Introducing a completely Lock-Free Parallel Delta-Stepping Algorithm

Ha Duy Nguyen

June 8, 2025

Abstract

This report presents the implementation and analysis of a lock-free parallel delta-stepping algorithm for solving the single-source shortest path problem. The implementation focuses on achieving high performance through efficient thread pool management and careful synchronization techniques. We evaluate the performance across various graph types and sizes, comparing different delta values and thread counts. Compared to sequential delta-stepping, parallel delta-stepping provides significant speed up. at least the same performance (with the right delta and thread count) as sequential Dijkstra's algorithm in all tested graph types, and moderate to significant speedup in some of them.

Contents

1 Introduction

The suggested paper "Engineering a Parallel Δ-stepping Algorithm" by Duriakova et al. proposed two distinct implementations of the parallel delta-stepping algorithm that differ fundamentally in their approach to workload distribution and synchronization:

Static Partitioning Implementation: This approach employs a static assignment of nodes to threads at the algorithm's initialization phase. Each node is randomly allocated to a specific thread, and throughout the algorithm's execution, only the assigned thread can perform relaxation operations on edges leading to that node. The implementation partitions all data structures (buckets B_i, request lists R_l and R_h) across threads, where each thread maintains its own local portions: $B_i = \cup_t B_i^t$ and $R_l = \cup_{t,r} R_l^{t,r}$. This design eliminates race conditions entirely since write operations are restricted to the owning thread, thus avoiding the need for any locking mechanisms. However, the static assignment can lead to workload imbalance, as the distribution of active nodes and edge relaxations may not be perfectly uniform across threads.

Dynamic Partitioning Implementation: In contrast, this approach utilizes global data structures (B_i, R_l, R_h) shared among all threads and dynamically distributes work during execution. Nodes and relaxation requests are allocated to threads on-demand during bucket processing phases, enabling better load balancing. The implementation incorporates the "strictest request" optimization, maintaining only the most beneficial relaxation request for each node to minimize redundant work. However, this flexibility comes at the cost of requiring extensive synchronization through locking mechanisms to prevent race conditions when multiple threads attempt to update the same node's tentative distance concurrently. Additionally, bucket updates require careful coordination using prefix-sum operations to ensure thread-safe insertions.

The key trade-off between these approaches lies in the balance between synchronization overhead and workload distribution. While dynamic partitioning achieves near-perfect load balance (typically $<$ 0.002% imbalance), it can spend up to 20% of execution time waiting for locks. Conversely, static partitioning tolerates moderate workload imbalance ($4 - 8\%$) but eliminates synchronization bottlenecks entirely.

Clearly, for general cases, the dynamic partitioning implementation is preferred, as it can efficiently handle the workload imbalance that might be present in some graph classes. Ideally, we would want to have good load balancing and low synchronization overhead. I have extensively studied different approaches to achieve this (... with too many ideas, hence the overdue submission). In the end, I managed to implement a lock-free dynamic algorithm, which achieves the best of both worlds. This report will detail the optimizations used, and the results compared to the sequential delta-stepping algorithm, and Dijkstra's algorithm.

1.1 Original Algorithm

The parallel delta-stepping algorithm suggested by the paper consists of the following steps:

The algorithm maintains the following invariants:

- A vertex v is in bucket B_i if $i = \lfloor d^*(v)/\delta \rfloor$
- Processing buckets in ascending order ensures correctness
- Light edges are processed before heavy edges to minimize bucket re-insertions

For the dynamic partitioning implementation, the variables B, R_l, R_h are shared among all threads. The "strictest request" optimization is used, which means that only the most beneficial relaxation request for each node is maintained. This will ensure during the relaxation phase, each node will be considered at most once. This is beneficial, as it eliminates the need for synchronizing d^*.

The original approach is to use thread-safe data structures for R_l and R_h, per-node locks for the strictest request optimization, thread-safe bucket structure B_i, and per-thread information for node insertion. While this approach is simple and easy to implement, it is not efficient. The main bottleneck is the synchronization overhead of the strictest request optimization, the cost of node removal/insertion tracking, and the cost of the prefix-sum operation for bucket updates. Furthermore, using per-node locks for the strictest request optimization is not scalable, as it will not scale to large number of threads.

