Wasp: Efficient Asynchronous Single-Source Shortest Path on Multicore Systems via Work Stealing

RESEARCH-ARTICLE

Wasp: Efficient Asynchronous Single-Source Shortest Path on Multicore Systems via Work Stealing

MARCO D'ANTONIO, Queen's University Belfast, Belfast, Northern Ireland, U.K.

THAI SON MAI, Queen's University Belfast, Belfast, Northern Ireland, U.K.

PHILIPPAS TSIGAS, Chalmers University of Technology, Gothenburg, Vastra Gotaland, Sweden

H. VANDIERENDONCK, Queen's University Belfast, Belfast, Northern Ireland, U.K.

Published: 16 November 2025

Citation in BibTeX format

SC '25: The International Conference for High Performance Computing, Networking, Storage and Analysis
November 16 - 21, 2025
MO, St. Louis, USA

Conference Sponsors:
SIGHPC

SC '25: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (November 2025)
https://doi.org/10.1145/3712285.3759872
ISBN: 9798400714665

Wasp: Efficient Asynchronous Single-Source Shortest Path on Multicore Systems via Work Stealing

Marco D'Antonio
Queen's University Belfast
Belfast, United Kingdom
mdantonio01@qub.ac.uk

Thai Son Mai
Queen's University Belfast
Belfast, United Kingdom
thaison.mai@qub.ac.uk

Philippas Tsigas
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden
philippas.tsigas@chalmers.se

Hans Vandierendonck
Queen's University Belfast
Belfast, United Kingdom
h.vandierendonck@qub.ac.uk

Abstract

The Single-Source Shortest Path (SSSP) problem is a fundamental graph problem with an extensive set of real-world applications. State-of-the-art parallel algorithms for SSSP, such as the Δ-stepping algorithm, create parallelism through priority coarsening. Priority coarsening results in redundant computations that diminish the benefits of parallelization and limit parallel scalability.

This paper introduces Wasp, a novel SSSP algorithm that reduces parallelism-induced redundant work by utilizing asynchrony and an efficient priority-aware work stealing scheme. Contrary to previous work, Wasp introduces redundant computations only when threads have no high-priority work locally available to execute. This is achieved by a novel priority-aware work stealing mechanism that controls the inefficiencies of indiscriminate priority coarsening.

Experimental evaluation shows competitive or better performance compared to GAP, GBBS, MultiQueues, Galois, Δ*-stepping, and ρ-stepping on 13 diverse graphs with geometric mean speedups of 2.23× on AMD Zen 3 and 2.16× on Intel Sapphire Rapids using 128 threads.

CCS Concepts

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

Keywords

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

ACM Reference Format:
Marco D'Antonio, Thai Son Mai, Philippas Tsigas, and Hans Vandierendonck. 2025. Wasp: Efficient Asynchronous Single-Source Shortest Path on Multicore Systems via Work Stealing. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3712285.3759872

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/25/11
https://doi.org/10.1145/3712285.3759872

1 Introduction

Single-Source Shortest Path (SSSP) is an important problem that arises in various domains, including routing [35], network analysis [33, 65], and calculating the betweenness centrality of a network [13]. To keep up with the ever-increasing size of graph datasets, new techniques and optimizations are developed to process graphs faster on multicore [29, 55, 58, 67], distributed [18, 22, 41, 42] and heterogeneous systems [19, 44, 49, 59]. This paper focuses on multicore shared-memory systems that, through compression techniques, accommodate most publicly available real-world graphs [29].

SSSP is usually solved using either Δ-stepping [45] or parallel Dijkstra's algorithm [30], which is based on concurrent priority queues [52, 53, 62]. 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 [45]. 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, 52, 53, 63]. 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.

This paper proposes a novel algorithm, *Work-Stealing Shortest Path* (WSSP→/wdsp/→**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. The core of the algorithm resides in a novel work-stealing protocol that accounts for the thief's and the victim's currently available vertices and their priorities, issuing a steal operation only when advantageous. This is a fundamental difference from traditional work-stealing protocols [8, 9] as the priority-based mechanism enables more efficient scheduling choices. Moreover, the protocol accounts for the NUMA hierarchy of modern multicore processors and favors stealing from topologically closer threads. Experimental evaluation of Wasp using two different machine configurations shows consistent improvements over the Δ-stepping based implementations of GAP [2, 5], GBBS [29], and Galois [39, 48] by respectively 1.72×, 3.42×, and 1.94× using geometric mean (gmean). Additionally, Wasp outperforms the parallel Dijkstra-based solution using the relaxed MultiQueue [53, 62], with a 2.74× gmean speedup, and the state-of-the-art parallel algorithms for SSSP, Δ*-stepping and ρ-stepping [31] by respectively gmean 1.66× and 2.15×.

