Lock-free Parallel Delta-stepping Algorithm

Dijkstra's Algorithm

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with minimum dist[u]
        Q.remove(u)

        for each arc (u, v) in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

Runtime:
Optimal case: O(|E| + |V| log |V|) using Fibonacci heap.

BUT: Fibonacci heap is a theoretical data structure, which is inapplicable to real-world scenario.

So, we implemented Dijkstra's using **Binary heap (std::priority_queue)**

Dijkstra’s Algorithm

```cpp
std::vector<double> compute(const Graph &graph, int source) const override {
    std::priority_queue<std::pair<double, int>> pq;
    int n = graph.size();
    std::vector<double> dist(n, std::numeric_limits<double>::infinity());
    std::vector<bool> vis(n, false);
    dist[source] = 0;
    pq.push({0, source});
    while (!pq.empty()) {
        auto u = pq.top().second;
        pq.pop();
        if (vis[u]) continue;
        vis[u] = true;
        for (const auto &[v, w] : graph[u]) {
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                pq.push({-dist[v], v});
            }
        }
    }
    return dist;
}
```

Delta-stepping Algorithm

Algorithm 1 Parallel Δ-stepping SSSP

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

1: **for** $v \in V \setminus \{s\}$ **do**
2: $N_v^l \leftarrow \{(u, w) \in E \mid u = v \wedge w < \delta\}$ ▷ Light edges
3: $N_v^h \leftarrow \{(u, w) \in E \mid u = v \wedge w \geq \delta\}$ ▷ Heavy edges
4: $d^*(v) \leftarrow \infty$
5: $B_\infty \leftarrow B_\infty \cup \{v\}$
6: **end for**
7: $B_\infty \leftarrow B_\infty \setminus \{s\}$
8: $B_0 \leftarrow B_0 \cup \{s\}$
9: $d^*(s) \leftarrow 0$
10: **while** $\exists i < \infty : B_i \neq \emptyset$ **do**
11: $k \leftarrow \min\{i : B_i \neq \emptyset\}$
12: $S \leftarrow B_k$
13: **while** $S \neq \emptyset$ **do**
14: **for** $v \in S$ **in parallel do**
15: $B_k \leftarrow B_k \setminus \{v\}$
16: $R_l \leftarrow R_l \cup \{(v, w) \mid (v, w) \in N_v^l\}$
17: $R_h \leftarrow R_h \cup \{(v, w) \mid (v, w) \in N_v^h\}$
18: **end for**
19: **for** $(v, w) \in R_l$ **in parallel do**
20: **if** $d^*(v) + w < d^*(w)$ **then**
21: $i \leftarrow \lfloor d^*(w)/\delta \rfloor$
22: $j \leftarrow \lfloor (d^*(v) + w)/\delta \rfloor$
23: $B_i \leftarrow B_i \setminus \{w\}$
24: $B_j \leftarrow B_j \cup \{w\}$
25: $d^*(w) \leftarrow d^*(v) + w$
26: **end if**
27: **end for**
28: $S \leftarrow B_k$
29: **end while**
30: **for** $(v, w) \in R_h$ **in parallel do**
31: **if** $d^*(v) + w < d^*(w)$ **then**
32: $i \leftarrow \lfloor d^*(w)/\delta \rfloor$
33: $j \leftarrow \lfloor (d^*(v) + w)/\delta \rfloor$
34: $B_i \leftarrow B_i \setminus \{w\}$
35: $B_j \leftarrow B_j \cup \{w\}$
36: $d^*(w) \leftarrow d^*(v) + w$
37: **end if**
38: **end for**
39: $R_l \leftarrow \emptyset$
40: $R_h \leftarrow \emptyset$
41: **end while**
42: **return** d^*

"Dynamic" implementation: each thread's workload is adapted to the total workload (i.e. divided equally), instead of always fixed.

→ Needs proper synchronization, as multiple threads might be writing to the same address at once.

→ The original author suggested using locks to ensure thread-safety.

Why go Lock-free?

- The dynamic implementation, while is better for load-balancing, thus making the most out of parallelism.
- But in the original paper, it still lagged behind the static approach (poor load-balancing, but no locks needed).
- Locks require use of system calls, which are expensive.

→ A lock-free dynamic implementation incorporates the best of both worlds.

→ We will see the improvement going lock-free adds later.