Algorithm 1 Parallel Δ-stepping SSSP

Require: Graph $G = (V, E)$, source $s \in V$, $\delta > 0$

for $v \in V \setminus \{s\}$ **do**
  $N_v^l \leftarrow \{(u, w) \in E \mid u = v \wedge w < \delta\}$ ▷ Light edges
  $N_v^h \leftarrow \{(u, w) \in E \mid u = v \wedge w \geq \delta\}$ ▷ Heavy edges
  $d^*(v) \leftarrow \infty$
  $B_\infty \leftarrow B_\infty \cup \{v\}$
end for
$B_\infty \leftarrow B_\infty \setminus \{s\}$
$B_0 \leftarrow B_0 \cup \{s\}$
$d^*(s) \leftarrow 0$
while $\exists i < \infty : B_i \neq \emptyset$ **do**
  $k \leftarrow \min\{i : B_i \neq \emptyset\}$
  $S \leftarrow B_k$
  while $S \neq \emptyset$ **do**
    for $v \in S$ **in parallel do**
      $B_k \leftarrow B_k \setminus \{v\}$
      $R_l \leftarrow R_l \cup \{(v, w) \mid (v, w) \in N_v^l\}$
      $R_h \leftarrow R_h \cup \{(v, w) \mid (v, w) \in N_v^h\}$
    end for
    for $(v, w) \in R_l$ **in parallel do**
      if $d^*(v) + w < d^*(w)$ **then**
        $i \leftarrow \lfloor d^*(w)/\delta \rfloor$
        $j \leftarrow \lfloor (d^*(v) + w)/\delta \rfloor$
        $B_i \leftarrow B_i \setminus \{w\}$
        $B_j \leftarrow B_j \cup \{w\}$
        $d^*(w) \leftarrow d^*(v) + w$
      end if
    end for
    $S \leftarrow B_k$
  end while
  for $(v, w) \in R_h$ **in parallel do**
    if $d^*(v) + w < d^*(w)$ **then**
      $i \leftarrow \lfloor d^*(w)/\delta \rfloor$
      $j \leftarrow \lfloor (d^*(v) + w)/\delta \rfloor$
      $B_i \leftarrow B_i \setminus \{w\}$
      $B_j \leftarrow B_j \cup \{w\}$
      $d^*(w) \leftarrow d^*(v) + w$
    end if
  end for
  $R_l \leftarrow \emptyset$
  $R_h \leftarrow \emptyset$
end while
return d^*

2 My Approach

This part is dedicated to showing all observations and optimizations that I have made to the original algorithm in order to make it completely lock-free.

2.1 Re-using buckets

The first observation is that we don't need to allocate buckets infinitely. In fact, we can prove that we only need $MAX\_BUCKETS = \lceil c^*/\delta \rceil + 1$ buckets, where c^* is the maximum edge weight in the graph. This is because:

- If we reach bucket i, it means that all vertices in buckets $0, 1, \ldots, i-1$ are already processed, and these buckets are empty.
- A request with distance d can only change buckets up to $\lceil d/\delta \rceil$ away, as $d \leq c*$, the amount of active buckets at any moment cannot exceed $\lceil d/\delta \rceil + 1$.

This means we can allocate a fixed number of buckets at the start, and reuse them throughout the algorithm. We can wrap around when we reach the end of the array, and stop when we have traversed $MAX\_BUCKETS$ buckets without finding any non-empty one.

2.2 Making R_l and R_h lock-free

R_l and R_h are apparently the easiest data structures to make lock-free. In fact, we can represent them as a concurrent stack, which is a well-studied problem. Given the algorithm, we can see that there won't be any push and pop operations on them at the same time, i.e only push operations (in phase 1) or pop operations (in phase 2 and 3), so we can easily implement them using a single atomic counter to track the size of the stack. It is possible to use an `std::vector` of fixed size $|V|$ to represent the stack, with an atomic integer counter to track its (logical) size. Due to the strictest request optimization, we can see that the maximum number of requests at any stage is $|V|$, as each vertex can only generate at most one, so fixing this size is correct. When pushing, I use the atomic counter to get the next available position, and update that location in the vector. This is thread-safe, as the vector has already been pre-allocated. To clear, simply set the counter to 0. Iterating over it is also easy,

so dividing the vector into chunks to feed to each thread is trivial.

2.3 Lock-free strictest request optimization

Making the strictest request optimization is the trickiest part of the algorithm. At this stage, we might have multiple requests with the same neighbor node v, and we need to only keep the request with the minimum distance $d^*(u)+w$. We have these observations:

- Denote map_u the map from vertex u to the best possible request (i.e. lowest distance) generated by u so far. Then map can be represented as an `std::vector` of size $|V|$.
- We don't need to represent a request as a triplet u, v, w, as a pair $v, d^*(u)+w$ is enough. In fact, if we know the node v that generated the request, we only need to know the distance $d^*(u) + w$. Fortunately, from the definition of map, we already know that v generates map_v. So we can simply represent map_v as a double.
- We need to be able to do concurrent updates on map_u. When receiving a request, we first check if map_u is already set. If it is, we compare the new distance with the old one, and update it if the new one is lower. If it is not set (which will mean that map_u is ∞ by convention), we simply set it to the new one. All these operations can be done a combination of operations on atomic variables (and this kind of problem is also well-studied). So, we can represent map as an `std::vector<std::atomic<double»`.

Knowing that we can represent a request simply as a double is the most important observation. It allows us to wrap the request with `std::atomic`, which is not possible had the request been any larger (as the maximum size supported by truly atomic CAS is 64 bits).

