Toward Efficient Asynchronous Shortest Path

Marco D'Antonio
Supervised by Hans Vandierendonck, Thai Son Mai
Queen's University Belfast
Belfast, United Kingdom

Abstract

Single-Source Shortest Path (SSSP) is a fundamental graph problem that arises in various applications and complex problems. State-of-the-art solutions to the parallel SSSP problem create parallelism through priority coarsening, which results in redundant work and reduces the efficiency of the solution. Our research investigates a novel solution for SSSP that addresses the parallelism-redundant work problem using on-demand priority relaxation. Our preliminary results show that our implementation has competitive or better performance than state-of-the-art implementations of SSSP, including GAP, GBBS, and the MultiQueue, on 13 diverse graphs, with speedups up to 2.94× higher than the state of the art.

CCS Concepts

• **Theory of computation** → **Shared memory algorithms**; **Shortest paths**; *Concurrent algorithms.*

Keywords

Graph Algorithms, Single-Source Shortest Path, Shared-Memory

1 Problem Statement

Single-Source Shortest Path (SSSP) is an important problem that arises in various domains, including routing [17], network analysis [16, 31], and calculating the betweenness centrality of a network [6]. Given a weighted graph $G = (V, E, w)$ with a set of vertices V, a set of edges $E = \{(u, v) \mid (u, v) \in V^2 \wedge u \neq v\}$, an edge weight function $w : E \to \mathbb{R}_{\geq 0}$, and a source vertex $s \in V$, the Single-Source Shortest Path problem is the problem of finding the shortest path from s to every other vertex in the graph.

SSSP is usually solved using either Δ-stepping [20] or parallel Dijkstra's algorithm [13]. The main challenge in parallel SSSP is the trade-off between parallelism and redundant work. The sequential Dijkstra's algorithm completes with minimal work due to exploring paths based on priority order. However, the priority queue does not expose enough parallelism to be efficiently parallelized. Creating parallelism requires an alteration of the priority order, which we define as priority drifting. This leads to less efficient exploration and results in redundant work, such as traversing suboptimal paths. Δ-stepping overcomes the lack of parallelism through Δ-coarsening: the distance of each vertex is coarsened by a factor Δ, and vertices with the same coarsened distance are processed in parallel [20]. Recently, relaxed priority queues – that allow elements to be extracted in a relaxed order – have been proposed to improve parallelism of the traditional Dijkstra's algorithm [1, 23, 24, 29].

We conjecture that the priority drifting introduced by both solution approaches is indiscriminate and hence sub-optimal, as they may introduce priority drifting at times when it is not helpful, and may fail to introduce priority drifting when it is necessary. We claim it is necessary to identify events during the execution of SSSP algorithms that are good occasions for introducing priority drifting, and that this will lead to a more balanced and justified priority drifting, hence the execution of less redundant work and increased efficiency.

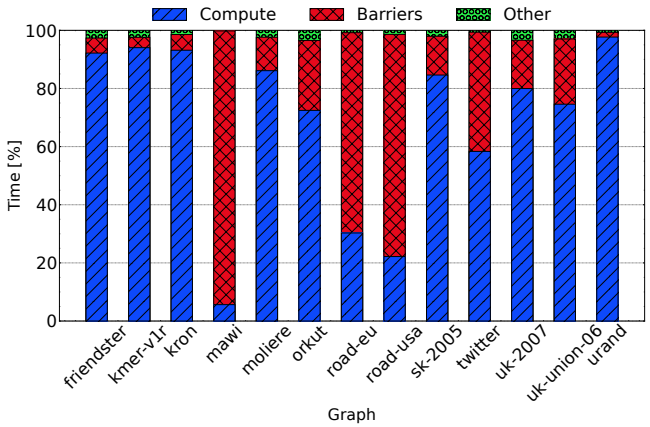

Figure 1: The execution breakdown of synchronous Δ-stepping on the real graph datasets of our experimental evaluation using the GAP [3].