Re-using buckets

Algorithm 1 Parallel Δ-stepping SSSP

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

1: **for** $v \in V \setminus \{s\}$ **do**
2: $N_v^l \leftarrow \{(u, w) \in E \mid u = v \wedge w < \delta\}$ ▷ Light edges
3: $N_v^h \leftarrow \{(u, w) \in E \mid u = v \wedge w \geq \delta\}$ ▷ Heavy edges
4: $d^*(v) \leftarrow \infty$
5: $B_\infty \leftarrow B_\infty \cup \{v\}$
6: **end for**
7: $B_\infty \leftarrow B_\infty \setminus \{s\}$
8: $B_0 \leftarrow B_0 \cup \{s\}$
9: $d^*(s) \leftarrow 0$
10: **while** $\exists i < \infty : B_i \neq \emptyset$ **do**
11: $k \leftarrow \min\{i : B_i \neq \emptyset\}$
12: $S \leftarrow B_k$
13: **while** $S \neq \emptyset$ **do**
14: **for** $v \in S$ **in parallel do**
15: $B_k \leftarrow B_k \setminus \{v\}$
16: $R_l \leftarrow R_l \cup \{(v, w) \mid (v, w) \in N_v^l\}$
17: $R_h \leftarrow R_h \cup \{(v, w) \mid (v, w) \in N_v^h\}$
18: **end for**
19: **for** $(v, w) \in R_l$ **in parallel do**
20: **if** $d^*(v) + w < d^*(w)$ **then**
21: $i \leftarrow \lfloor d^*(w)/\delta \rfloor$

Let c* be the maximum weight of all edges in the graph.

At any point during the relaxation phase, only buckets that are at most c* / delta distance away will be impacted.

→ Use a fixed array for buckets, and use modulo to wrap around every time the bound is reached.

```
int generations_without_bucket = 0;
for (current_generation = 0; ; ++current_generation, ++generations_without_bucket) {
    if (generations_without_bucket >= MAX_BUCKET_COUNT) {
        break;
    }
    if (current_generation >= MAX_BUCKET_COUNT) {
        current_generation = 0;
    }
    while (!buckets[current_generation].empty()) {
        generations_without_bucket = 0;

        {
```

Request generation

Algorithm 1 Parallel Δ-stepping SSSP

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

1: **for** $v \in V \setminus \{s\}$ **do**
2: $N_v^l \leftarrow \{(u, w) \in E \mid u = v \wedge w < \delta\}$ ▷ Light edges
3: $N_v^h \leftarrow \{(u, w) \in E \mid u = v \wedge w \geq \delta\}$ ▷ Heavy edges
4: $d^*(v) \leftarrow \infty$
5: $B_\infty \leftarrow B_\infty \cup \{v\}$
6: **end for**
7: $B_\infty \leftarrow B_\infty \setminus \{s\}$
8: $B_0 \leftarrow B_0 \cup \{s\}$
9: $d^*(s) \leftarrow 0$
10: **while** $\exists i < \infty : B_i \neq \emptyset$ **do**
11: $k \leftarrow \min\{i : B_i \neq \emptyset\}$
12: $S \leftarrow B_k$
13: **while** $S \neq \emptyset$ **do**
14: **for** $v \in S$ **in parallel do**
15: $B_k \leftarrow B_k \setminus \{v\}$
16: $R_l \leftarrow R_l \cup \{(v, w) \mid (v, w) \in N_v^l\}$
17: $R_h \leftarrow R_h \cup \{(v, w) \mid (v, w) \in N_v^h\}$
18: **end for**
19: **for** $(v, w) \in R_l$ **in parallel do**
20: **if** $d^*(v) + w < d^*(w)$ **then**
21: $i \leftarrow \lfloor d^*(w)/\delta \rfloor$

The original author proposed “strictest request” with per-node locking.

Notice that we only need a floating point value to represent a request.

→ Use std::atomic<double> for lock-free request generation.