Finally, we will change the purpose of R_l and R_h a bit. Instead of containing the requests, they will contain the nodes that generated the requests instead. Now, to extract a request, we pop an integer v from R_l and R_h, and use it to index into map to get the request (which is a double specifying $d^*(u) + w$). We have successfully obtained the pair $(v, d^*(u) + w)$ for the relaxation phase, completely lock-free. After relaxation of v, we simply set map_v back to ∞. Rinse and repeat.

2.4 Implementing Lock-free B_i

2.4.1 A lock-based solution

The paper suggests using a dynamic array for B_i. Removing an vertex from the bucket is as simple as marking its value to null (which is always thread-safe). Inserting into the bucket is done by appending it to the array. This creates two issues: internal fragmentation within a bucket (the non-null elements are not contiguous), and inserting is not thread-safe itself. We will wrap an std::vector with a std::mutex, and lock it for every operation that changes its size, obtaining a thread-safe concurrent vector. We will compare our final solution with this one to demonstrate the performance gain by making B_i lock-free.

2.4.2 A lock-free solution

In the paper, the authors suggest that "when the proportion of gaps reaches a threshold, a bucket is collapsed to reuse empty space", but we found out that collapsing buckets is not necessary. The following observation is the key to making B_i lock-free.

Assume we already know the size of the bucket. Then, we can pre-allocate a fixed-size array for each bucket B_i, and perform insertions lock-free by keeping an atomic (logical) tail index of the array, similar to how we use R_l and R_h. This will make it lock-free, and thus making the entire algorithm lock-free.

There are only $|V|$ vertices, so each bucket only need $O(|V|)$ space for this algorithm to work. Furthermore, we claim that it is possible to make it work using buckets of size exactly $|V|$, even counting internal fragmentation! I will prove this in my defense, but the general idea is that only kind of internal fragmentation that can "harm" a fixed-size bucket is the internal fragmentation caused by requests that do not move the vertex to another bucket. So, we can simply choose not to move the vertex to its own bucket again (thus causing "harmful" fragmentation). Fragmentation caused by moving a node from a bucket to another is not "harmful", as a node can be inserted into a bucket only once, while (logically) removed only once. This essentially shows that it is possible to fix

$|V|$ as the bucket size. This makes up an $O(c^*/\Delta|V|)$ space complexity algorithm.

```
relax(v, map):
    double new_dist
        = map[v].exchange(INF);
    if (new_dist < dist[v]):
        int old_b = get_bucket(v);
        dist[v] = new_dist;
        int new_b = get_bucket(v);
        if (old_b != -1
        && old_b != current_generation
        && old_b != new_b):
            remove (logically) v
            from bucket old_b,
            O(1) if we save the position
            of v in the bucket
        if (old_b == current_generation
        || old_b != new_b):
            insert v into bucket new_b,
            saving its position
            in the bucket
```

We have now a completely lock-free delta-stepping algorithm.

2.5 Implementing a Thread Pool

Creating threads is costly, as we have seen in class (TD6). We will use a thread pool to create threads, and reuse them. The thread pool will be based on a concurrent, single-producer multiple-consumer (SPMC) queue. The producer will be the main thread, and the consumers will be the threads that will be used to run the algorithm. The queue will be used to store the tasks that need to be executed. The thread pool will be used to run the algorithm in parallel.

Based on my thread pool implementation from TD6, I improved it to be more efficient, with support for synchronization, i.e waiting for all tasks to finish. This is done by using an atomic active worker counter, and waiting on an atomic "running" boolean flag. This is achieved with C++20's std::atomic::wait feature. The thread pool works with both blocking and non-blocking queues. This is FlexiblePool, implemented in `src/ds/pools/flexible_pool.h`.

Additionally, I implemented FastPool, a thread pool that synchronizes using std::barrier, another C++20 feature. This is implemented in `src/ds/pools/fast_pool.h`. This is also what I use by default in my benchmarked code.

For reference, I also grabbed an open-source, so-called "production-grade" thread pool from GitHub, which is in `src/pools/lib/BS_thread_pool.h`.

All thread pool implementations are designed to be compatible with queues based on the base class in `src/ds/thread_safe_queue_base.h`. I have implemented several queue types, including a coarse-grained queue that locks the entire queue for every operation (seen from TD4), a Michael-Scott Head-Tail lock-based queue, a simple SPMC lock-free queue using std::atomic, and a queue based on two lock-free stacks. You can find all of them in `src/ds/queues`. Unfortunately, the Michael-Scott lock-free queue from his paper had a memory bug, so I was not able to implement it without causing memory leaks. I additionally grabbed an open-source queue implementation from GitHub, which is in `src/ds/queues/lib`.

Surprisingly, the production-grade thread pool had the same performance as my FastPool and FlexiblePool. So in the end, I decided to use my own implementation, as it is more flexible and easier to understand.

The choice of queue is the main bottleneck of the thread pool. However, as the graph size grows, the overhead of the thread queue becomes negligible. I decided to choose a blocking queue as it is more intuitive. In my testing, Head-Tail lock-based queue had the best performance among my implementations of blocking queues.

