École Polytechnique

IP Paris

Final Project Report - CSE305

Parallel shortest paths

Author:
Adan FHIMA, Dan SUISSA

Supervised by:
Gleb Pogudin Eric Goubault

Academic year 2024/2025

Contents

1 Introduction and Background

The shortest path problem constitutes a fundamental challenge in graph theory with applications spanning GPS navigation, network routing, and social network analysis. Given a weighted graph, the single-source shortest path (SSSP) problem seeks minimum-weight paths from a designated source vertex to all reachable vertices.

Classical algorithms have long dominated this domain. Dijkstra's algorithm (1959) achieves optimal $O((V+E)\log V)$ performance for non-negative weights through greedy vertex selection, but its sequential nature limits parallel scalability. The Bellman-Ford algorithm (1958) offers better parallelization potential through iterative edge relaxation, though at higher $O(V \times E)$ computational cost.

Meyer and Sanders introduced delta-stepping in 2003 to bridge this gap, providing a parallelizable approach that maintains efficiency comparable to Dijkstra's algorithm. The algorithm partitions vertices into distance-based buckets, enabling concurrent processing while preserving correctness guarantees.

1.1 Delta-Stepping Algorithm Principles

The *delta-Stepping* algorithm solves the single-source shortest-path problem by partitioning the distance space into fixed-width intervals (*buckets*) so that many vertices can be processed in parallel.

1. **Initialisation**
   Set $d[s] = 0$ for the source s and $d[v] = \infty$ for every other vertex v. Create buckets $B_0, B_1, \ldots$ and place s in B_0.

2. **Edge classification**
   Fix a parameter $\Delta > 0$. An edge $e = (u, v, w)$ is *light* if $w \leq \Delta$ and *heavy* otherwise.

3. **Main loop**
   Let `idx` be the index of the smallest non-empty bucket.

   (a) **Repeated light-edge scans**
   While $B_{\texttt{idx}} \neq \varnothing$, extract its current content C; relax every light edge out of C in parallel; place any vertex whose tentative distance decreases back into the bucket determined by its new distance (still within $[\texttt{idx}\,\Delta, (\texttt{idx}+1)\Delta)$).

   (b) **Single heavy-edge sweep**
   Once $B_{\texttt{idx}}$ stays empty after a light scan, relax in parallel every heavy edge whose tail lies in C; insert updated targets into their proper buckets.

   (c) **Advance pointer**
   Increase `idx` until a non-empty bucket is found, or halt if none remain.

4. **Termination**
   When all buckets are empty, each $d[v]$ equals the true shortest-path distance from s to v.

Choosing Δ. A small Δ yields narrow buckets with minimal redundant work but limited parallelism; a large Δ widens buckets, increasing concurrency at the cost of extra relaxations. Practical implementations tune Δ empirically or set it near the mean weight of candidate light edges.

Edges are classified as either *light* (weight $\leq \Delta$) or *heavy* (weight $> \Delta$). Light edges are processed repeatedly within bucket phases, while heavy edges are processed once per bucket to ensure correctness.

The parameter Δ critically influences performance: small values reduce parallelism but maintain precision, while large values increase parallelism but may cause redundant work. Optimal Δ selection depends on graph structure and computational resources.

2 Implementation Details

We implemented four distinct algorithms: Dijkstra's algorithm as baseline, sequential delta-stepping, and two parallel delta-stepping variants with different optimization strategies.

Sequential Dijkstra Implementation

The Dijkstra implementation employs a binary min-heap priority queue for vertex selection, maintaining optimal $O((V+E)\log V)$ complexity and serving as our performance baseline.

Sequential Delta-Stepping Implementation

Our sequential delta-stepping follows the original Meyer-Sanders design [1] with bucket-based vertex organization. The implementation uses dynamic bucket allocation and processes light edges iteratively within buckets before processing heavy edges once per bucket completion.