2 Related Work

Dijkstra's algorithm [13] has the best-known bound of $O(|E| + |V| \log |V|)$ time complexity using a priority queue [7, 25], greedily processing the closest vertices to the source in the frontier of unsettled vertices. Since distances are stored in a priority queue, we often refer to the distance as priority. Note that the highest priority vertex has the smallest distance from the source. As is, Dijkstra's algorithm does not expose enough parallelism to be efficiently parallelized because, in order to respect priorities, we would only find parallelism of degree k if there are k vertices with the same, highest priority in the queue. It is rare that k would be high. Relaxed concurrent priority queues solve this problem by sacrificing work efficiency. By relaxing priorities, relaxed priority queues allow threads to retrieve one of the k closest vertices to the sources [1, 23, 24, 29].

The other common approach to parallel SSSP is **Δ-stepping** [20]. It maintains the frontier of active vertices in a set of buckets. Bucket i stores vertices u such that the distance of u from the source, $d(u) \in [i \cdot \Delta, (i+1) \cdot \Delta)$, where Δ is a graph-dependent constant. Therefore, vertex v with distance $d(v)$ will be stored in bucket $i' = \lfloor d(v)/\Delta \rfloor$. The buckets are then processed in parallel, processing each of the vertices in them. The Δ-coarsening of priorities creates **priority drifting**: mapping vertices with different distances into the same bucket leads to processing that is out of the work-efficient order

of Dijkstra's algorithm. This additional processing is considered redundant work and reduces work efficiency. The larger the Δ, the lower the work efficiency. However, the loss of work efficiency is counterbalanced by the creation of more parallelism. As a result, the choice of the Δ parameter is crucial to balance the trade-off between efficiency and parallelism.

State-of-the-art parallel Δ-stepping implementations often emulate the sequential behavior of the original algorithm, i.e. the parallel computation is organized in bulk-synchronous steps [26], during which buckets are processed in parallel. Once a bucket has been processed, threads synchronize through a barrier and coordinate to prepare the bucket for the next step. In more detail:

- The **GAP Benchmarking Suite** [3] implements Δ-stepping using thread-local buckets. At each step, a shared frontier array is populated and then processed in parallel. GAP implements the bucket fusion optimization [33], in which each thread processes the local content of the current bucket after processing the frontier. This allows for a reduction in the number of steps and, therefore, the synchronization costs.
- **Julienne** [11] implements a centralized, parallel bucketing structure. It provides an efficient parallel interface to retrieve all vertices mapped to a bucket and to apply updates and resize the buckets in parallel.

Other synchronous approaches that we will consider are Δ^*-stepping and ρ-stepping [14], which process vertices with a distance up to a certain threshold in parallel at each step. The Lazy-Batched Priority Queue is introduced to support this framework and is implemented as a parallel hash-bag to extract and update vertices [27]. One drawback of synchronous implementations is the cost of thread synchronization, as a barrier must be issued at the end of each step. This can be problematic in large-diameter graphs, such as road networks, due to the large number of steps required while few vertices are processed per step. Moreover, skewed-degree graphs can also incur large synchronization costs, as due to workload imbalance issues, threads processing high-degree vertices take longer to complete and can hold back other threads from progressing. We show this for the graphs used in our experimental evaluation in Figure 1.

Asynchronous designs for both Dijkstra's algorithm and Δ-stepping have been proposed in the literature. The parallel Dijkstra's algorithm implementations use a relaxed priority queue, and threads can independently retrieve vertices from the priority queue. The asynchronous Δ-stepping implementations use different underlying data structures, with lower sequential overheads since priorities are coarsened.