3 Benchmarking Results

3.1 Test cases

We will run the benchmark on a variety of graph types, using uniform edge weights and Power-law distributed edge weights in [0, 1). The graph types will be:

- Random graphs $G(n, m)$ with fixed number of edges $m = cn$ and edges chosen uniformly at random.
- RMAT-1 graphs $RMAT(n, m)$ with $m = cn$ generated using PaRMAT with parameters $A = 0.45$, $B = C = 0.22$ to produce highly skewed degree distributions

- Grid graphs $Grid(n_1, n_2)$ consisting of an $n_1 \times n_2$ grid where each vertex connects to immediate neighbors (left, right, above, below if they exist), with edges removed independently with probability $p = 0.1$

For delta-stepping, we will run each test case with delta values in 0.01, 0.1, 0.2, ..., 0.9. For parallel algorithms, we will run each test case with thread counts in 2, 4, 8, 10, 16.

The benchmark is done on a 24-core @ 5.4GHz machine with 48GB RAM.

3.2 Results

3.2.1 Random and RMAT graphs

Random and RMAT graphs represent the core test cases for evaluating parallel shortest path algorithms, as they capture the irregular connectivity patterns and scale-free properties commonly found in real-world networks. Our comprehensive evaluation across dense graphs (G(1e6,1e8)), sparse graphs (G(2e6,6e6), G(500k,5e6)), and RMAT graphs with varying parameters demonstrates the effectiveness of the lock-free approach.

The performance overview reveals dramatic differences between graph types and algorithms. Dense graphs with 100M edges show the most significant speedups, with the lock-free algorithm achieving up to 1.6× improvement over Dijkstra for uniform distributions and 1.3× for power-law distributions. The lock-based algorithm also performs well but consistently lags behind the lock-free approach. Sequential delta-stepping, while competitive on some graphs, generally underperforms due to its inability to exploit parallelism in the relaxation phases.

The aggregate threading performance shows that optimal thread counts vary by algorithm: the lock-free algorithm achieves peak performance at 8-10 threads, while the lock-based algorithm peaks earlier at 4-6 threads before synchronization overhead dominates. This fundamental difference highlights the superior scalability of the lock-free approach.

The delta value analysis reveals sophisticated performance relationships that depend critically on graph structure and edge weight distribution. Power-law distributed graphs exhibit higher sensitivity to delta selection, with performance variations of up to 2-3× between optimal and suboptimal values. This sensitivity stems from the irregular degree distribution creating uneven bucket fill patterns that require careful delta tuning to balance parallelism and synchronization overhead.

Uniform distribution graphs show more predictable delta relationships, typically achieving optimal performance at $\delta = 0.1 - 0.3$ for dense graphs and $\delta = 0.2 - 0.5$ for sparse graphs. The more balanced edge weight distribution creates more uniform bucket occupation, making the algorithm less sensitive to delta parameter selection.

RMAT graphs demonstrate intermediate behavior, with power-law RMAT graphs requiring higher delta values ($\delta = 0.3 - 0.7$) to achieve optimal performance, while uniform RMAT graphs perform well across a broader delta range. This reflects the interaction between topological structure (RMAT's scale-free properties) and edge weight distribution in determining optimal bucket granularity.

Algorithmic Performance Insights:

The lock-free algorithm consistently outperforms alternatives across all tested configurations, achieving speedups of 1.3-1.6× over Dijkstra on favorable graphs. Key performance drivers include:

- **Elimination of lock contention:** The lock-free design removes synchronization bottlenecks that plague the lock-based approach, particularly evident at higher thread counts.
- **Reduced cache coherence overhead:** Atomic operations on individual nodes create less memory traffic than coarse-grained locking strategies.
- **Optimal bucket utilization:** The relaxed bucket insertion strategy maintains high parallelism while preserving correctness guarantees.

The performance advantages are most pronounced on graphs with irregular degree distributions and high edge densities, where traditional approaches struggle with load imbalancing and synchronization overhead. Sequential delta-stepping, while theoretically optimal for single-threaded execution, fails to capitalize on the abundant parallelism available in large-scale graph processing tasks.

