Relax and don't Stop: Graph-aware Asynchronous SSSP

Citation for the original published paper (version of record):
D'antonio, M., von Geijer, K., Mai, T. et al (2025). Relax and don't Stop: Graph-aware Asynchronous SSSP. Proceedings of the 1st Fastcode Programming Challenge Fcpc 2025: 43-47.
http://dx.doi.org/10.1145/3711708.3723446

Relax and don't Stop: Graph-aware Asynchronous SSSP

Marco D'Antonio*
Queen's University Belfast
Belfast, United Kingdom

Kåre von Geijer*
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden

Thai Son Mai
Queen's University Belfast
Belfast, United Kingdom

Philippas Tsigas
Chalmers University of Technology
and University of Gothenburg
Gothenburg, Sweden

Hans Vandierendonck
Queen's University Belfast
Belfast, United Kingdom

Abstract

The Parallel Single-Source Shortest Path (SSSP) problem has been tackled through many implementations, yet no single approach consistently outperforms others across diverse graph structures. Moreover, most implementations require extensive parameter tuning to reach peak performance. In this paper, we introduce the AdaMW scheduler, which dynamically selects between the schedulers Wasp and MultiQueue. To achieve this, we use graph sampling and heuristics to select and configure the scheduler. In contrast to common state-of-the-art bulk-synchronous implementations, AdaMW is fully asynchronous, thus not needing to stop for global barriers. The resulting scheduler is highly competitive with the best manually-tuned, state-of-the-art implementations.

CCS Concepts: • **Theory of computation** → **Shortest paths**; • **Computing methodologies** → **Shared memory algorithms**; **Concurrent algorithms**.

Keywords: Shortest Path, Shared-memory, Asynchrony, Priority Scheduler

ACM Reference Format:
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 *FastCode Programming Challenge (FCPC '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3711708.3723446

1 Introduction

Single-Source Shortest Path (SSSP) is a fundamental problem with plenty of applications [6, 13, 14, 22] and well-known solutions in the sequential setting [5, 10, 12]. However, in a parallel context, achieving scalable performance across the large variety of existing graphs is still an open challenge with no clear winner [3]. For this reason, one of the FastCode Programming Challenge (FCPC) tracks at PPoPP'25 was to efficiently parallelize SSSP on a shared-memory machine across a variety of both small-diameter and large-diameter graphs [21]. Shared-memory implementations for SSSP have recently garnered increased interest, as modern machines can now completely fit large graphs in memory [9].

A common parallel implementation is Δ-stepping [15], which groups vertices in buckets based on their coarsened distance $dist[u]/\Delta$, and processes each bucket in parallel in a bulk-synchronous fashion. Compared to the traditional Dijkstra's algorithm [10], this approach increases parallelism but also introduces redundant work due to vertices being processed out of the work-efficient order.

Although Δ-stepping is an efficient parallel approach, its bulk-synchronous nature imposes significant synchronization overhead. This issue is particularly pronounced in large-diameter graphs, such as road networks, where buckets often contain few vertices, leading to frequent global barriers and performance degradation [24]. To address these limitations, alternative approaches relax the strict synchronization constraints while maintaining efficient parallel execution.

One such approach is Wasp [7], which builds upon coarsening of Δ-stepping but relaxes its synchronization by allowing threads to process buckets asynchronously. The asynchronous Δ-coarsening enables Wasp to perform well across most graphs, and especially ones with uniform edge weights, because the number of items is balanced across Δ-buckets. Another approach is the MultiQueue [17, 20], a relaxed priority queue that allows highly parallel operations at the expense of a degree of processing disorder. The MultiQueue exhibits lower peak throughput, but performs well on large-diameter graphs which typically expose less parallelism. Moreover, as the MultiQueue does not use Δ-coarsening, it is more resilient than Wasp to skewed edge weights, which generate more redundant work when using bucketing.