Parallel Delta-Stepping v1

Our first parallel prototype relied on simple concurrency primitives: a single mutex protected the entire bucket structure, each bucket was kept in a thread-safe deque, and all relaxation requests were funnelled through a central queue. In practice this design imposed a severe bottleneck. The global lock became a hot spot that often left the "parallel" code running slower than its sequential baseline, and thread-management overhead only amplified the problem on graphs with limited inherent concurrency.

Parallel Delta-Stepping v2

In V1, every time a vertex's distance improved, all threads competed for a single lock to remove and reinsert vertices into buckets, and each relaxation step required scanning entire buckets under that lock.
V2 eliminates the big lock by using atomic distance updates and one mutex per bucket. When a thread finds a shorter path to v, it atomically updates distances[v] and then locks only the new bucket to append v. There is no need to remove v from its old bucket explicitly, because any outdated update simply fails the compare-and-swap. Also Light and heavy-edge relaxations are handled in two phases: each thread collects candidate updates in a array, which are merged and filtered to keep only the best update per vertex, and only then are atomic writes and bucket insertions performed.

3 Experimental Setup

All experiments were conducted on an Apple M1 Pro system with 8-core CPU and 16 GB LPDDR5 RAM, using C++17 with g++ optimization.

3.1 Benchmark Datasets

Synthetic Graphs

We generated diverse synthetic graphs to test algorithm behavior across different structural patterns:

Random graphs with uniform edge placement, grid graphs representing regular 2D lattices that simulate spatial networks, scale-free graphs with power-law degree distributions modeling real-world networks, and small-world graphs characterized by high clustering and short paths. Graph sizes ranged from 100 to 5,000 vertices with varying densities (0.5% to 20%) and different weight distributions (uniform, exponential, normal, bimodal).

Real-World Graphs

Five collaboration networks from the SNAP dataset collection [3] provided realistic performance validation: ca-GrQc (5,242 vertices) representing General Relativity collaborations, ca-HepTh (9,877 vertices) for High Energy Physics Theory, ca-AstroPh (18,772 vertices) for Astrophysics collaborations, ca-CondMat (23,133 vertices) for Condensed Matter Physics, and Slashdot0811 (77,360 vertices) for user interaction networks.

4 Benchmarking Results

Below we summarize our updated experiments, highlighting both strengths and weaknesses of the parallel δ-Stepping implementations .