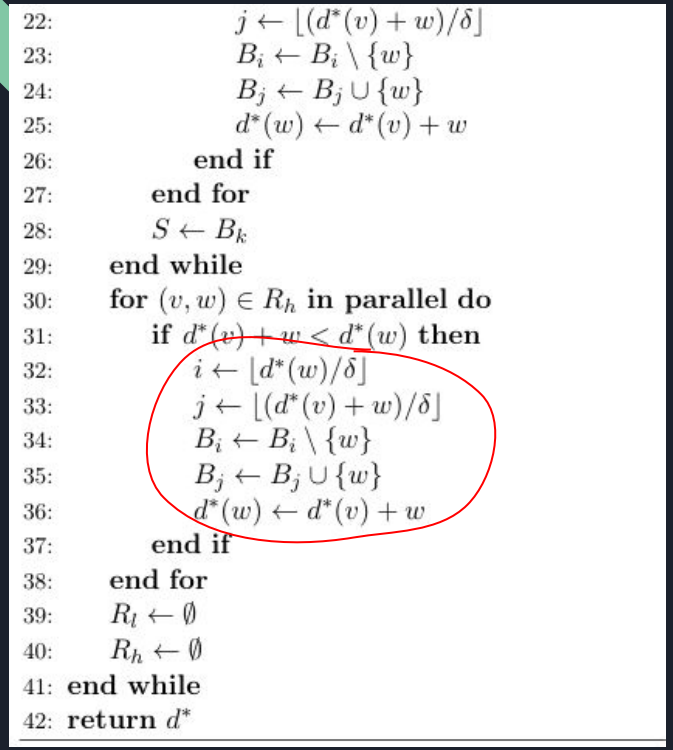
Request generation

```cpp
auto gen_light_request = [&] (int u) {
    for (const auto &[v, w] : light[u]) {
        if (dist[u] + w < dist[v]) {
            add_request(light_nodes_requested, light_nodes_counter, light_request_map, Request{u, v, w});
        }
    }
};
```

```cpp
auto gen_heavy_request = [&] (int u) {
    for (const auto &[v, w] : heavy[u]) {
        if (dist[u] + w < dist[v]) {
            add_request(heavy_nodes_requested, heavy_nodes_counter, heavy_request_map, Request{u, v, w});
        }
    }
};
```

```cpp
// Strictest request optimization -- No mutexes
auto add_request = [&] (std::vector<int> &requested_nodes, std::atomic<size_t> &idx_counter, std::vector<std::atomic<double>> &requests, const Request &request) {
    std::atomic<double> &state = requests[request.v];
    double new_distance = dist[request.u] + request.w;

    if (std::isinf(state.load())) {
        double curr_state = state.load();
        while (std::isinf(curr_state) && !state.compare_exchange_weak(curr_state, new_distance));
        if (std::isinf(curr_state)) {
            size_t curr_idx = idx_counter.fetch_add(1);
            requested_nodes[curr_idx] = request.v;
        }
    }

    double current_distance = state.load();
    while (new_distance < current_distance && !state.compare_exchange_weak(current_distance, new_distance));
};
```

Bucket insertion/removal



```
22:                 j ← ⌊(d*(v) + w)/δ⌋
23:                 B_i ← B_i \ {w}
24:                 B_j ← B_j ∪ {w}
25:                 d*(w) ← d*(v) + w
26:             end if
27:         end for
28:         S ← B_k
29:     end while
30:     for (v, w) ∈ R_h in parallel do
31:         if d*(v) + w < d*(w) then
32:             i ← ⌊d*(w)/δ⌋
33:             j ← ⌊(d*(v) + w)/δ⌋
34:             B_i ← B_i \ {w}
35:             B_j ← B_j ∪ {w}
36:             d*(w) ← d*(v) + w
37:         end if
38:     end for
39:     R_l ← ∅
40:     R_h ← ∅
41: end while
42: return d*
```

During the relaxation phase, values might be added/removed from buckets.

→ Each bucket needs thread-safe insertion/removal of values.

Key observation: It is possible to make each bucket contain no more than |V| elements throughout its entire life time. This is each vertex can only be pushed into a bucket once, and removed from that bucket once.

→ Solution: Use fixed size array for buckets, and an atomic tail counter. Each time the bucket is reset, simply set tail to 0.

Bucket insertion/removal

- Node removed from bucket → set value to -1.
- Node inserted to bucket → insert to the tail.
- “Harmful” fragmentation only caused by moving a node to the same bucket → can be prevented.
- Thus the correctness.

```
auto relax = [&] (int v, std::vector<std::atomic<double>> &requests) {
    double new_distance = requests[v].exchange(std::numeric_limits<double>::infinity());
    if (new_distance < dist[v]) {
        int old_bucket = get_bucket(v);
        dist[v] = new_distance;
        int new_bucket = get_bucket(v);
        if (old_bucket != -1 && old_bucket != current_generation && old_bucket != new_bucket) { // since current generation bucket is always cleared
            buckets[old_bucket][position_in_bucket[v]] = -1;
        }
        if (old_bucket == current_generation || old_bucket != new_bucket) {
            position_in_bucket[v] = buckets[new_bucket].push(v);
        }
    }
};
```