In summary, the contributions of this paper are:

- We identify the shortcomings of state-of-the-art parallel SSSP algorithms, analyzing and identifying when to usefully introduce priority drifting.
- We design a novel parallel SSSP algorithm that enables on-demand priority relaxation to increase available parallelism and thread occupancy.
- We design a novel, lightweight, and efficient NUMA-aware work-stealing protocol for asynchronously retrieving high-priority work in parallel SSSP.
- We provide an experimental evaluation against six state-of-the-art parallel SSSP algorithms on 13 heterogeneous graph datasets on two modern multicore machines.

The rest of the paper is organized as follows. Section 2 presents the background on the parallel SSSP problem, including algorithms and implementations. Section 3 illustrates the main drawbacks in state-of-the-art parallel SSSP algorithms and opportunities for judicious priority relaxation. Section 4 presents the Wasp algorithm's design, the stealing protocol's design, and the optimizations performed. The experimental methodology, the results, and their discussion are presented in Section 5. Finally, Section 6 discusses related work, and Section 7 concludes the paper.

2 Background

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 $w : E \rightarrow \mathbb{R}_{\geq 0}$, and a source vertex $s \in V$, the Single-Source Shortest Path (SSSP) problem is the problem of finding the shortest path from s to every other vertex in the graph.

The fundamental process in SSSP algorithms is edge relaxation[1]. Given the tentative shortest-path distance of a vertex u from the source s, $\boldsymbol{d(u)}$, the relaxation given an edge $e = (u, v)$ checks if $d(u)+w(e) < d(v)$, and if so updates the distance $d(v)$ to the smaller value. **Dijkstra's algorithm** [30] has the best-known bound of $O(|E| + |V| \log |V|)$ time complexity using a priority queue [14, 56], greedily relaxing 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 throughout the paper. 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, with theoretical bounds on k [1, 52, 53, 63].

The other common approach to parallel SSSP is **Δ-stepping** [45]. It maintains the frontier of active vertices in a set of buckets. Bucket i stores vertices u such that $d(u) \in [i \cdot \Delta, (i + 1) \cdot \Delta)$, 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, relaxing each of the vertices in it. The Δ-coarsening of priorities creates **priority drifting**: mapping vertices with different distances into the same bucket leads to relaxations that are out of the work-efficient order of Dijkstra's algorithm. Any of these additional relaxations are considered redundant work, which 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, choosing the right Δ 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 [57], 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** [5] 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 [67], 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** [28] 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.
- **Δ*-stepping** and **ρ-stepping** [31] process vertices with a distance up to a certain threshold at each step. The two algorithms differ in how the threshold is calculated. The Lazy-Batched Priority Queue is introduced to support this framework and is implemented as a parallel hash-bag to extract and update vertices [61].

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. Two state-of-the-art designs are:

[1]The term is used to refer to both **edge relaxation**, which arises in SSSP computations, and **priority relaxation**, a broader concept in which lower-priority work is executed prior to or concurrently with higher-priority work in priority-driven algorithms.

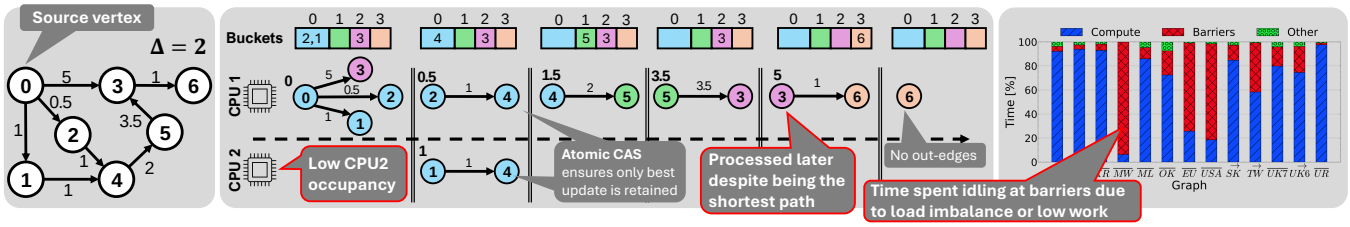


Figure 1: A sample graph (left) and the execution of Δ-stepping [45] on it (center). The main drawbacks of the method are highlighted in red. We show the execution breakdown on the real graph datasets of our experimental evaluation using the GAP [5] implementation (right).

