Adaptive Optimizations for Parallel Single-Source Shortest Paths

Runbang Hu
University of Texas at Arlington

Chaoqun Li
University of Texas at Arlington

Xiaojiang Du
Stevens Institute of Technology

Yuede Ji
University of Texas at Arlington

Abstract

The single-source shortest path (SSSP) problem is essential in graph theory with applications in navigation, biology, social networks, and traffic analysis. The Δ-Stepping algorithm enhances parallelism by grouping vertices into "buckets" based on their tentative distances. However, its performance depends on Δ values and graph properties. This paper introduces an adaptive parallel Δ-Stepping implementation with three innovations: neighbor reordering, bucket fusion, and graph type-aware Δ selection. Tested on 11 diverse graphs, it achieves an average 7.1× speedup over serial Dijkstra's algorithm on a 48-threads CPU.

ACM Reference Format:
Runbang Hu, Chaoqun Li, Xiaojiang Du, and Yuede Ji. 2025. Adaptive Optimizations for Parallel Single-Source Shortest Paths. In *FastCode Programming Challenge (FCPC '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3711708.3723450

1 Introduction

The single-source shortest path (SSSP) problem is a fundamental problem in graph theory with applications in navigation [26, 28], social network analysis [5, 9, 13, 17, 18, 24], traffic analysis [25], cybersecurity applications [6, 12, 15, 16], and computing other graph algorithms [10].

Dijkstra's algorithm is a classic algorithm for finding SSSP on non-negative weighted regular graphs, while its sequential priority queue operations limit parallelism [7, 14, 29]. To address that, the Δ-Stepping algorithm groups vertices into "buckets" based on tentative distances, enabling parallel processing. However, its performance depends on Δ values and graph properties, such as diameter, weight distribution, and sparsity. While previous optimizations exist for specific graphs [3, 8], an adaptive approach remains an open challenge [22, 31].

FCPC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1446-7/2025/03
https://doi.org/10.1145/3711708.3723450

This paper presents an adaptive parallel Δ-Stepping implementation that integrates graph-specific preprocessing and dynamic strategy selection. There are three key innovations. (i) *Neighbor reordering* reorganizes edges based on their weights and classifies graphs by properties such as diameter and density to improve Δ selection. (ii) *Bucket fusion* reduces synchronization overhead by combining consecutive processing rounds for the same bucket. It will relax the current bucket repeatedly until no new insertions occur. (iii) *Adaptively optimizations* adaptively switches between standard Δ-Stepping, and bucket fusion. This can optimize the trade-off between parallelism and synchronization.

We evaluated our method on 11 diverse graphs, achieving an average 7.1× speedup over serial Dijkstra's algorithm on a 48-thread system, with more than 12× speedup for dense, and small-diameter graphs.

2 Background and Challenge

To improve the performance of serial SSSP implementations, parallel approaches have been proposed, such as Bellman-Ford [11] and Δ-Stepping [27].

Δ-Stepping improves efficiency by reducing redundant computations in parallel Bellman-Ford while balancing parallelism and synchronization overhead. It groups vertices into buckets based on their tentative distances and processes them in two stages: first relaxing light edges ($\leq \Delta$) in parallel, followed by handling heavy edges ($> \Delta$). This design maximizes concurrency for small-weight edges while minimizing excessive fragmentation and synchronization, making Δ-Stepping highly effective for favorable graph structures.

Selection of Δ value. The bucket size directly impacts performance. A small Δ increases parallelism but raises synchronization costs, while a large Δ reduces synchronization but may over-group vertices, limiting efficiency. Finding the optimal Δ across different graph types remains challenging.

Workload imbalance is another challenge in parallel SSSP algorithms. Sparse graphs may lead to underutilization of resources due to mostly empty buckets, while dense graphs with uniform edge weights require careful dependency management to prevent bottlenecks. Existing solutions, such as bucket fusion (which consolidates adjacent buckets to reduce synchronization) and dynamic Δ tuning, help mitigate these issues. However, these techniques often require manual tuning and are not universally effective across all graph types.