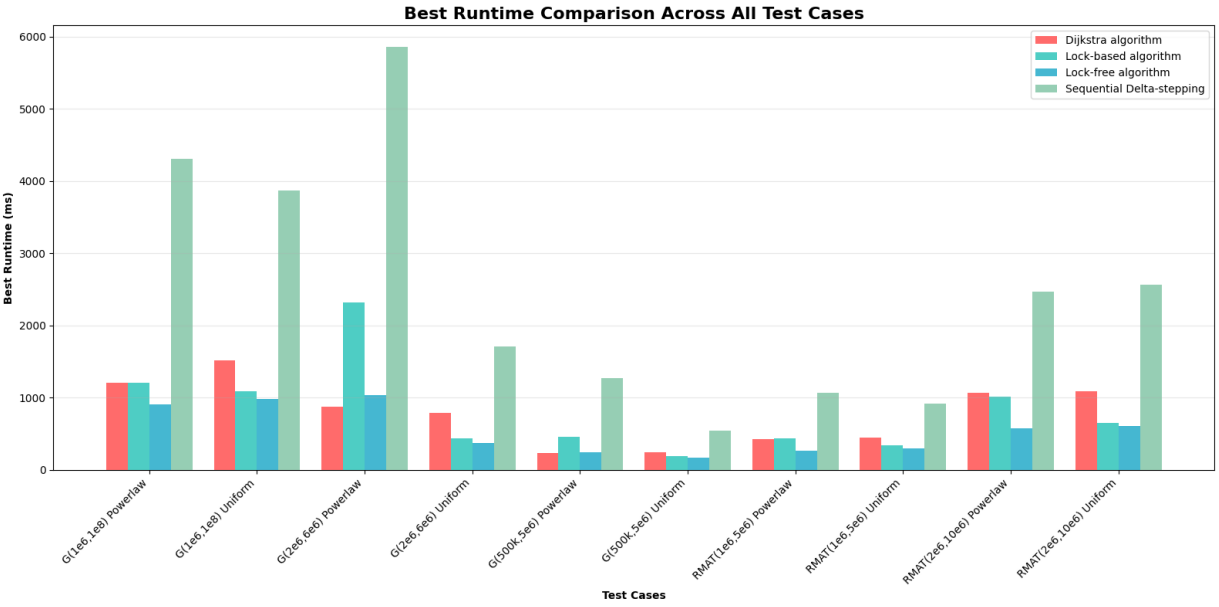

Figure 1: Best runtime comparison across all test cases

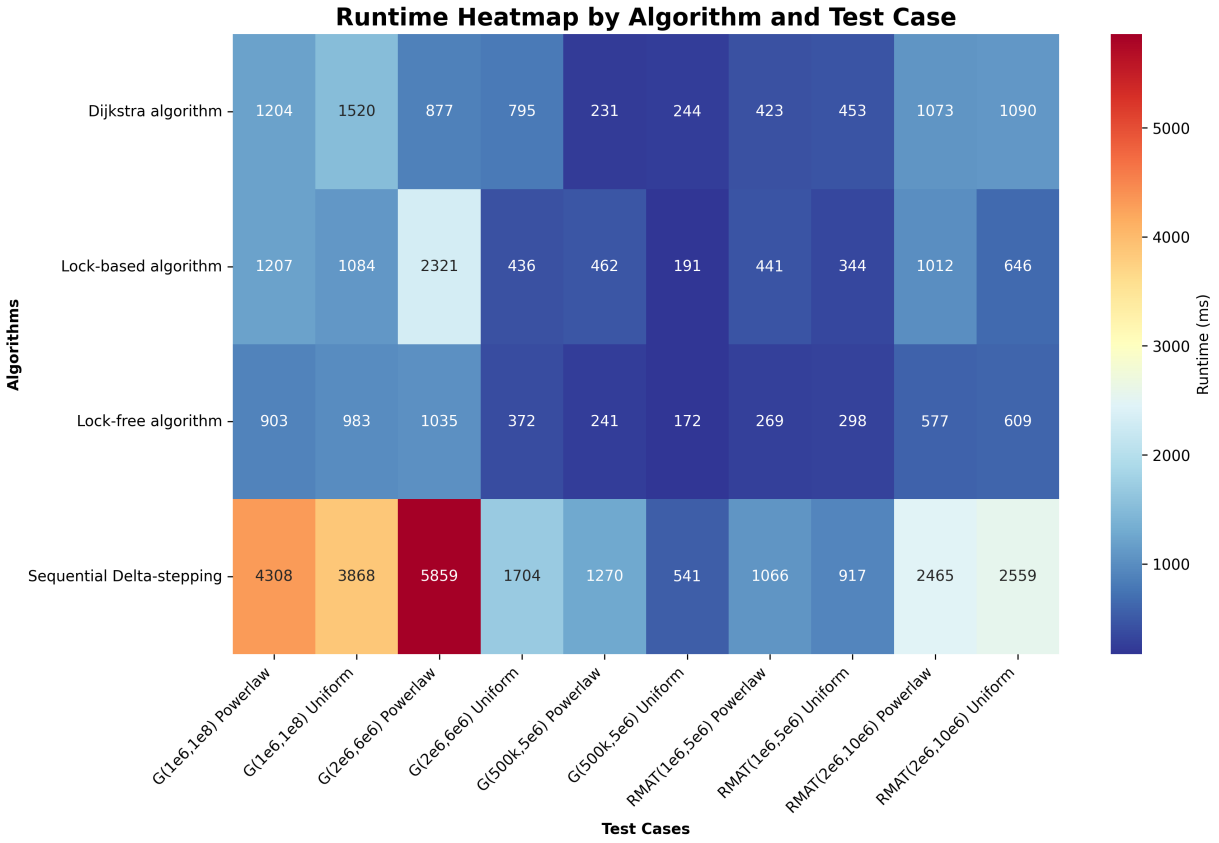

Figure 2: Runtime heatmap by algorithm and test case

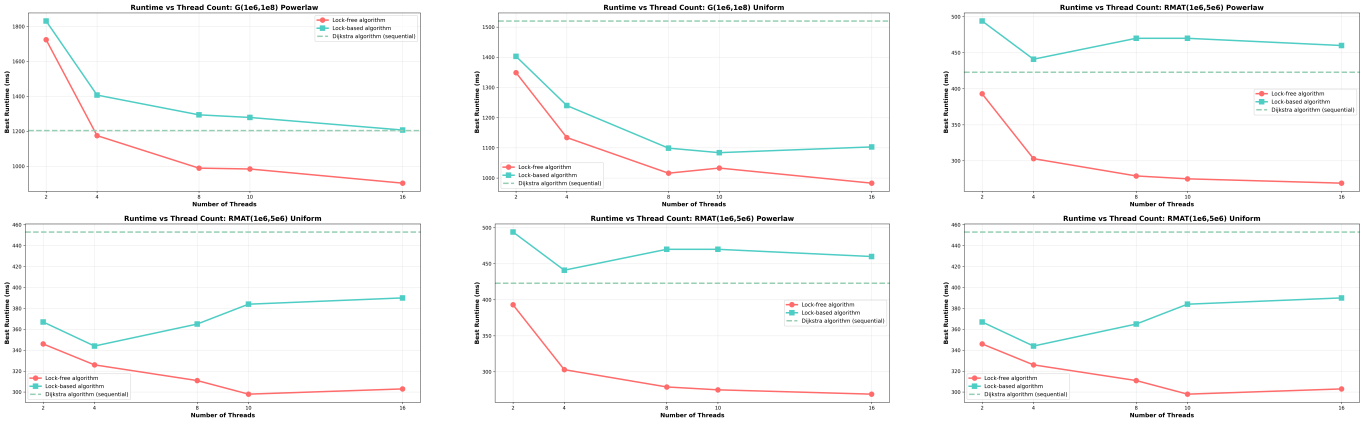

Figure 3: Runtime vs Thread Count across Graph Types and Distributions

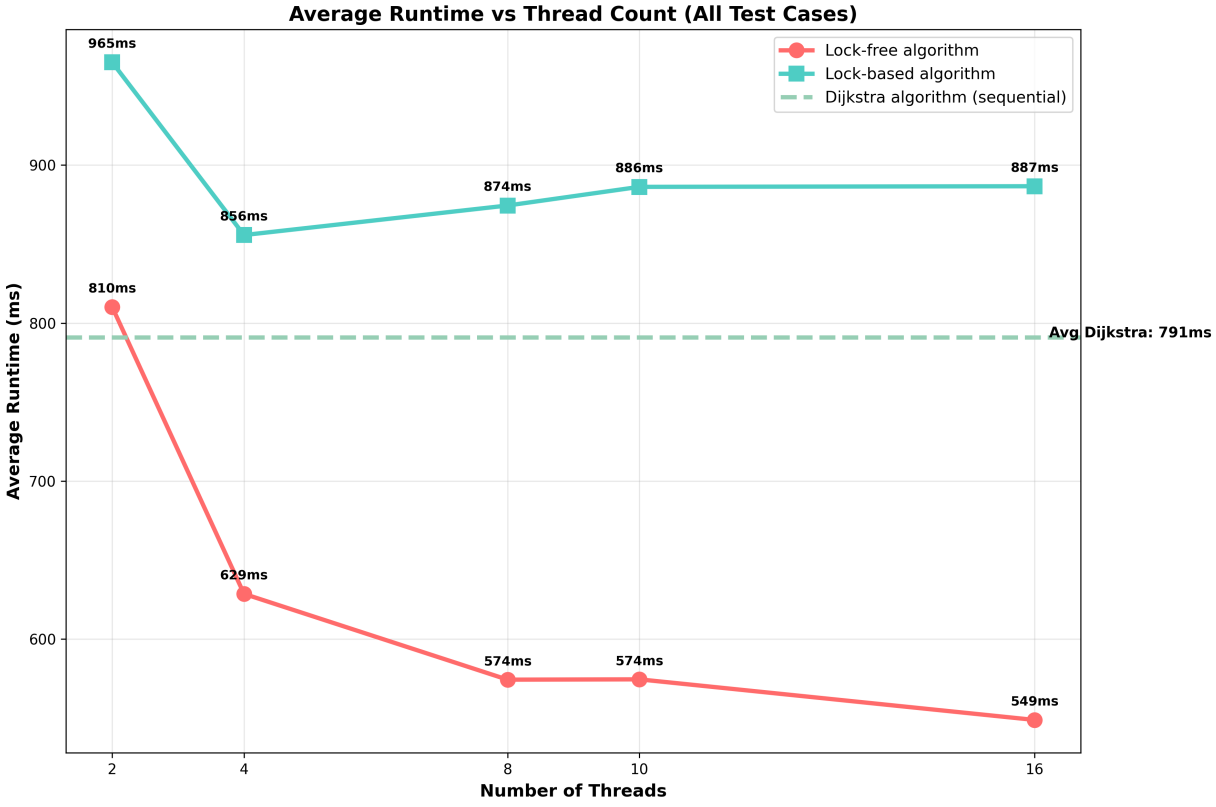

Figure 4: Average runtime vs thread count across all test cases

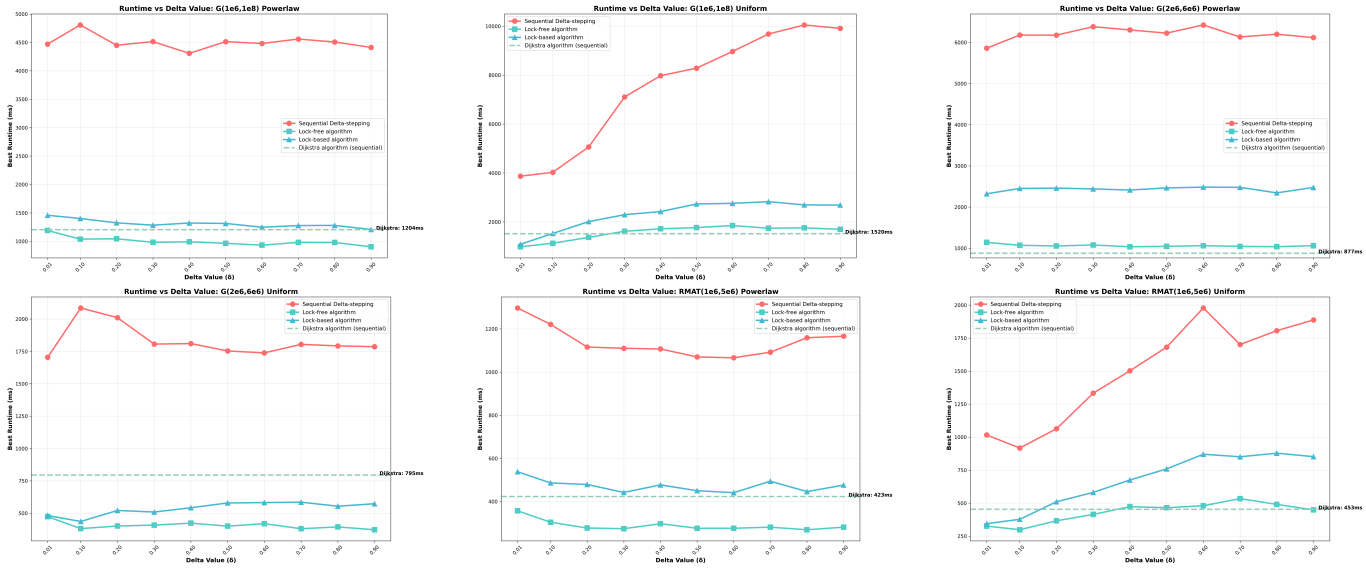

Figure 5: Runtime vs Delta Value across Graph Types and Distributions

3.2.2 Grid graphs

Grid graphs present unique challenges for parallel delta-stepping algorithms due to their structural properties. Our analysis of a Grid(2k×2k) uniform graph with 4 million vertices and 14.4 million edges reveals several key insights about algorithm performance on grid topologies.