- The **MultiQueue** [53] 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.
- **Galois**' [48, 51] SSSP uses the OBIM [39] scheduler and implements asynchronous Δ-stepping. Vertices are first pushed to thread-local bags, while excess vertices go into global bags. Threads work on the highest-priority local bag and then synchronize with the global bag to find higher-priority work.

3 Motivation

The main drawback of the **synchronous approach** is the limitation of parallelism. Figure 1 illustrates a sample execution of Δ-stepping. Starting from the source vertex, at each step, edges are relaxed, and updated vertices are added to the buckets. From CPU2's low occupancy, we can see that when work is not available for all workers, parallel execution is not effective, and the overhead of barriers weighs in. However, bucket 3 has work available since step 1, but this is not accessible because buckets are processed in order. One way to change that is to increase Δ, but this comes at the cost of more priority drifting. Finally, synchronization can be more costly if the workloads are unbalanced across threads, e.g. in skewed-degree graphs, threads processing high-degree vertices take longer to complete and can hold back other threads from progressing. In Figure 1 (right), we show the barrier overhead of the GAP implementation [5] on the graphs used in our experimental evaluation in Section 5. The largest barrier overheads are for road graphs with low average degree ($\overline{EU}$ and $\overline{USA}$), and for some skewed-degree graphs ($\overrightarrow{TW}$ and $\overline{MW}$). In Δ-stepping, the degree of coarsening must be tuned to make a sufficient number of parallel tasks available at each priority level. This ensures that the amount of work is larger than the overhead incurred by barrier synchronization. If, however, a too small Δ is applied, then the barrier overhead will increase drastically. It is unfortunate that the parameter that minimizes the synchronization overhead is also the parameter that controls the degree of priority drifting. As such, one is **forced to choose between one of the two sources of inefficiency**. Ideally, one would want to use Δ-coarsening only when little parallelism is available and otherwise use a very small Δ to reduce priority coarsening and, therefore, redundant work.

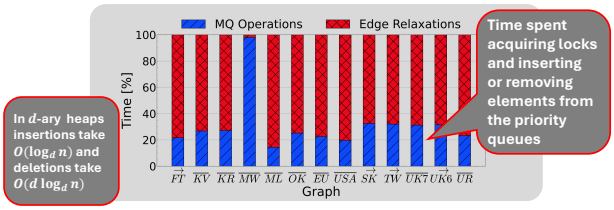


Figure 2: Execution breakdown of parallel Dijkstra's algorithm using the MultiQueue with d-ary heaps. In the experiments, $d = 8$.

The main challenge of the **asynchronous approach** is keeping the priority queue consistent while threads concurrently access it, as this will limit priority drifting. One approach is to split the storage of the queue into a shared and a thread-local part, so that accessing the local portion does not require synchronization [39, 52, 63]. However, this involves a trade-off between redundant work and synchronization: the less work is shared, the more redundant work occurs, but with reduced synchronization costs. The other approach is a centralized design [1, 53, 66]. The MultiQueue [53], for instance, uses cp lock-protected priority queues that are shared across threads, where c is a constant and p is the number of threads. Therefore, to access the queues, threads must contend for a lock. The execution time of a priority queue-based 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**. On the positive side, the MultiQueue is **very flexible in extracting parallelism across priority levels**. Due to maintaining priority queues, it can process lower-priority vertices asynchronously as soon as all vertices with the highest priority have been extracted.

The analyses show that both the synchronous Δ-stepping and asynchronous parallel Dijkstra's algorithms have multifactorial performance bottlenecks. The strong aspects of synchronous execution relate to the efficiency of processing large numbers of vertices in bulk without intermittent synchronization operations. The strong aspects of the asynchronous Dijkstra's algorithm relate to fine-grain priority drifting, which occurs only when all higher-priority work has been dispatched. A new solution is required that combines these strengths without inheriting the limitations of these schemes.