3 Methodology

Our method is based on Δ-Stepping algorithm by creating parallel regions to reduce repeated thread allocation overhead. First, we implement a hybrid bucket architecture combining thread-local storage with synchronized global coordination, where each thread independently manages its bucket for tentative distance updates, minimizing communication between threads during light edge relaxation. After processing the light edges, each thread handles heavy edges once. Periodically, the threads atomically commit their local minimum bucket index to a shared global bucket, consolidating related vertices, and processing them in parallel. In addition, we apply three optimizations as discussed below, including neighbor reordering, bucket fusion, and adaptive optimizations.

Neighbor Reordering. The Δ-Stepping algorithm requires separating light and heavy edges, which can be done in three ways: (i) scanning edges at runtime, (ii) using a status array of size $|E|$ to mark edge types, (iii) maintaining separate queues for light and heavy edges. However, these approaches have drawbacks: redundant computation (i, ii), extra memory usage (ii, iii), and poor cache utilization (iii). To address these issues, we introduce neighbor reordering. Particularly, we reorder each vertex's adjacency list in ascending weight order and use an additional offset array with the size of $|V|$ ($offset[|V|]$) to track the number of light edges per vertex.

During SSSP computation, for a vertex v, light edges are stored in $[beg\_pos[v], beg\_pos[v] + offset[|V|] - 1]$, Heavy edges are in $[beg\_pos[v] + offset[|V|], beg\_pos[v+1] - 1]$. This design minimizes redundant processing and extra space usage while improving cache efficiency with only a small additional $|V|$-sized array.

Bucket Fusion. A single bucket often requires multiple processing rounds to relax the bucket, updating tentative distances, and reinserting vertices until no new insertions occur. This repeated reprocessing introduces significant overhead. To mitigate this, we apply bucket fusion, which combines consecutive processing rounds within the same bucket. By allowing threads to continue execution without frequent global barriers, this technique substantially lowers synchronization costs. We introduce a threshold-based approach (e.g., 1,000 vertices) to optimize workload distribution: If a bucket has a small workload, the owning thread processes it immediately. Otherwise, it is passed to a global bucket for shared processing. This strategy prevents straggler overload and ensures efficient workload balancing.

Adaptive Optimizations. Real-world graphs vary widely in structure, including sparse vs. dense, real vs. uniform weights, and small vs. large diameters. To optimize Δ-Stepping across different graphs, we introduce adaptive strategies.

Δ *value* is crucial for balancing parallelism and redundant computation in the Δ-Stepping algorithm. It determines the granularity of buckets, influencing synchronization costs and workload distribution. We define Δ as: $\Delta = (max\_weight - min\_weight) \times \alpha + min\_weight$. Where α is a hyperparameter that controls how aggressively Δ is adjusted. It is determined empirically based on experiments, ensuring an optimal trade-off between parallelism and computational efficiency. A smaller α leads to a smaller Δ, increasing the number of buckets, enhancing parallelism but raising synchronization overhead. A larger α value results in a larger Δ, reducing the number of buckets and synchronization but limiting parallel execution. To adapt Δ based on graph characteristics: (i) For dense, small-diameter graphs, Δ is set below 90% of edge weights, capturing most distance contributions while minimizing heavy-edge updates. (ii) For sparse, large-diameter graphs, a larger Δ value prevents excessive iterations and mitigates workload imbalance, reducing underutilization of threads.

Small graphs are special as the dominant cost may shift from computation to thread management and synchronization overhead, making multi-threading less efficient than a serial approach. In such cases, we switch to serial Δ-Stepping with a small Δ value to optimize performance. A graph is classified as small if its Compressed Sparse Row (CSR) [4] representation fits within the L1 cache, ensuring efficient memory access. CSR is a graph representation that is widely used in graph analytics [10, 19–21, 23]. Given a 32 KB L1 cache, and assuming Unsigned integer for the begin position and adjacency list arrays, and Single-precision floating point for edge weights, this condition is expressed as $4|V| + 4|E| + 4|E| < 32 * 1024$.