Thread pooling

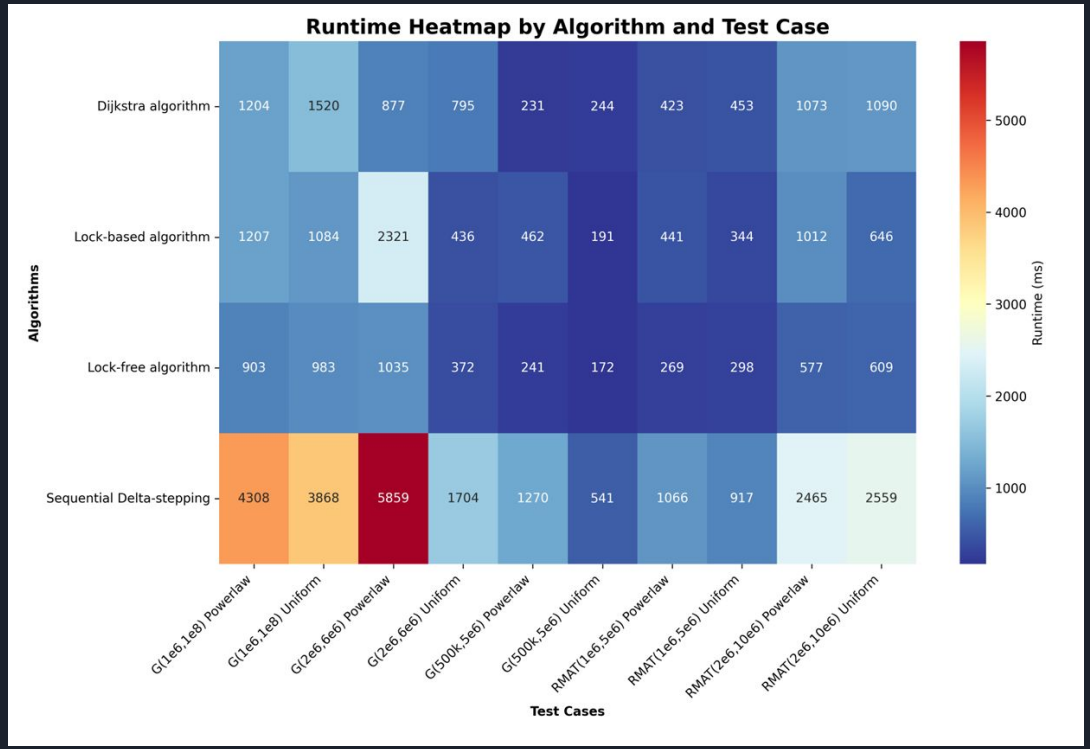
⚠️ Similar to locks, thread creation requires system calls → expensive.

→ Solution: Use a thread-pool to keep threads alive.

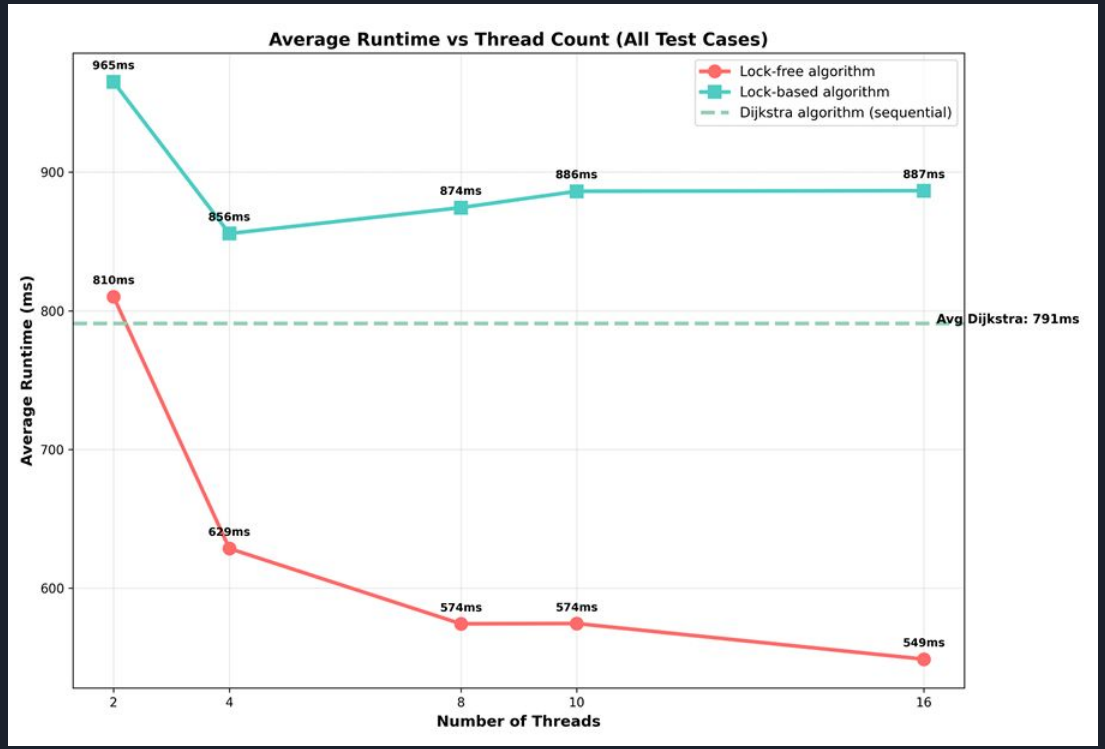
→ Needs a thread-safe concurrent queue (SPMC).

