases the total work but gets high speedup on our test cases. We were able to tune parameters to enhance performance on our distribution of random input data, and measure our overall speedup against existing algorithms.

References

U. Meyer, P. Sanders, Δ-stepping: a parallelizable shortest path algorithm, Journal of Algorithms, Volume 49, Issue 1, 2003, Pages 114-152, ISSN 0196-6774, https://doi.org/10.1016/S0196-6774(03)00076-2.

Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing ordered graph algorithms with GraphIt. In Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020). Association for Computing Machinery, New York, NY, USA, 158–170. https://doi.org/10.1145/3368826.3377909

Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21). Association for Computing Machinery, New York, NY, USA, 184–197. https://doi.org/10.1145/3409964.3461782

List of Work & Credit

Distribution of credit - 50/50

Task	Completed By

Graph Generation Script	Enrico
Correctness testing & timing code	Enrico
Dijkstra’s Algorithm	Daniel
Bellman-Ford Algorithm	Daniel
Initial Parallel OpenMP Implementation	Both
Parallel OpenMP Optimization	Both
Initial CUDA Implementation	Daniel
CUDA Debugging & Optimization	Both
Final Report	Both
Poster	Enrico