4.1 Graph-Type Comparison

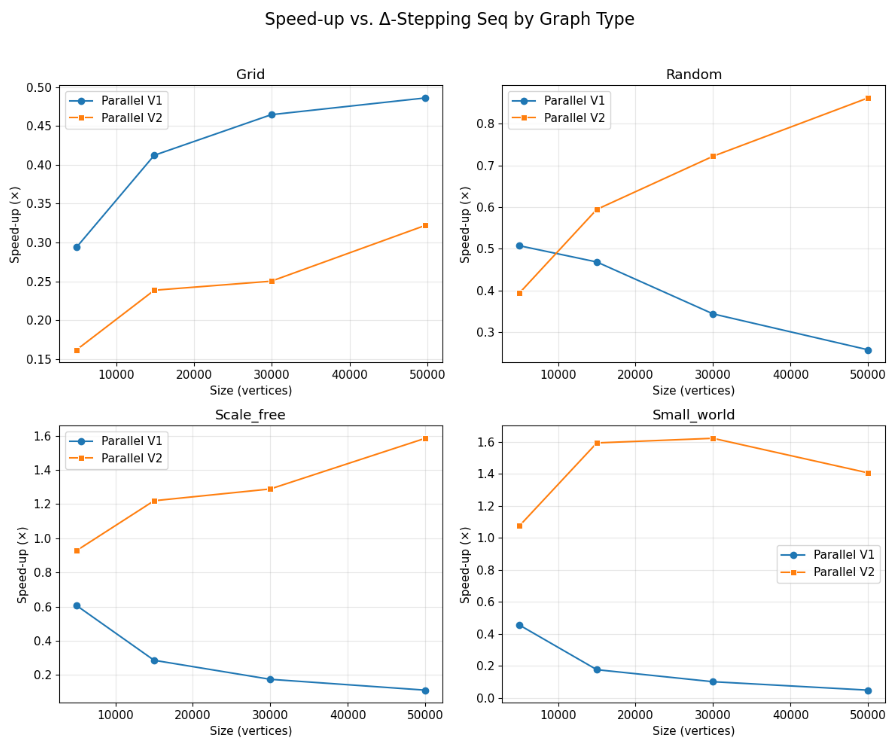

Figure 1: Speed-up (relative to sequential δ-Stepping) of Parallel V1 and V2 versus graph size, for grid, random, scale-free, and small-world topologies.

On simple grid graphs, neither V1 nor V2 ever matches sequential δ-Stepping: bucket overhead remains too large when buckets stay small. In random graphs, V1's performance degrades as size grows, whereas V2 steadily improves, nearly reaching parity at the largest sizes, large random structures eventually produce enough independent buckets to amortize overhead. Scale-free and small-world topologies reveal V2's strength: V2 uses clustered neighborhoods to form large, balanced buckets, outperforming the sequential version on sufficiently large graphs. V1's heavier locking and global sorts, by contrast, fail in every case.

4.2 Density Effects

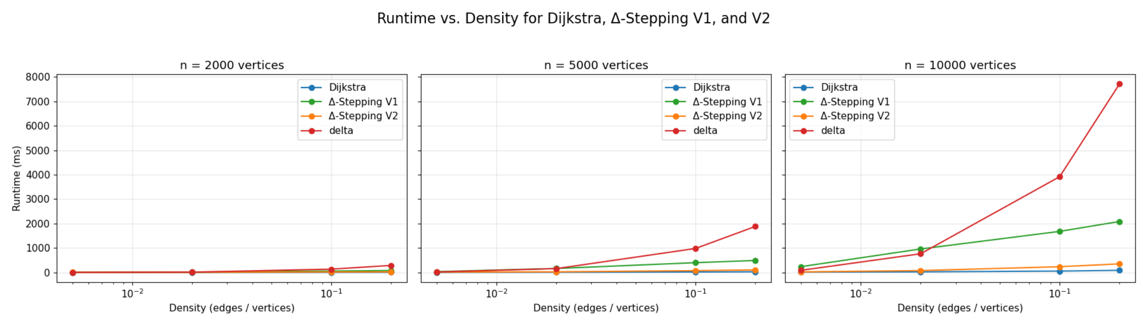

Figure 2: Speed-up of Parallel V1 and V2 over sequential δ-Stepping as density (m/n) increases, for three values of n.

When graphs are sparse, Dijkstra outperforms both parallel δ-Stepping variants—there simply isn't enough work per thread. As density rises above a small threshold, V1 approaches sequential speed and V2 quickly far exceeds it, because denser graphs produce larger buckets and more parallel work. At high density, V2 delivers dramatic speed-ups, while V1 gains moderately but remains slower than V2. In short, parallel δ-Stepping only becomes worthwhile when the average degree is large enough that bucket overhead is amortized.