→ However, with the specifics of delta-stepping algorithm, it is sufficient to have a fixed task pool, i.e. use a fixed size array to store the tasks, as we can make sure the number of tasks at any time could not exceed num_threads.

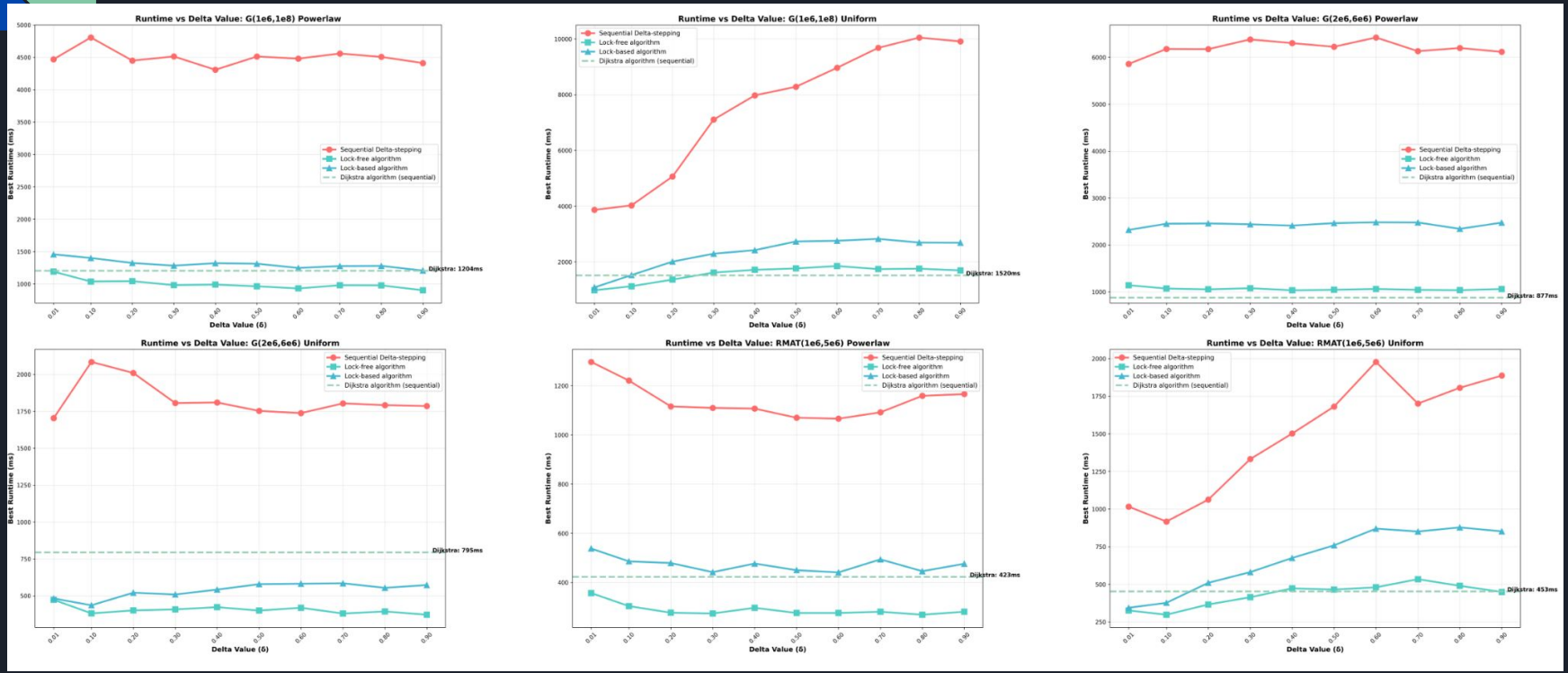
Results: Best runtime



Results: Thread count vs. Run time



Results: Delta value vs Run time



Even better: Load-balancing by edges

- Our previous implementations load balance the vertices in a bucket.
- But what actually impacted the runtime was the total number of requests these vertices generate.