Combining the strengths of Wasp [7] and the MultiQueue [20], we introduce the fused ***Adaptive MultiQueue-Wasp*** (AdaMW) scheduler, which dynamically selects one of the two schedulers based on the target graph. By analyzing the average

*Both authors contributed equally to this work.

ACM ISBN 979-8-4007-1446-7/2025/03
https://doi.org/10.1145/3711708.3723446

Table 1. Graph datasets used in the experimental evaluation. $|V|$ is the number of vertices, $|E|$ is the number of undirected edges. SD and LD in Properties stand for "Small Diameter" and "Large Diameter". SR stands for "Skewed Random edge weights", UR for "Uniform Random edge weights", RE for "Real Edge weights".

Abbr.	Graph	$\|V\|$	$\|E\|$	Properties
LJ	LiveJournal	4.7 M	41.9 M	SD - UR
HW	Hollywood	1.0 M	55.1 M	SD - SR
ER	ER	9.89 M	490 M	SD - UR
EW	enwiki-2023	6.5 M	147.1 M	SD - SR
GR	Grid-1k-10k	9.89 M	19.6 M	LD - UR
GL	Geolife	24.6 M	77.1 M	LD - RE
NA	North America	86.0 M	107.9 M	LD - RE
SA	South America	21.8 M	29.1 M	LD - RE

degree and a sample of the edge weights, it both selects the appropriate scheduler and configures its parameters to enable suitable optimizations. AdaMW was the fastest SSSP solver at FCPC 2025 [21], reaching a geometric average performance of 150.9 *MEdges/s* across their selected graphs, while in comparison, the second-best performing solution achieved a geometric average of 52.6 *MEdges/s*.

In this paper, we introduce AdaMW and demonstrate its great performance across a variety of graphs while requiring minimal preprocessing for selecting and configuring the scheduler. Preprocessing also enables us to selectively apply optimizations based on the graph structure. Finally, our experimental evaluation shows AdaMW's efficiency compared to Wasp [7], Δ^*-stepping [11], and the MultiQueue [20], each configured with optimal Δ or *stickiness* per graph. The results show that AdaMW can achieve competitive or better performance than the state-of-the-art over different graph characteristics, without offline tuning of parameters.

Outline. Section 2 gives a background to the problem and utilized schedulers. Section 3 highlights some related work for parallel SSSP. Section 4 describes AdaMW. Section 5 contains our evaluation, comparing AdaMW to Wasp, the MultiQueue, and Δ^*-stepping. Section 6 concludes the paper.

2 Background

Given an undirected weighted graph $G = (V, E, w)$, with vertices $v \in V$, edges $(u, v) \in E \subseteq V \times V$, and weights $w : E \to \mathbb{R}^+$, as well as a source $s \in V$, the SSSP problem is to find the shortest path from s to every other reachable $u \in V$. Our algorithm is allowed some time for graph initialization and minor preprocessing before it is given s and should minimize the time to compute a correct shortest-path distance mapping $d : V \to \mathbb{R}^+$ for all vertices. The graphs used in the evaluation are shown in Table 1 and comprise small and large diameter graphs, with varied edge weight distributions.

2.1 The Wasp Priority Scheduler

Wasp [7] is an asynchronous priority scheduler with a distributed bucketing structure and priority-aware work stealing. When a thread processes an item, it is pushed into a thread-local bucket. This alleviates the need for synchronization between threads to manage the buckets. Threads share work through a priority-aware work-stealing protocol. Stealing is performed only if the items to be stolen have a higher priority than locally available items. Otherwise, asynchrony allows the thread to process local items with no barriers.

Wasp utilizes priority coarsening to discretize the number of priority levels, defining a parameter Δ. Therefore, for an efficient Δ-stepping implementation, Δ must be tuned to each graph.