- The **MultiQueue** [24] is a relaxed priority queue used to implement a parallel version of Dijkstra's algorithm. It uses a total number of cp lock-protected priority queues, where c is a tuning parameter and p is the number of threads. Vertices are extracted by randomly selecting two queues and extracting the higher-priority vertex, while generated vertices are pushed to a randomly chosen queue. The rank error of the relaxed priority queue is $O(p \log p)$ w.h.p.

In the future, we will consider Galois' [21, 22] implementation of SSSP, the Multi Bucket Queue [32], and the Stealing MultiQueue [23]. In the case of a priority queue-based implementation,

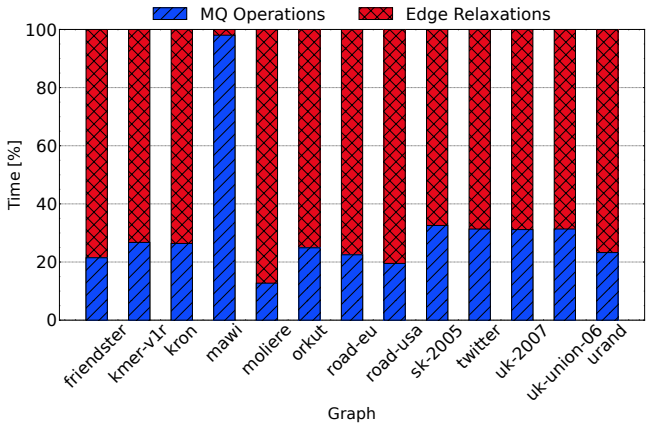

Figure 2: The execution breakdown in MultiQueue (MQ) operations (Push+Pop) and algorithm execution (Edge Relaxations) of asynchronous parallel Dijkstra's algorithm on the real graph datasets of our experimental evaluation using the MultiQueue [24, 28].

the execution time of an algorithm is mainly split between computation and queue operations (Figure 2). The queue operations include not only the synchronization costs, but also the sequential costs of managing the priority queue. In the case of the MultiQueue, each priority queue is a d-ary heap, with non-constant insertion and deletion costs. The chart shows that, for most graphs, the time spent accessing concurrent shared data structures is between 20-30% of the execution time.

3 Proposed Approach

We propose a novel algorithm, Wasp, that introduces priority drifting only when high-priority work is not available. This strategy allows Wasp to increase thread occupancy when parallelism is low, e.g., on large-diameter graphs, keeping threads busy instead of idle. On the other hand, when parallelism is available, e.g., on small-diameter graphs, Wasp follows the priority order, minimizing the generated redundant work.

Wasp organizes vertices in buckets based on their distance from the source, similar to Δ-stepping. However, Wasp operates asynchronously, allowing threads to progress independently without waiting for others to complete a priority level. Each thread works with a distributed bucket structure consisting of:

- Thread-local buckets for each coarsened priority level.
- A special work-stealing enabled "current bucket" that holds vertices at the current priority level.

This design eliminates synchronization overhead with constant-time insertion and deletion operations, while enabling workload balancing through work stealing. In contrast, the MultiQueue, for example, has non-constant sequential insertion and deletion operations, due to the underlying heap structure.

During execution, each thread extracts vertices from its current bucket and processes them if they aren't stale (meaning no better path has been found concurrently). When processing edges, destination vertices are updated using atomic Compare-and-Swap

Table 1: Graph datasets used in the experimental evaluation. $|V|$ is the number of vertices, $|E|$ is the number of edges.