→ Would be sick to load balance with respect to the number of edges instead of vertices.

⚠️This is non-trivial, and makes the code much more complicated

✅Implemented using prefix sum, and binary search.

→ Two versions: one that calculates the prefix sum sequentially (more cache-friendly), one that calculates the prefix sum in parallel (better parallelism, less cache-friendly).

Calculating the prefix sum sequentially

```
while (!buckets[current_generation].empty()) {
    generations_without_bucket = 0;

    {
        // Loop 1: request generation
        CircularVector<int> &curr_bucket = buckets[current_generation];
        size_t curr_bucket_size = curr_bucket.size();

        prefix[0] = 0;
        for (size_t i = 0; i < curr_bucket_size; ++i) {
            int u = curr_bucket[i];
            if (i > 0) {
                prefix[i] = prefix[i - 1];
            }
            if (u >= 0) {
                prefix[i] += adj_sizes[u];
            }
        }
        size_t total_edges = prefix[curr_bucket_size - 1];

        // prefix ready - no need for extra barrier here
        // (D) Even split of edges across threads using the global prefix
        const size_t edge_chunk = (total_edges + num_threads - 1) / num_threads;
```

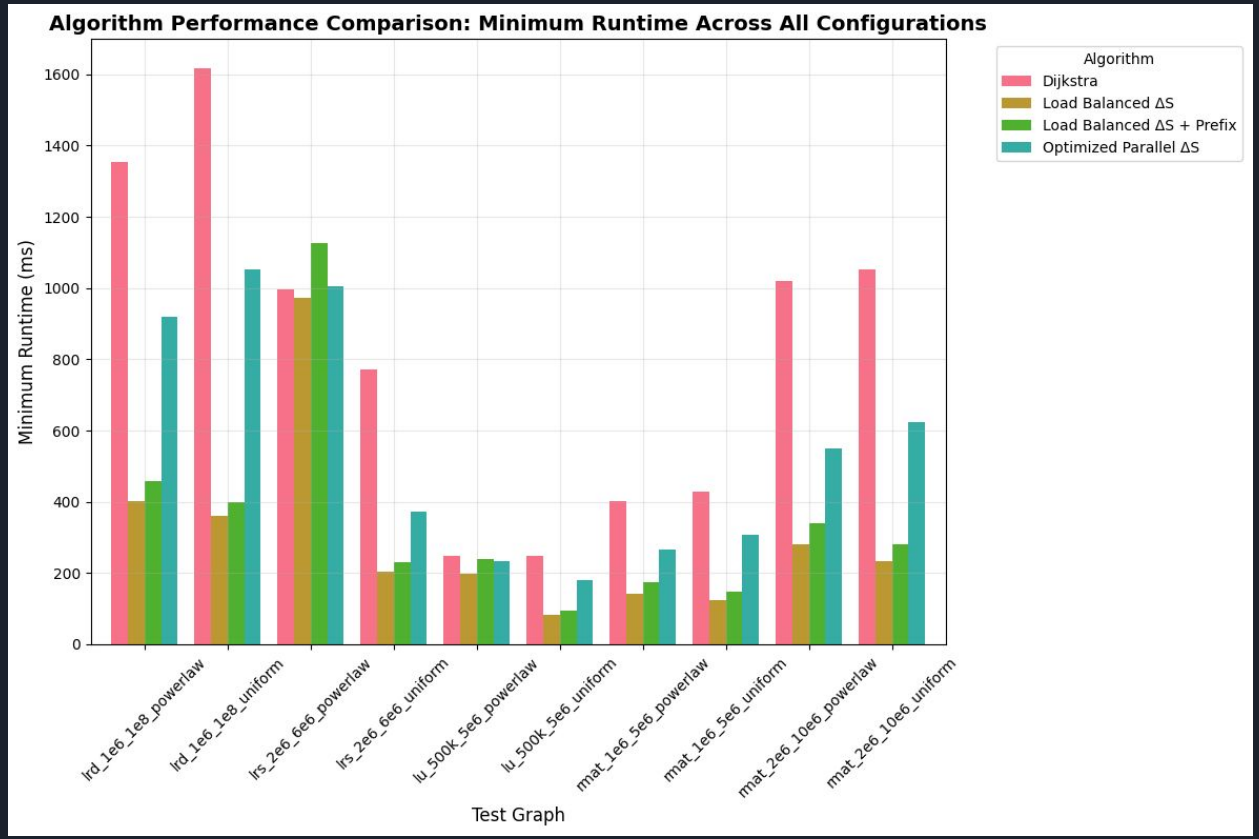
```
for (int tid = 0; tid < num_threads; ++tid) {
    size_t start_e = static_cast<size_t>(tid) * edge_chunk;
    size_t end_e   = std::min(total_edges, start_e + edge_chunk);

    pool.push(tid, [&, start_e, end_e] {
        if (start_e >= end_e) {
            return;
        }

        size_t node_idx = std::upper_bound(prefix.begin(), prefix.begin() + curr_bucket_size, start_e) - prefix.begin();
        size_t edge_off = start_e;
        if (node_idx > 0) edge_off -= prefix[node_idx - 1];
        size_t curr_edge = start_e;

        while (curr_edge < end_e && node_idx < curr_bucket_size) {
            int u = curr_bucket[node_idx];
            if (u >= 0) {
                size_t deg = adj_sizes[u];
                for (size_t k = edge_off; k < deg && curr_edge < end_e; ++k, ++curr_edge) {
                    const auto &[v, w] = graph[u][k];
                    if (dist[u] + w < dist[v]) {
                        if (w < delta) {
                            add_request(light_nodes_requested, light_nodes_counter, light_request_map, Request{u, v, w});
                        }
                        else {
                            add_request(heavy_nodes_requested, heavy_nodes_counter, heavy_request_map, Request({u, v, w}));
                        }
                    }
                }
            }
            ++node_idx;
            edge_off = 0;
        }
    });
}
```