2.2 The Relaxed MultiQueue

The base MultiQueue [17] by Rihani et al. is a concurrent relaxed priority queue, which keeps $c \cdot T$ lock-protected sequential priority queues (the sub-queues), where c is a constant tuning parameter and T is the number of threads. When an item is inserted in the MultiQueue, it is uniformly at random inserted into one of the sub-queues. To dequeue an item, the operation randomly samples the highest priority item from two sub-queues and then proceeds by dequeuing from the sampled sub-queue with the highest priority item. The *rank error* of a dequeue is defined as the number of items currently in the queue which has a higher priority than the dequeued item, and there are theoretical guarantees that it is rather low and stable [1, 2, 20].

3 Related Work

The Δ^*-stepping and ρ-stepping algorithms are built around a stepping, bulk-synchronous algorithm framework that abstracts Δ-stepping [11]. A Lazy-Batched Priority Queue supports the framework for managing vertex distances. The queue supports updates to record new distances and extraction of vertices with keys below a given threshold. Extracted vertices are processed in parallel, and relaxed neighbors have their distance updated in the queue. The framework supports both push and pull operations for SSSP, allowing the processing of dense frontiers efficiently [4].

The simple MultiQueue [17, 20] core has also led to the Stealing MultiQueue [16], which integrates thread-affine sub-queues and work stealing, as well as the Multi Bucket Queue [23], which uses Δ-coarsening to speed up its sub-queues. Additionally, a strong potential argument [1, 2] as well as a Markov chain analysis [19] give strong guarantees on the rank errors in the MultiQueue.

4 The AdaMW Scheduler

Building on top of Wasp [7] and the MultiQueue [20], we develop AdaMW to get the best of both worlds. By utilizing the allowed preprocessing period in FCPC, AdaMW analyses

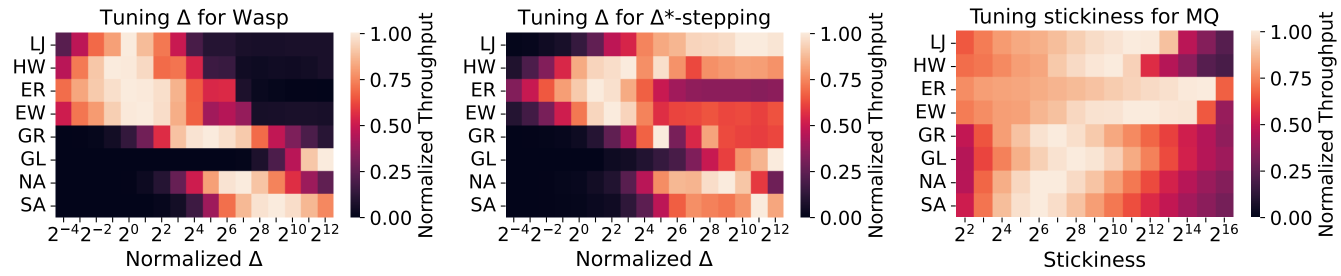

Figure 1. Performance of different Δ and *stickiness* for the schedulers. The throughput of each graph is normalized by the largest observed throughput. Δ is normalized by $\frac{M(W)}{\overline{d}}$, where $M(W)$ is the sampled median weight and $\overline{d}$ the average degree.

the graph structure, selecting and configuring the preferred scheduler for that graph type.

Our experiments show that Wasp is best for graphs with uniform weight distributions, while the MultiQueue is best for large-diameter graphs with skewed weight distributions. To characterize the graph structure, AdaMW samples a set of edge weights W. In the competition, we used $|W| = 100,000$. It then estimates whether the weights are skewed or not by looking at the ratio $\frac{\overline{W}}{M(W)}$, where $\overline{W}$ is the average and $M(W)$ is the median of W. Furthermore, as small-diameter graphs usually have a large average degree $\overline{d}$, AdaMW selects the underlying scheduler as follows:

$$Sched = \begin{cases} Wasp, & \text{if } \overline{d} > 8 \text{ or } \frac{\overline{W}}{M(W)} < 2 \\ MQ, & \text{otherwise} \end{cases}$$