4.3 Weight-Distribution Effects

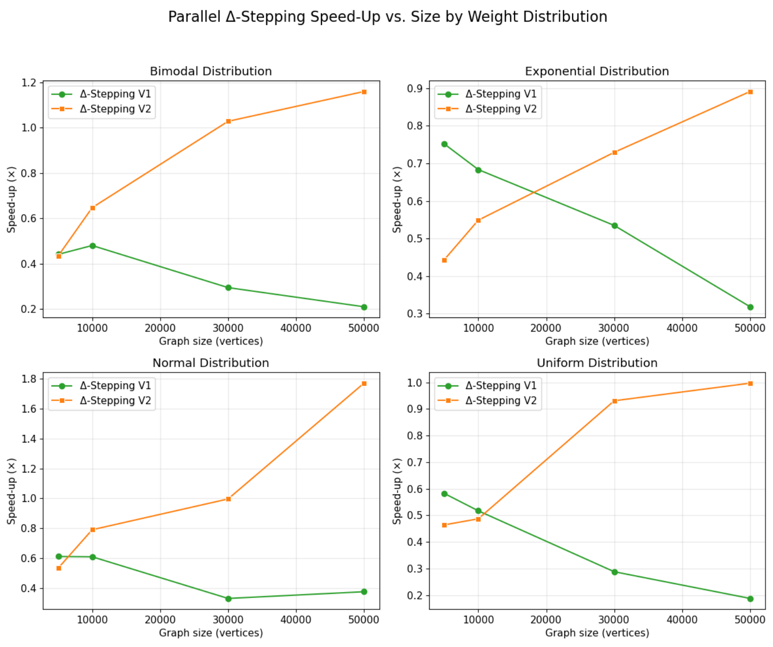

Figure 3: Speed-up of V1 and V2 over sequential δ-Stepping for a fixed random topology, under different weight distributions.

With weights drawn uniformly or from a heavy-tailed distribution, buckets fragment and neither V1 nor V2 surpasses sequential performance at moderate sizes. When weights follow a normal or bimodal pattern, buckets become more balanced as the graph grows, allowing V2 to break even or outperform sequential δ-Stepping, while V1's heavier synchronization still prevents it from ever reaching parity.

4.4 δ-Parameter Sensitivity

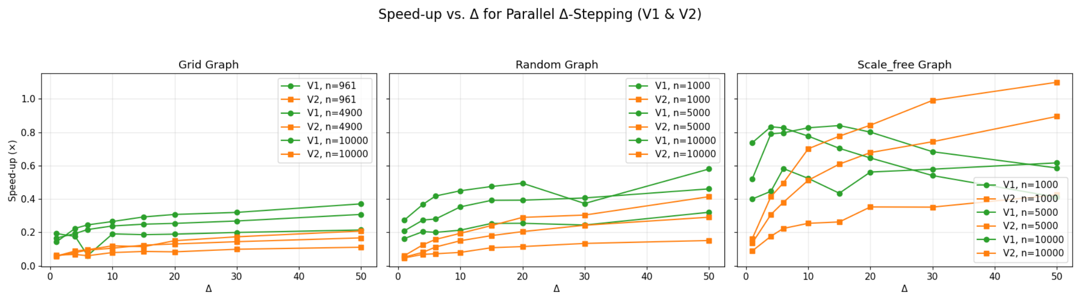

Figure 4: Speed-up of V1 and V2 over sequential δ-Stepping as δ varies, for random, grid, and scale-free graphs.

Choosing δ correctly is essential. On random graphs, our heuristic (`findDelta`) returns a small value that matches the peak of the parallel curves, but V2 still never exceeds sequential speed, since random topology lacks locality. On grid graphs, `findDelta` picks a moderate δ that underestimates the bucket size needed, so V2 peaks below parity. On scale-free graphs, `findDelta` suggests a larger δ than ideal; although V2 still outperforms sequential, it would perform better with a slightly lower δ. In short, our static heuristic often misses the true "sweet spot," leading to suboptimal parallel performance; a finer δ-selection strategy or dynamic adjustment could remedy this.

4.5 Thread Scaling

Parallel V2 scales well up to a modest number of threads (typically 2–4) but then experiences diminishing returns or slight slowdowns due to bucket contention and memory limits. V1 gains almost no benefit beyond one or two threads, since its synchronization bottlenecks overwhelm any parallel work. In highly regular graphs like a large grid, even V2 loses scalability as threads contend; in contrast, on scale-free or dense graphs, V2 sustains significant speed-up with a few threads before plateauing.