Calculating the prefix sum in parallel

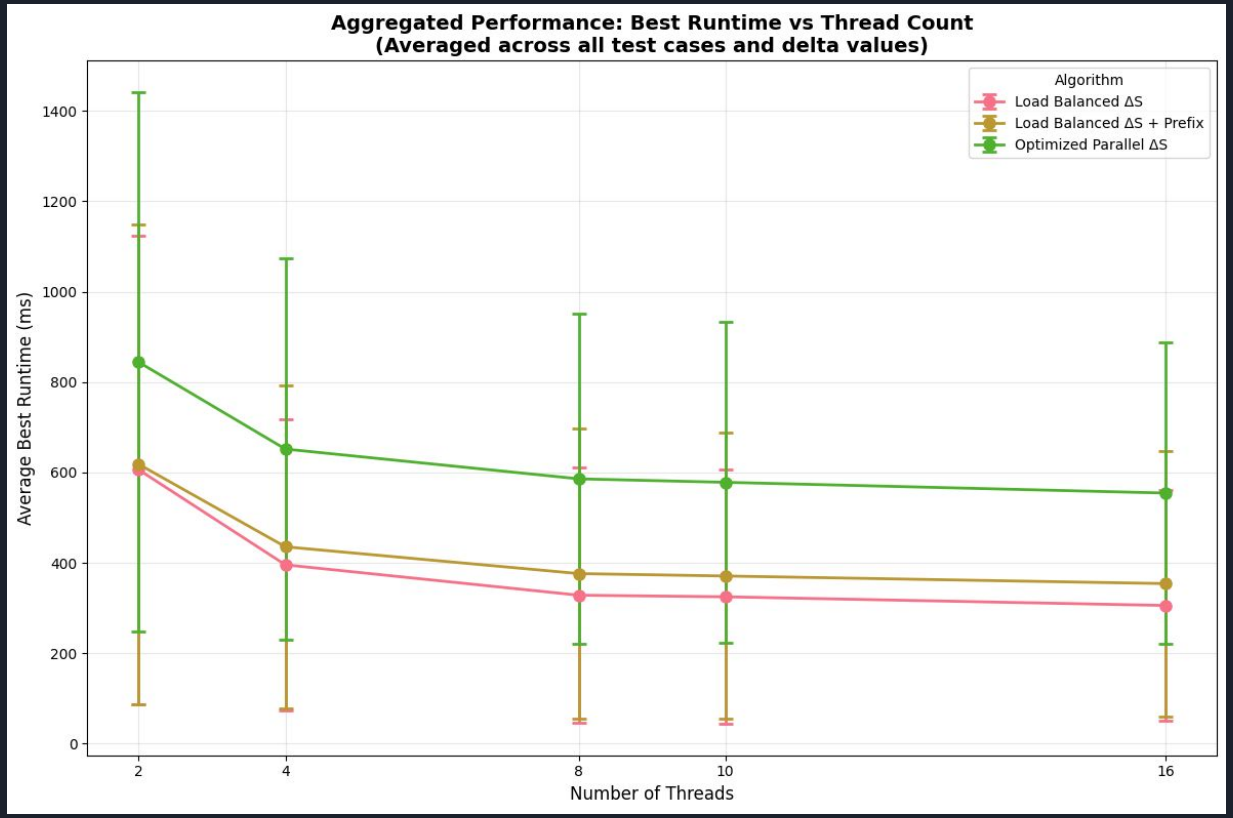
```
while (!buckets[current_generation].empty()) {
    generations_without_bucket = 0;

    {
        // Loop 1: request generation
        CircularVector<int> &curr_bucket = buckets[current_generation];
        size_t curr_bucket_size = curr_bucket.size();

        size_t nodes_per_thread = (curr_bucket_size + num_threads - 1) / num_threads;

        // (A) each thread fills prefix for its slice + counts edges
        for (size_t tid = 0; tid < num_threads; ++tid) {
            int l = tid * nodes_per_thread;
            int r = std::min(curr_bucket_size, l + nodes_per_thread);
            pool.push(tid, [&curr_bucket, &adj_sizes, &prefix, &thread_totals, tid, l, r] {
                size_t running = 0;
                for (int i = l; i < r; ++i) {
                    int u = curr_bucket[i];
                    if (u >= 0) {
                        running += adj_sizes[u];
                    }
                    prefix[i] = running;
                }
                thread_totals[tid] = running;
            });
        }
        barrier.arrive_and_wait();

        // (B) master thread computes exclusive scan of thread_totals
        thread_pref[0] = 0;
        for (size_t tid = 0; tid < num_threads; ++tid) {
            if (tid > 0) {
                thread_pref[tid] = thread_pref[tid - 1];
            }
            thread_pref[tid] += thread_totals[tid];
        }

        size_t total_edges = thread_pref[num_threads - 1];
```

```
size_t curr_ptr = 0; // idx of current node batch

for (size_t tid = 0; tid < num_threads; ++tid) {
    size_t start_e = static_cast<size_t>(tid) * edge_chunk;
    size_t end_e   = std::min(total_edges, start_e + edge_chunk);