Graph	$\|V\|$	$\|E\|$	Graph Type
Friendster [30]	68.3 M	2.58 B	Social Network
Kmer-v1r	214. M	465.4 M	Biological Network
Kron [19]	134.2 M	4.22 B	Synthetic Graph
Mawi	226.2 M	480 M	Network Traffic
Moliere	30.2 M	6.67 B	Semantic Network
Orkut [30]	3.1 M	234.4 M	Social Network
Road-EU [2]	54.1 M	108.1 M	Road Network
Road-USA [10]	23.9 M	57.7 M	Road Network
sk-2005	50.6 M	1.93 B	Web Crawl
Twitter [18]	61.6 M	1.46 B	Social Network
uk-2007 [2]	104.3 M	6.6 B	Web Crawl
uk-union-06	131.8 M	7.11 B	Web Crawl
Urand [15]	134.2 M	4.29 B	Synthetic Graph

operations. Updated vertices are then pushed either to the current bucket or thread-local buckets based on their priority level. When the current bucket of a thread becomes empty, the thread attempts to steal high-priority vertices from other threads before processing its own lower-priority vertices; this allows the algorithm to always prioritize high-priority work. This is a fundamental difference from traditional work-stealing protocols [4, 5] as the priority-based mechanism enables more efficient scheduling choices. Importantly, stolen vertices are processed immediately and cannot be stolen again. Without synchronization barriers, threads can advance independently, i.e. when one thread exhausts its current bucket, it can steal work while others continue processing. After handling stolen vertices, threads process any newly discovered vertices in their current bucket. If none exist, they find the next priority level to work on from their thread-local buckets. This asynchronous approach enables greater parallelism than synchronous schedulers, where threads might idle waiting for others before advancing to the next bucket. Wasp achieves this while still prioritizing high-priority work through its efficient work-stealing mechanism.

4 Preliminary Results

Wasp is evaluated on two machines: Epyc and Xeon. Epyc is a 128-core AMD EPYC 7713 processor, while Xeon is a 64-core Intel Sapphire Rapids Xeon Gold 6438Y+ processor with hyper-threading. In our preliminary results, we compare Wasp against three state-of-the-art implementations: the GAP benchmarking suite [3], the GBBS graph framework [12], that uses the Julienne bucketing approach described in Section 2, and parallel Dijkstra's algorithm using thr MultiQueue [24].

The evaluation is carried out on 13 graphs, 11 of which are real-world graphs. The graphs, their number of vertices and edges, and their type are listed in Table 1. We use different types of graphs: road and biological graphs are characterized by large diameters and vertices with low degrees; Urand is an Erdős–Rényi random graph with uniform degree distribution, while the others have a skewed degree distribution and a small diameter.

Table 2: Execution time in seconds of SSSP. The fastest execution time for each graph is highlighted in bold. The last row shows the aggregated speedup of Wasp over the baselines.

	Epyc				Xeon			
Graph	Wasp	GAP	GBBS	MQ	Wasp	GAP	GBBS	MQ
Friendster	**1.52**	1.75	3.28	3.63	**1.07**	1.17	1.82	2.56
Kmer-v1r	**1.15**	1.33	3.71	2.03	**0.96**	0.97	2.35	1.92
Kron	**2.80**	3.00	4.16	3.88	1.75	**1.66**	2.22	3.46
Mawi	0.77	2.80	**0.55**	31.93	1.95	3.46	**0.94**	77.78
Moliere	**2.15**	2.43	4.56	7.59	**1.41**	1.62	2.08	2.91
Orkut	**0.10**	0.13	0.16	0.21	**0.07**	0.10	0.11	0.18
Road-EU	**0.15**	0.23	1.18	0.39	**0.13**	0.15	0.83	0.42
Road-USA	**0.13**	0.24	6.05	0.21	**0.15**	0.21	3.98	0.22
sk-2005	**0.51**	0.63	1.26	1.28	**0.56**	0.61	0.96	1.18
Twitter	**0.97**	1.83	1.88	1.66	**0.78**	1.38	1.36	1.37
uk-2007	**0.57**	0.94	1.38	2.30	**0.43**	0.70	0.92	2.04
uk-union-06	**1.63**	2.47	2.90	6.42	**1.74**	2.48	2.63	5.90
Urand	**5.59**	6.06	9.05	12.12	3.59	**3.54**	5.40	8.67
gmean		1.45×	2.62×	2.91×		1.27×	2.1×	2.98×

Tuning the Δ parameter in Δ-stepping appropriately is critical for good performance [8]. In the case of Wasp, selecting $\Delta = 1$ for skewed-degree graphs is a safe estimate resulting in reliably good performance, with at most a 20% performance loss compared to the optimal Δ. This is a direct result of Wasp's design, as it creates parallelism and priority drifting through Δ-coarsening and selectively executing low-priority tasks.