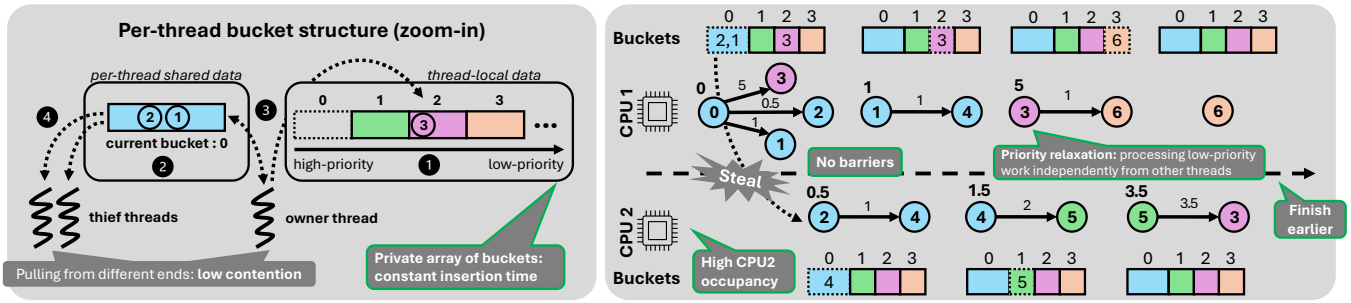


Figure 3: The distributed bucketing structure of Wasp (left) and the execution of the Wasp algorithms on the sample graph of Figure 1 (right). Each CPU's current bucket is depicted with a dashed line in the algorithm execution.

4 Design of Wasp

We design **Wasp**, a parallel SSSP algorithm that reduces thread idling and the synchronization overhead introduced by the data structures while at the same time limiting priority drifting. We achieve this through three key design components.

- A distributed bucketing structure that leverages data structures with constant-time sequential access and strong locality, and that enables lock-free work stealing.
- An asynchronous execution model that allows threads to process work without barriers, unlocking parallelism and facilitating access to useful lower-priority vertices.
- A NUMA-aware priority-based work stealing protocol that favors stealing high-priority vertices and load balances the work by controlling the asynchrony and priority drifting.

4.1 Overview

Wasp organizes vertices in buckets based on their distance from the source, like Δ-stepping does. However, the computation is not organized in steps. Instead, threads make progress independently and do not wait for other threads to complete a priority level. Buckets are distributed: each thread works on its own buckets but can share work through work-stealing. In Figure 3, we show the execution of the Wasp algorithm and its **per-thread bucketing structure**, which is composed of:

- A list of **thread-local buckets**, one for each coarsened priority level ❶, i.e. bucket indices.
- A special, work-stealing enabled **shared** bucket ❷, called **current bucket**, that holds the vertices belonging to the current priority level being processed by a thread.

The thread-local bucketing structure avoids the synchronization overheads of managing buckets in parallel and guarantees constant-time insertion and deletion operations, while the shared current bucket enables workload balancing through work stealing. Wasp groups vertices into **chunks**: fixed-size ring buffers of vertices. Therefore, internally, both the current bucket and the thread-local buckets store chunks of vertices. In particular, the current bucket makes a chunk available for stealing only once the chunk is full.

Algorithm 1 shows the simplified, high-level pseudo-code of the Wasp asynchronous algorithm, starting at Line 16. A vertex, its priority level, and the range of neighbors to be processed are pulled from the current bucket (Line 19). We discuss the details of how the range of neighbors is calculated in Section 4.4. Redundant relaxations are avoided by checking if the vertex is stale, i.e. a better path has been concurrently found already (Line 20). If the vertex is not stale, the range of outgoing edges is relaxed (Lines 12-15). The relaxation process (Lines 1-8) tries to update the distance of the

Algorithm 1: The Wasp algorithm.

```
1  def relax(u, v):
2      new_dist ← distance[u] + w(u, v)
3      old_dist ← distance[v]
4      while new_dist < old_dist do
5          if CAS(distance[v], old_dist, new_dist) then
6              return new_dist
7          new_dist ← distance[u] + w(u, v)
8      return void
9  def push_to_buckets(v, prio):
10     if prio = curr then push v to current bucket
11     else push v to buckets[prio]
12 def process_neighborhood(u, range):
13     for v in out-neighbors[u][range.begin : range.end] do
14         if dist ← relax(u, v) and v is not a leaf then
15             push_to_buckets(v, ⌊dist/Δ⌋)
16 def work_stealing_shortest_path():
17     in parallel: while termination is not detected do
18         while current bucket is not empty do
19             u, prio, range ← pop from current bucket
20             if distance[u] ≥ Δ · prio then
21                 process_neighborhood(u, range)
22         stolen chunks ← work_stealing()
23         curr ← arg min_prio {stolen chunks}
24         for each chunk of stolen chunks do
25             while chunk is not empty do
26                 u, prio, range ← pop from chunk
27                 if distance[u] ≥ Δ · prio then
28                     process_neighborhood(u, range)
29         if current bucket is empty then
30             curr ← arg min_prio {non-empty buckets[prio]}
31             if curr is not ∞ then
32                 pour buckets[curr] into current bucket
33     return distance
```

destination vertices using atomic Compare-and-Swap (CAS). If the destination vertex is updated, it is pushed into the buckets (Line 15). If the vertex's priority level matches the current bucket's priority level, it is directly pushed to the current bucket. Otherwise, it is pushed into the thread-local bucket list ❸. To discriminate between the two cases, a *curr* variable is kept for each thread, storing the priority level tracked by the current bucket.