4.6 Real-World Graph Performance

On small or very sparse networks (e.g. ca-GrQc), neither parallel variant matches sequential δ-Stepping, as there is too little work per bucket. Once graphs surpass about 10 000–20 000 nodes (e.g. ca-HepTh, ca-AstroPh), V2's bucket optimizations pay off dramatically, delivering several-fold speed-up over sequential δ-Stepping. V1 remains slow in all cases. The largest dataset (Slashdot0811) highlights V2's strength—reducing sequential runtime by roughly fivefold—while V1 collapses under synchronization overhead. Overall, in realistic, mid-sized sparse networks, δ-Stepping V2 can beat sequential δ-Stepping when n is large enough, but it still trails optimized Dijkstra in many cases.

In summary, Parallel V2 is clearly superior to V1 in nearly every scenario, but its success hinges on forming large, balanced buckets—whether through topology, density, or weight distribution. Our `findDelta` heuristic often selects a suboptimal δ, which explains why performance sometimes lags: refining δ selection will be crucial for consistently high speed-ups. When δ is chosen well

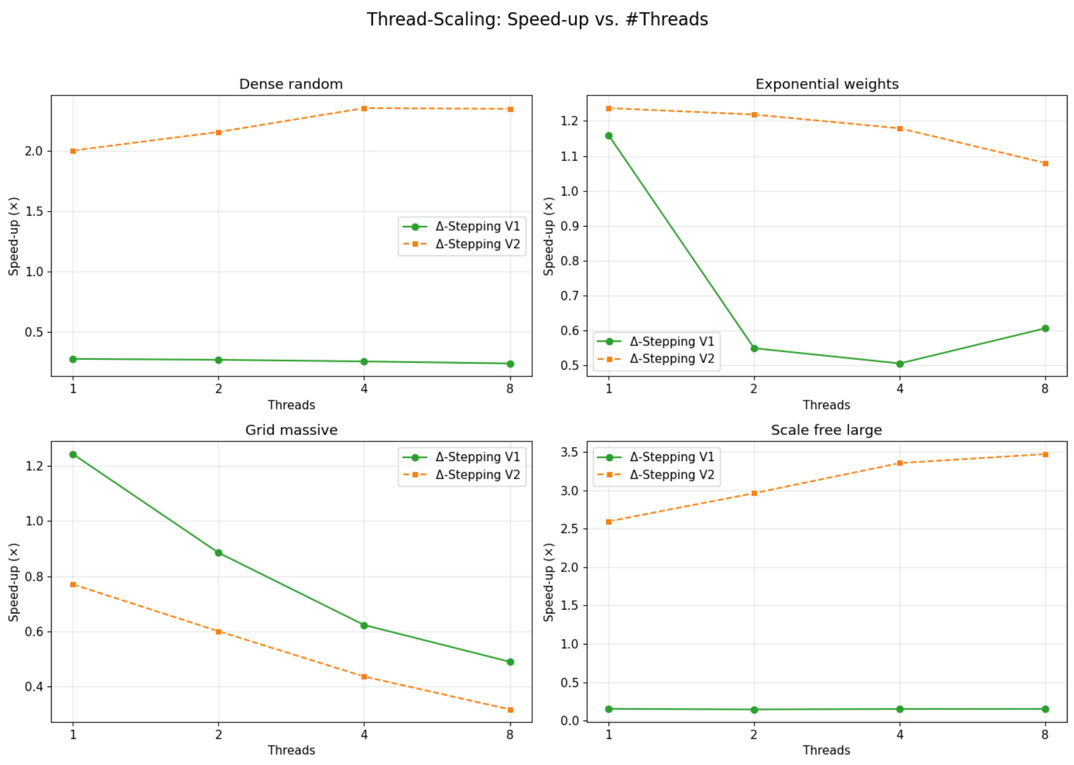

Figure 5: Speed-up of V1 and V2 over single-thread δ-Stepping as thread count increases, on four large graphs.

and thread count is limited to avoid contention, V2 can deliver dramatic acceleration; otherwise, overhead nullifies any parallel gains.

5 Discussion

5.1 Performance Dependencies

Parallel δ-Stepping's effectiveness hinges on bucket size, which is shaped by topology, density, and weight distribution. On regular grids or very sparse graphs, buckets remain too small and parallel overhead dominates. As density or clustering increases such as in dense random, scale-free, or small-world graphs—buckets grow large enough that V2's lightweight, in-place relaxations can amortize overhead and surpass sequential performance. V1's heavier locking prevents it from scaling in almost every scenario.