4.1 Estimating Δ in Wasp-mode

The Δ-coarsening used in Wasp requires the Δ parameter to be carefully tuned in order to balance the trade-off between parallelism and redundant work. Since increasing Δ allows for an increase in parallelism, a first-stage decision can be made based on the graph structure. Graphs with a large average degree $\overline{d}$ usually need smaller values of Δ, as vertices have more neighbors and, therefore, expose more parallelism. Low-degree graphs, instead, need a larger Δ to increase parallelism and keep all threads busy. Therefore, we initially set $\Delta = 1$ for small-diameter graphs, while for large-diameter graphs, we start with $\Delta = T$, the number of threads. Then, these values are scaled by the graph-specific ratio $\frac{M(W)}{\overline{d}}$ that accounts for the weights and the degrees of the graph, obtaining the final formula:

$$\Delta = \begin{cases} \frac{M(W)}{\overline{d}}, & \text{if } \overline{d} > 8 \\ \frac{M(W)}{\overline{d}}T, & \text{otherwise} \end{cases}$$

4.2 Configuring MultiQueue-mode

We use an optimized MultiQueue implementation by Williams et al. [20] that utilizes 8-ary heaps as sub-queues and insertion and deletion buffers to improve cache locality. Furthermore, they implement *stickiness*: threads return to the same sub-queues several operations in a row, improving cache performance at the expense of priority order [18]. We set the number of sub-queues to twice the number of threads used, as it has previously proven a good balance [17, 20]. Additionally, we set the buffer size to 16 after trying a few different options, seeing that it consistently performed slightly better than the others. Configuring stickiness is the hardest part, as we found that the optimal choice, much like Δ, varied for different graphs. However, it is a bit easier to configure than Δ, as it does not depend on the size of the weights, but rather the degree distribution. In the competition, AdaMW used *stickiness* = 128 for all sparse graphs with good results.

4.3 SSSP Optimizations

AdaMW uses two SSSP-specific optimizations. These are part of the base Wasp [7], but not of the MultiQueue [20]. The first optimization is to add bidirectional relaxation [11]. When processing a vertex, before relaxing outgoing edges, we first relax all incoming edges to possibly decrease the tentative distance of the processed vertex. This optimization is always enabled for high-diameter graphs, while for small-diameter graphs, it is only used if the neighborhood fits into an L1 cache line.

Secondly, AdaMW does not add degree-1 vertices back into the scheduler when relaxing their incoming edge. These vertices are leaves of the SSSP tree; hence, relaxing their outgoing edge will not find a lower distance to any vertex. All leaf vertices are precomputed and tracked in a bit-set. AdaMW only uses this optimization when more than 10% of vertices are leaves, as constantly querying the bit-set becomes too costly otherwise.

5 Evaluation

We evaluate AdaMW, Wasp [7], the MultiQueue[1] [20], and Δ*-stepping[2] [11] – a representative from state-of-the-art – on the graphs from FCPC, presented in Table 1. All experiments are run on a dual-socket AMD EPYC 7713 processor with 64 cores per socket, simultaneous multi-threading disabled, 1TB DRAM, and 4 NUMA nodes per socket. The implementations are compiled using gcc-14.1.0 and C++20

[1]https://github.com/marvinwilliams/multiqueue
[2]https://github.com/ucrparlay/Parallel-SSSP