    while (curr_ptr < num_threads && start_e >= thread_pref[curr_ptr]) {
        ++curr_ptr;
    }
    size_t start_e_batch = start_e;
    if (curr_ptr > 0) {
        start_e_batch -= thread_pref[curr_ptr - 1];
    }

    pool.push(tid, [&, start_e, end_e, start_e_batch, curr_ptr] {
        if (start_e >= end_e) {
            return;
        }

        size_t node_idx = std::upper_bound(prefix.begin() + curr_ptr * nodes_per_thread,
                                           prefix.begin() + std::min((curr_ptr + 1) * nodes_per_thread,
                                           curr_bucket_size),
                                           start_e_batch) - prefix.begin();      You, 1 second ago • Uncommitted changes
        size_t edge_off = start_e_batch;
        if (node_idx > curr_ptr * nodes_per_thread) edge_off -= prefix[node_idx - 1];
        size_t curr_edge = start_e;

        while (curr_edge < end_e && node_idx < curr_bucket_size) {
            int u = curr_bucket[node_idx];
            if (u >= 0) {
                size_t deg = adj_sizes[u];
                for (size_t k = edge_off; k < deg && curr_edge < end_e; ++k, ++curr_edge) {
                    const auto &[v, w] = graph[u][k];
                    if (dist[u] + w < dist[v]) {
                        if (w < delta) {
                            add_request(light_nodes_requested, light_nodes_counter, light_request_map, Request{u, v, w});
                        }
                        else {
                            add_request(heavy_nodes_requested, heavy_nodes_counter, heavy_request_map, Request({u, v, w}));
                        }
                    }
                }
            }
            ++node_idx;
            edge_off = 0;
        }
```

Results: Best runtime



Results: Thread count vs. Run time



Demo