Table 2 shows the execution time and aggregate speedup using geometric mean (gmean) of Wasp and the baseline we compare against on both machines. Wasp is slower only in four out of 26 cases. Wasp performs better than GBBS in most cases. Wasp obtains better speedups on large-diameter graphs. Due to the bulk-synchronous nature of GBBS, many barriers are performed because of the large diameter, while only a few relaxations happen at each round due to the low degree of vertices. Conversely, GBBS is more competitive on scale-free graphs such as Kron. Wasp has a 1.41× and a 2× slowdown, respectively, on Epyc and Xeon compared to GBBS on the Mawi graph. This is due to the graph's structure, which presents a high-degree vertex connected to 93% of the vertices and is better handled with the synchronous structure of GBBS. To achieve better performance on this graph, we prevent adding degree-1 leaf vertices to the scheduler to avoid useless work. We use this optimization on all graphs.

Wasp consistently outperforms MQ for all graphs. Speedups range from 1.38× to 4× on Epyc and from 1.52× to 4.7× on Xeon. Compared to GAP, higher performance improvement is due to the reduction of thread idleness. Wasp's speedup over GAP reaches 3.6× for Mawi and 1.9× for Twitter on Epyc, and 1.7× for Mawi and Twitter on Xeon. Wasp is competitive with GAP on graphs like Friendster and Kmer-v1 and on par with GAP on synthetic graphs. For Kron and Uran on Xeon, Wasp is slightly behind.

5 Conclusion and Future Work

Wasp is a promising novel algorithm for solving SSSP in parallel. It employs an asynchronous design with distributed buckets, enabling load balancing and priority drifting under low parallelism. Experiments on 13 graph datasets show Wasp outperforms GAP by

1.36×, GBBS by 2.34×, and MQ by 2.94× using their best Δ values. We used Wasp in the Fastcode Programming Challenge, as part of a scheduler for solving SSSP on real-world graphs [9], achieving first place in the SSSP track.

In the future, we aim to further improve our algorithm with other optimizations, in particular for NUMA-awareness and for increasing the workload-balancing on skewed-degree graphs. Moreover, we would like to address the Δ-tuning problem, a challenge that is still unsolved – although our algorithm provides good performance with a small Δ for skewed-degree graphs – proposing a data structure that automatically balances the parallelism-redundant work trade-off, without requiring manual tuning of parameters.

References

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A Scalable Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 11–20. doi:10.1145/2688500.2688523

[2] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2012-02-13/2012-02-14. *Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop*. Contemporary Mathematics, Vol. 588. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science. https://sites.cc.gatech.edu/dimacs10

[3] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. doi:10.48550/arXiv.1508.03619 arXiv:1508.03619 [cs]

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. *ACM SIGPLAN Notices* 30, 8 (Aug. 1995), 207–216. doi:10.1145/209937.209958

[5] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. doi:10.1145/324133.324234

[6] Ulrik Brandes. 2001. A Faster Algorithm for Betweenness Centrality*. *The Journal of Mathematical Sociology* 25, 2 (June 2001), 163–177. doi:10.1080/0022250X.2001.9990249

[7] Gerth Stølting Brodal. 1996. Worst-Case Efficient Priority Queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '96)*. Society for Industrial and Applied Mathematics, USA, 52–58.