Table 1. Graph benchmarks (sorted by vertex count).

Graph	Abbr.	Diameter	|V|	|E|	Weight
Random_1k_5k	R1	Small	1k	5K	random
Random_2k_10k	R2	Small	2k	10K	random
Random_10k_50k	R3	Small	10k	50K	random
Collab network	CN	Small	1.1M	113M	uniform
Social network	SN	Small	4.9M	85.5M	skewed
Web graph	WG	Small	6.6M	300M	skewed
Synthetic dense	SD	Small	10M	1B	uniform
Synthetic sparse	SS	Large	10M	40M	uniform
Road network 1	RN1	Large	22.1M	30M	real
kNN graph	KG	Large	24.9M	158M	real
Road network 2	RN2	Large	87M	112.9M	real

4 Experiment

4.1 Overall benchmark

Table 1 summarizes the details of the tested graph benchmark. We evaluate our method on the graph benchmarks and compare against a baseline algorithm, i.e., the priority queue-based Dijkstra's algorithm. We use a CPU instance with 48 threads and 96 GB memory on the Speedcode platform [30].

Table 2 summarizes the results. (i) On average, our method achieves 7.1× speedup over the Dijkstra's algorithm, which is a significant improvement. (ii) For most graphs, our parallel Δ-Stepping algorithm significantly outperforms the Dijkstra's algorithm. (iii) Our method underperforms on three graphs

Table 2. Runtime performance.

Graph	Dijkstra (s)	Ours (s)	M edges/s	Speedup (×)
R1	0.00016	0.00008	130.47	2
R2	0.00036	0.00016	126.75	2.25
R3	0.00219	0.00062	161.42	3.5
CN	1.37	7.50	14.72	0.18
SN	4.30	10.90	7.7	0.4
WG	8.41	0.49	605.74	17
SD	28.54	1.56	629.41	18.2
SS	4.28	0.21	186.01	20.3
RN1	7.89	2.29	25.44	3.1
KG	4.58	21.72	7.11	0.21
RN2	36.76	3.07	70.3	12
Avg.	-	-	-	7.1

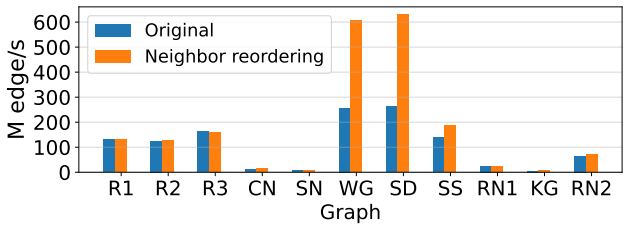

Figure 1. Runtime performance (in million edges/second) with and without neighbor reordering.

(CN, SN, and KG), which illustrates that parallelism does not always guarantee performance gains. Additionally, the lack of a clear boundary between graph categories can occasionally lead to suboptimal strategy selection.

4.2 Benefits of Specific Techniques

Neighbor reordering. Figure 1 shows the performance in terms of million edges per second with and without neighbor reordering. We have two interesting observations. (i) On average, neighbor reordering can improve the performance by 1.4× speedup on all the graphs, showing a stable performance improvement. (ii) Neighbor reordering achieves the highest speedup for WG and SD graphs, which are 2.3×, 2.4×, respectively. This clearly shows the benefits of neighbor reordering.

Bucket fusion. Figure 2 shows the performance of Dijkstra's algorithm, original (without bucket fusion), and with bucket fusion. We show the performance for three representative graphs, including a sparse graph (SN), a dense graph (KG), and an extremely dense graph (CN). We observe that it has varying effects based on graph properties. In sparse graphs (e.g., SN), it lowers synchronization overhead and improves performance. However, in dense graphs (e.g., KG), fusion causes workload imbalance, as profiling shows a single thread often dominates fusion operations, leaving others idle. This workload imbalance arises from non-uniform bucket sizes, where frequent fusion is unevenly distributed.