5.2 Limitation and Applications

Version 2 still inherits several bottlenecks we did not have time to remove. All threads write into one global bucket array, so when only a few buckets are active they queue on the same locks and parallelism collapses. Buckets are visited in numeric order; on grid-like graphs this means marching over long runs of empty buckets before reaching useful work.
Our `findDelta` implementation is not perfect and can miss the ideal Δ, and a bad choice magnifies every other cost. Finally we considered implementing a thread pool because in our current implementation we create an important amount of thread even for trivial taks. However we felt like this was a bit beyond our capacity and the scope of the course. Hence threads are still created and destroyed at every bucket phase, which is adding measurable overhead. Because

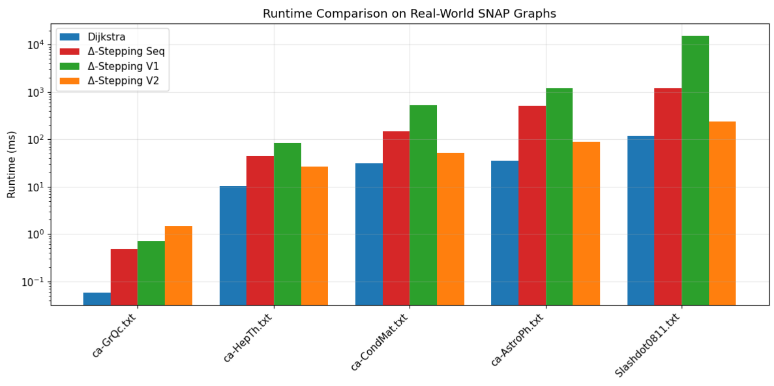

Figure 6: Speed-up of V1 and V2 over sequential δ-Stepping on five SNAP datasets.

of these issues, the current solver excels on medium-sized sparse graphs but may lag behind an optimized sequential Dijkstra on road grids or very dense social networks. .

6 Conclusion

This project into parallel shortest path algorithms reveals fundamental insights about challenges and opportunities in parallel graph computation. Our work demonstrates that parallel efficiency cannot be achieved simply by adding parallelism to sequential algorithms, it requires careful codesign of strategy, data structures, and synchronization mechanisms.
Parallel delta-stepping achieves significant speedups up to 10x under favorable conditions, but these conditions prove more restrictive than initially anticipated. Graph topology influences performance more dramatically than problem size alone, with sparse and regular structures enabling inferior efficiency compared to irregular ones. Current implementations face fundamental synchronization bottlenecks that limit scaling beyond modest thread counts, suggesting that improvement can still be made.

7 Code Implementation

The full implementation and additional resources can be found in the GitHub repository: `https://github.com/dansuissa/CSE305-Project.git`.

8 Contributions

Dan Suissa developed the sequential Dijkstra baseline and the initial parallel delta-stepping (v1), establishing the core algorithmic infrastructure. Adan Fhima implemented the optimized v2 parallel version and conducted comprehensive benchmarking across diverse graph types. Llms were used during development to assist with secondary tasks such as generating CSV test files and creating varied test scenarios. They were mainly used to support development within the main.cpp and graph.cpp files. The core algorithmic design and parallelisation logic were carried out by the us.

References

[1] Meyer, U. and Sanders, P. (2003). Delta-stepping: a parallel single source shortest path algorithm. *Journal of Algorithms*, 49(1), 114–152.

[2] Duriakova, E., Ajwani, D., and Hurley, N. (2019). Engineering a Parallel Δ-stepping Algorithm. In *2019 IEEE International Conference on Big Data (Big Data)* (pp. 281–290). IEEE.

[3] Jure Leskovec et al. Stanford Large Network Dataset Collection (SNAP). `https://snap.stanford.edu/data/`. Accessed 2025-06-06.