[8] Venkatesan T. Chakaravarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, and Yogish Sabharwal. 2017. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (July 2017), 2031–2045. doi:10.1109/TPDS.2016.2634535

[9] Marco D'Antonio, Kåre von Geijer, Thai Son Mai, Philippas Tsigas, and Hans Vandierendonck. 2025. Relax and Don't Stop: Graph-aware Asynchronous SSSP. In *Proceedings of the 1st FastCode Programming Challenge (FCPC '25)*. Association for Computing Machinery, New York, NY, USA, 43–47. doi:10.1145/3711708.3723446

[10] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 74. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science. http://www.dis.uniroma1.it/~challenge9/

[11] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 293–304. doi:10.1145/3087556.3087580

[12] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Transactions on Parallel Computing* 8, 1 (April 2021), 4:1–4:70. doi:10.1145/3434393

[13] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. doi:10.1007/BF01386390

[14] 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. doi:10.1145/3409964.3461782

[15] Paul Erdős and Alfréd Rényi. 1960. On the Evolution of Random Graphs. *Publ. math. inst. hung. acad. sci* 5, 1 (1960), 17–60.

[16] R. Guimerà, S. Mossa, A. Turtschi, and L. A. N. Amaral. 2005. The Worldwide Air Transportation Network: Anomalous Centrality, Community Structure, and Cities' Global Roles. *Proceedings of the National Academy of Sciences* 102, 22 (May 2005), 7794–7799. doi:10.1073/pnas.0407994102

[17] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2009. Optimized Routing for Large-Scale InfiniBand Networks. In *2009 17th IEEE Symposium on High Performance Interconnects*. 103–111. doi:10.1109/HOTI.2009.9

[18] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What Is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. Association for Computing Machinery, New York, NY, USA, 591–600. doi:10.1145/1772690.1772751

[19] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. 2005. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *Knowledge Discovery in Databases: PKDD 2005 (Lecture Notes in Computer Science)*, Alípio Mário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama (Eds.). Springer, Berlin, Heidelberg, 133–145. doi:10.1007/11564126_17

[20] U. Meyer and P. Sanders. 2003. Δ-Stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms* 49, 1 (Oct. 2003), 114–152. doi:10.1016/S0196-6774(03)00076-2

[21] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 456–471. doi:10.1145/2517349.2522739

[22] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 12–25. doi:10.1145/1993498.1993501

[23] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. 2022. Multi-Queues Can Be State-of-the-Art Priority Schedulers. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 353–367. doi:10.1145/3503221.3508432 arXiv:2109.00657 [cs]

[24] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15)*. Association for Computing Machinery, New York, NY, USA, 80–82. doi:10.1145/2755573.2755616

[25] Robert E. Tarjan and Uzi Vishkin. 1985. An Efficient Parallel Biconnectivity Algorithm. *SIAM J. Comput.* 14, 4 (Nov. 1985), 862–874. doi:10.1137/0214061

[26] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. doi:10.1145/79173.79181

[27] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity Based on Faster Reachability. *Proc. ACM Manag. Data* 1, 2 (June 2023), 114:1–114:29. doi:10.1145/3589259

[28] Marvin Williams, Peter Sanders, and Roman Dementiev. 2021. Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ESA.2021.81*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2021.81

[29] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. 2015. The Lock-Free k-LSM Relaxed Priority Queue. *ACM SIGPLAN Notices* 50, 8 (Jan. 2015), 277–278. doi:10.1145/2858788.2688547

[30] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics (MDS '12)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/2350190.2350193

[31] F. Benjamin Zhan and Charles E. Noon. 1998. Shortest Path Algorithms: An Evaluation Using Real Road Networks. *Transportation Science* 32, 1 (Feb. 1998), 65–73. doi:10.1287/trsc.32.1.65

[32] Guozheng Zhang, Gilead Posluns, and Mark C. Jeffrey. 2024. Multi Bucket Queues: Efficient Concurrent Priority Scheduling. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 113–124. doi:10.1145/3626183.3659962

[33] 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. doi:10.1145/3368826.3377909