Small graph. Figure 3 shows the performance on small graphs. The serial Δ-Stepping algorithm demonstrates superior performance, as the overhead associated with thread

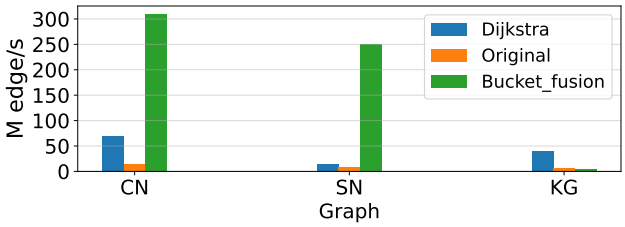

Figure 2. Runtime performance of bucket fusion.

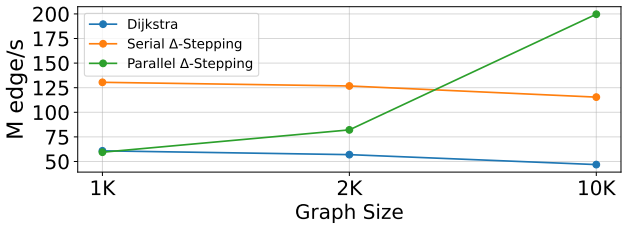

Figure 3. Runtime performance for the small graphs.

management in parallel Δ-Stepping outweighs its parallelization benefits. When the number of vertices is less than 2K, serial Δ-Stepping achieves a 1.5−2× speedup over the parallel implementation. However, as the graph size increases, the parallel algorithm gradually surpasses the serial version, while the efficiency of the serial approach declines, eventually converging to the performance of Dijkstra's algorithm. When the graph reaches 10K vertices, the parallel method achieves a 1.7× speedup over the serial approach.

5 Related Work and Conclusion

Improvements of Δ-Stepping [27] have focused on tuning Δ to graph properties and optimizing buckets [32] (e.g., fusion and priority-aware splitting) to reduce synchronization overhead. While merging underpopulated buckets improves performance in road networks, it can cause load imbalance in dense graphs [1, 2, 33]. Preprocessing, such as edge sorting, helps reduce redundant relaxations but often relies on static heuristics that overlook graph heterogeneity.

This paper presents an adaptive parallel Δ-Stepping implementation that integrates three key innovations, including neighbor reordering, bucket fusion, and adaptive optimization. We test our method on 11 diverse graphs in terms of size, diameter, density, and edge weight variance. On a CPU server with 48 threads, our method achieves an average of 7.1× speedup over the serial Dijkstra's algorithm.

Acknowledgment

This work was supported in part by National Science Foundation grants 2331301, 2508118, 2516003, 2419843. The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views of the National Science Foundation, or the U.S. Government.

References

[1] Matthew Agostini, Francis O'Brien, and Tarek Abdelrahman. 2020. Balancing graph processing workloads using work stealing on heterogeneous CPU-FPGA systems. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–12.

[2] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 44–55.

[3] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[4] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[5] Jian Cao, Qiang Li, Yuede Ji, Yukun He, and Dong Guo. 2016. Detection of forwarding-based malicious URLs in online social networks. *International Journal of Parallel Programming* 44, 1 (2016), 163–180.

[6] Lei Cui, Jiancong Cui, Yuede Ji, Zhiyu Hao, Lun Li, and Zhenquan Ding. 2023. API2Vec: Learning Representations of API Sequences for Malware Detection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–273.

[7] Edsger W Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[8] 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*. 184–197.

[9] Bo Feng, Qiang Li, Yuede Ji, Dong Guo, and Xiangyu Meng. 2019. Stopping the cyberattack in the early stage: assessing the security risks of social network users. *Security and Communication Networks* 2019 (2019).

[10] Wang Feng, Shiyang Chen, Hang Liu, and Yuede Ji. 2023. Peek: A Prune-Centric Approach for K Shortest Path Computation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[11] Gaurav Hajela and Manish Pandey. 2014. Parallel implementations for solving shortest path problem using Bellman-Ford. *International Journal of Computer Applications* 95, 15 (2014).