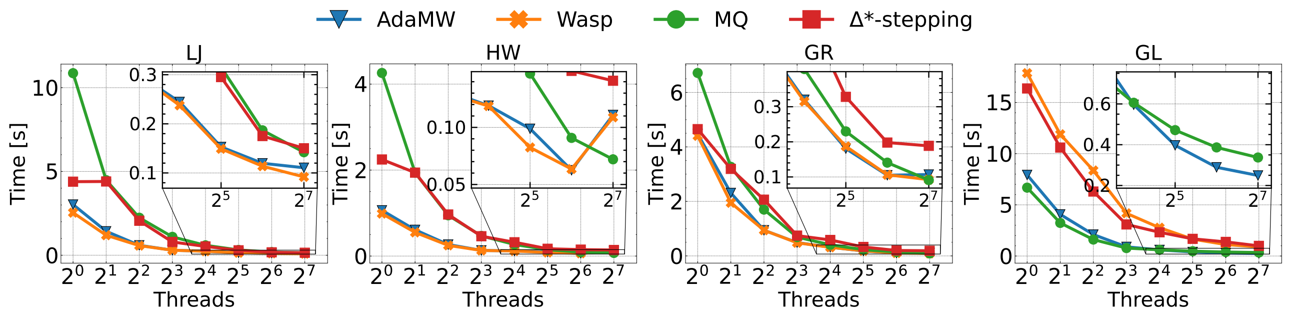

Figure 2. Scalability analysis of the different schedulers. Two graphs per typology are shown for space reasons.

Table 2. Precomputation time for scheduler selection and parameter tuning per graph. **% SSSP** is the percentage relative to the average SSSP execution time. The last column shows the average and geometric mean across all graphs.

	LJ	HW	ER	EW	GR	GL	NA	SA	**mean**
Time [s]	0.014	0.012	0.012	0.012	0.013	0.016	0.067	0.025	0.021
% SSSP	12.441	10.833	3.043	7.903	12.092	6.363	12.024	11.740	8.776

with the `-O3` and `-march=native` compilation options. All experiments used `numactl -i all` to interleave the memory allocations across NUMA nodes.

Since Wasp and Δ*-stepping use Δ-coarsening, in our evaluation we tune Δ by sampling powers of 2, as common in literature [8, 11]. For the MultiQueue, we tune the stickiness value, while the other parameters are the same as used in AdaMW. We tune Δ and the stickiness for each thread configuration in the evaluation.

To highlight the difficulty of selecting the correct configuration parameters, Figure 1 shows their impact on the different schedulers at 128 threads. The preprocessing time for AdaMW is not considered in these plots. The optimal configuration choice varies greatly between graphs. Especially Δ is rather sensitive to bad configurations while configuring the stickiness is slightly more forgiving. Interestingly, the optimal choice of Δ is distinctly different between Wasp and Δ*-stepping, even though serving the same purpose in both schedulers. One can verify that AdaMW's choice of *stickiness* = 2^7 for the MultiQueue is suitable for the sparse graphs. Conversely, AdaMW's use of normalized Δ = 1 for small-diameter graphs and Δ = T for large-diameter graphs can also be verified. An exception is the Geolife graph, which has skewed weights; AdaMW chooses the MultiQueue scheduler for this graph.

Table 2 reports the preprocessing time AdaMW requires to configure its schedulers. The weight sampling and leaves optimization bound this time to $O(|V| + |W|)$, and in FCPC, AdaMW currently utilizes the allowed preprocessing budget. However, this time can be further reduced, for example, by sampling a smaller W. This highlights that one can avoid tuning Δ for each graph with low impact on performance.

Figure 2 shows the strong scaling performance of the implementations. AdaMW performs well across all thread counts, despite only using heuristics for its configurations, unlike the others, which use the optimal choice. The throughput of Wasp is reduced while using both sockets for the Hollywood graph, letting the MultiQueue overtake it at 128 threads. This behavior is reproducible, but we could not determine why it only happens for the Hollywood graph.

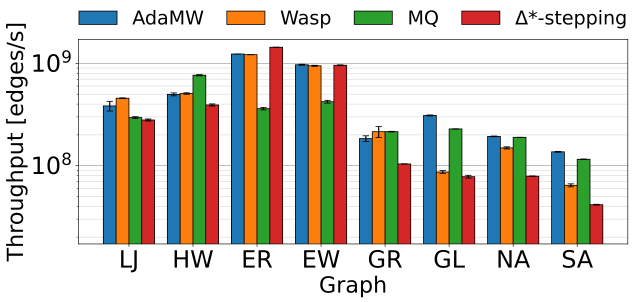

Figure 3. Scheduler throughput with 128 threads.