The threading analysis reveals that grid graphs exhibit fundamentally different scaling behavior compared to random and RMAT graphs. While parallel delta-stepping provides competitive performance compared to Dijkstra's algorithm for grid graphs, the benefits are more modest than with other graph types. This occurs because grid graph topology results in fewer vertices/requests to process at each bucket level, reducing available parallelism while increasing the number of synchronization epochs required.

The delta value analysis shows that higher delta values ($\delta = 0.7 - 0.9$) are essential for achieving competitive performance on grid graphs. For very low delta values ($\delta = 0.01$), the algorithm performance degrades significantly, which is why this value was excluded from our benchmark. The lock-free algorithm achieves a modest 1.10× speedup over Dijkstra with optimal configuration ($\delta = 0.9$, 2 threads), demonstrating that careful parameter tuning can overcome the structural limitations of grid topologies.

Notably, increasing thread count beyond 2 threads generally degrades performance on grid graphs, confirming that the overhead of parallelization overwhelms the benefits when work distribution becomes too fine-grained for this graph structure.

4 How to use the code & Acknowledgements

The project directory is separated into several parts:

- `src/algo`: Contains the implementation of the algorithms
- `src/ds`: Contains the implementation of the data structures
- `src/tests`: Contains the implementation of the tests
- `src/core`: Contains the base interfaces for the algorithms and data structures

Generative AI was used strictly to generate the code to test correctness, run benchmarks and graph results.