[12] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In *The 33rd USENIX Security Symposium (USENIX Security)*.

[13] Yukun He, Qiang Li, Jian Cao, Yuede Ji, and Dong Guo. 2017. Understanding socialbot behavior on end hosts. *International Journal of Distributed Sensor Networks* 13, 2 (2017), 1550147717694170.

[14] Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, and Novica Nosovic. 2012. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *2012 proceedings of the 35th international convention MIPRO*. IEEE, 1811–1815.

[15] Yuede Ji, Lei Cui, and H. Howie Huang. 2021. BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In *16th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*.

[16] Yuede Ji, Mohamed Elsabagh, Ryan Johnson, and Angelos Stavrou. 2021. DEFInit: An Analysis of Exposed Android Init Routines. In *30th USENIX Security Symposium (USENIX Security)*.

[17] Yuede Ji, Yukun He, Xinyang Jiang, Jian Cao, and Qiang Li. 2016. Combating the evasion mechanisms of social bots. *Computers & Security* (2016).

[18] Yuede Ji, Yukun He, Xinyang Jiang, and Qiang Li. 2014. Towards social botnet behavior detecting in the end host. In *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 320–327.

[19] Yuede Ji and H. Howie Huang. 2020. Aquila: Adaptive Parallel Computation of Graph Connectivity Queries. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[20] Yuede Ji, Hang Liu, and H. Howie Huang. 2018. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 731–742.

[21] Yuede Ji, Hang Liu, and H. Howie Huang. 2020. SwarmGraph: Analyzing Large-Scale In-Memory Graphs on GPUs. In *International Conference on High Performance Computing and Communications (HPCC)*. IEEE.

[22] Mandeep Khadka, Rachata Ausavarungnirun, and Christian Terboven. 2023. A Case Study of an Adaptive Delta-Stepping Algorithm in OpenMP. In *2023 Research, Invention, and Innovation Congress: Innovative Electricals and Electronics (RI2C)*. IEEE, 112–119.

[23] Hang Liu, H Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. ACM, 403–416.

[24] Rui Liu, Raj Velamur Srinivasan, Kiyana Zolfaghar, Si-Chi Chin, Senjuti Basu Roy, Aftab Hasan, and David Hazel. 2014. Pathway-finder: An interactive recommender system for supporting personalized care pathways. In *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 1219–1222.

[25] Tiantian Liu, Zijin Feng, Huan Li, Hua Lu, Muhammad Aamir Cheema, Hong Cheng, and Jianliang Xu. 2020. Shortest path queries for indoor venues with temporal variations. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2014–2017.

[26] Nirmesh Malviya, Samuel Madden, and Arnab Bhattacharya. 2011. A continuous query system for dynamic route planning. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 792–803.

[27] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[28] Sara Nazari, M Reza Meybodi, M Ali Salehigh, and Sara Taghipour. 2008. An advanced algorithm for finding shortest path in car navigation system. In *2008 First International Conference on Intelligent Networks and Intelligent Systems*. IEEE, 671–674.

[29] Amir R Soltani, Hissam Tawfik, John Yannis Goulermas, and Terrence Fernando. 2002. Path planning in construction sites: performance evaluation of the Dijkstra, A*, and GA search algorithms. *Advanced engineering informatics* 16, 4 (2002), 291–303.

[30] Speedcode. [n. d.]. Speedcode. https://speedcode.org/

[31] Upasana Sridhar, Maia P Blanco, Rahul Mayuranath, Daniele G Spampinato, Tze Meng Low, and Scott McMillan. 2019. Delta-stepping SSSP: From vertices and edges to GraphBLAS implementations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 241–250.

[32] 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*. 158–170.

[33] Yuan Zhang, Huawei Cao, Jie Zhang, Yiming Sun, Ming Dun, Junying Huang, Xuejun An, and Xiaochun Ye. 2023. A Bucket-aware Asynchronous Single-Source Shortest Path Algorithm on GPU. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*. 50–60.