Finally, the performance of the schedulers at scale is shown in Figure 3. For all evaluated graphs, AdaMW is always close to the optimal. On the GeoLife, North America, and South America graphs, it is even better than all optimally tuned schedulers, thanks to the optimizations from Section 4.3.

6 Conclusion

AdaMW combines the strengths of the asynchronous Wasp and MultiQueue schedulers, dynamically adapting to different graph structures to maximize performance. Moreover, it demonstrates that a graph sampling heuristic can be used to tune critical parameters, achieving performance on par with the optimal scheduler configuration. This approach addresses a critical challenge in many schedulers — balancing adaptability and efficiency without requiring exhaustive parameter tuning.

Acknowledgments

This work was partially funded by the European Union, Horizon Europe 2021-2027 Framework Programme, Grant No. 101072456, the UK Research and Innovation, Engineering and Physical Sciences Research Council, Grant No. EP/X029174/1, and the Swedish Research Council, Grant No. 2021-05443. Views or opinions are those of the authors.

References

[1] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z. Li, and Giorgi Nadiradze. 2018. Distributionally Linearizable Data Structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 133–142. doi:10.1145/3210377.3210411

[2] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. 2017. The Power of Choice in Priority Scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. Association for Computing Machinery, New York, NY, USA, 283–292. doi:10.1145/3087801.3087810

[3] Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Maia P. Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, Tim Davis, Kevin Deweese, Jesun Firoz, Henry A Gabb, Gurbinder Gill, Balint Hegyi, Scott Kolodziej, Tze Meng Low, Andrew Lumsdaine, Tugsbayasgalan Manlaibaatar, Timothy G Mattson, Scott McMillan, Ramesh Peri, Keshav Pingali, Upasana Sridhar, Gabor Szarnyas, Yunming Zhang, and Yongzhe Zhang. 2020. Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 216–227. doi:10.1109/IISWC50251.2020.00029

[4] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-Optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10. doi:10.1109/SC.2012.50

[5] Richard Bellman. 1958. On a Routing Problem. *Quart. Appl. Math.* 16, 1 (1958), 87–90. doi:10.1090/qam/102435

[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] Marco D'Antonio, Son T. Mai, and Hans Vandierendonck. 2025. Toward Efficient Asynchronous Single-Source Shortest Path . In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Los Alamitos, CA, USA.

[8] 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

[9] 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

[10] 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

[11] 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

[12] L. R. Ford. 1956. *Network Flow Theory*. Technical Report. RAND Corporation. https://www.rand.org/pubs/papers/P923.html

[13] 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

[14] 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

[15] 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

[16] 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* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 353–367. doi:10.1145/3503221.3508432

[17] 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

[18] Adones Rukundo, Aras Atalar, and Philippas Tsigas. 2019. Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework. In *33rd International Symposium on Distributed Computing (DISC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 146)*, Jukka Suomela (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:15. doi:10.4230/LIPIcs.DISC.2019.31

[19] Stefan Walzer and Marvin Williams. 2024. A Simple yet Exact Analysis of the MultiQueue. arXiv:2410.08714

[20] Marvin Williams, Peter Sanders, and Roman Dementiev. 2021. Engineering MultiQueues: Fast Relaxed Concurrent Priority Queues. In *29th Annual European Symposium on Algorithms*, Vol. 204. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 81:1–81:17. doi:10.4230/LIPIcs.ESA.2021.81

[21] Yihan Sun and Tim Kaler. 2025. FastCode Programming Challenge. https://fastcode.org/events/fastcode-challenge/ Accessed: 2025-01-30.

[22] 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

[23] 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

[24] 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