When grading the projects, you should be interested in the implementations in `src/algo` and `src/ds`. Files in `*/lib` folders are open-source implementations taken from

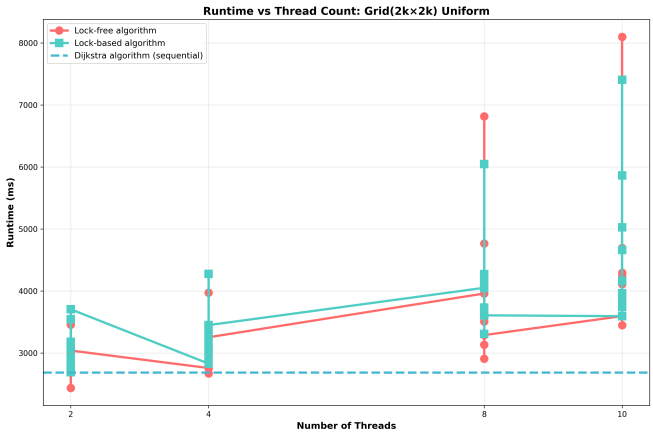

Figure 6: Runtime vs thread count for grid graphs

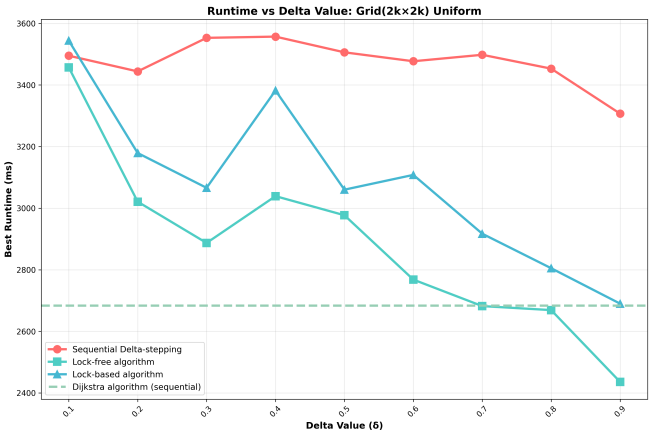

Figure 7: Runtime vs delta value for grid graphs

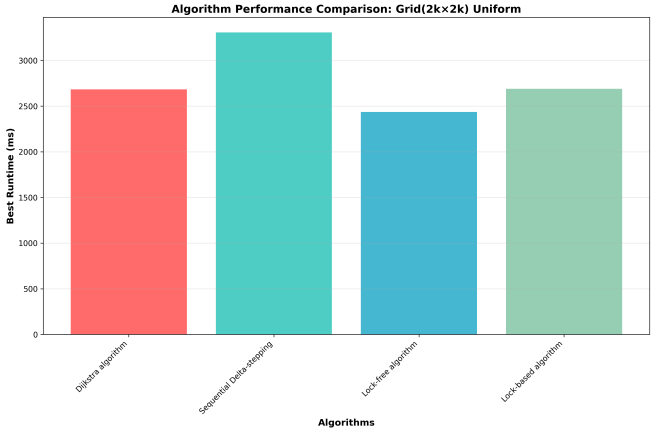

Figure 8: Algorithm performance comparison on grid graphs

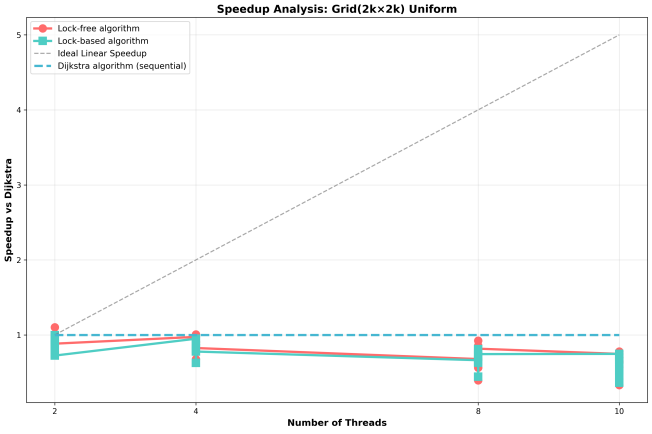

Figure 9: Speedup analysis for grid graphs

GitHub. The rest are my own implementations. `src/algo/delta_stepping_parallel.h` contains the lock-free implementation, while `src/algo/dsp_recycle_bucket.h` contains the lock-based implementation.

When checking the commit history, you should look for commits with prefix `[ALGO]` or `[DS]`. `[ALGO]` commits are the ones that contain the implementation of the algorithms, while `[DS]` commits are the ones that contain the implementation of the data structures.

5 Conclusion

Our implementation demonstrates that the lock-free parallel delta stepping algorithm can achieve significant speedup over sequential implementations while maintaining correctness. The careful design of thread pools and synchronization mechanisms plays a crucial role in performance.

6 Future Work

- GPU implementation
- Dynamic delta value optimization
- Advanced thread pool strategies
- Lock-free thread pool
- Shortcuts