Once the current bucket of a thread is empty, the thread tries to steal chunks of higher-priority vertices (Line 22) from other threads' current buckets ❹ before processing local, lower-priority vertices. The work-stealing protocol is described in the next section. The *curr* variable is updated by tracking the priority level of the highest-priority chunk (Line 23) to make sure updated vertices are pushed into the correct bucket. Stolen vertices are immediately processed (Lines 24-28). Therefore, once stolen, chunks cannot be stolen by other threads. Notice that no barriers are present, and threads can progress independently, e.g. if a thread finishes processing its current bucket, it can start stealing chunks in a lock-free manner while other threads are still working on their current bucket.

After processing the stolen chunks, the current bucket will contain those vertices whose distance has been updated to a value within the current bucket, i.e. in the interval $[curr \cdot \Delta, (curr+1) \cdot \Delta)$. The thread will continue processing these vertices as this is the highest-priority work available. If no vertices have been inserted in the current bucket during the processing of the stolen chunks, the thread finds the next priority level to work on in the thread-local bucket list, moves the chunks in the current bucket, and continue working (Line 29-32). If all thread-local buckets are empty, then a thread will either continuously try to steal until it finds work or possibly terminate. Again, threads can independently process the next available bucket without synchronizing with other threads. This is the opposite of synchronous schedulers, where threads might idle to wait for all others before advancing to the next bucket. In this regard, Wasp allows more parallelism while being able to retrieve high-priority work due to its efficient work-stealing scheme.

4.2 Work-Stealing Protocol

Wasp uses work-stealing to distribute work across the per-thread bucketing structures. Work-stealing is initiated before processing the lower-priority vertices of the thread-local buckets to check if higher-priority work is available for processing in the system. However, work-stealing does not progress to the solution if all other threads have lower-priority vertices than the locally available ones. Therefore, it is advantageous to steal vertices only if they have a priority higher than the next local bucket's priority level. The traditional work-stealing approach is to randomly select a victim and steal any available work [9]. This does not fit our requirements, as it would fail to find high-priority work among threads. Hence, our stealing protocol inspects multiple victims and only steals higher-priority work. Choosing the number of victims is also important. While the Wasp algorithm works well if work stealing is attempted from all threads, restricting the number of victims can improve performance. For this reason, we group the victim threads in tiers, based on their NUMA distance from the thief. Thieves on different NUMA nodes will have a different view of these tiers. Then, work stealing is performed first on closer threads, and potentially on further ones. Notice that, regardless of which NUMA node the data structures reside on, accessing NUMA-local caches is more efficient than accessing remote ones.

Algorithm 2: The work-stealing protocol of Wasp.

```
1 def work_stealing():
2     next ← arg min_prio{non-empty buckets[prio]};
3     for all NUMA tiers i do
4         for all threads t in NUMA tier i do
5             if thread t's priority level curr ≤ next then
6                 steal a chunk from thread t's current bucket;
7         if has stolen something then
8             return stolen chunks;
9     return none;
```

The stealing protocol is shown in Algorithm 2. Threads first find the non-empty local bucket with the highest priority level (Line 2) through a scan of the bucket list. If all the thread-local buckets are empty, *next* in Algorithm 2 will be equal to ∞. Then, the other threads' current priority level is inspected, looking at the *curr* variable presented in Algorithm 1. The protocol progresses through the NUMA tiers from closest to furthest based on NUMA distance (Line 3). The protocol tries to steal from all threads in the NUMA tier that are working on higher-priority vertices. If any higher-priority chunks are stolen at any tier, the protocol does not steal further and returns the stolen chunks; otherwise, it progresses to the next tier.

The introduction of priorities alters the traditional work-stealing context: after processing a bucket, a thread must choose between stealing or processing the thread-local buckets. We evaluated a traditional random-victim stealing protocol with different numbers of retry attempts; we report geometric mean values across graphs: random work stealing was 50% (no-retry) to 36% (up to 64-retries) slower than Wasp's stealing protocol. Moreover, we tested a MultiQueue-like protocol, where two victims are chosen randomly and the stealing is performed on the highest-priority bucket, obtaining a 39% (no-retry) to 27% (up to 64-retries) slowdowns over our stealing protocol. Therefore, these approaches failed to find high-priority work, motivating the need for a priority-based scheme.

4.3 Implementation Details

Batching. Vertices in the Wasp algorithm are grouped in chunks. Their size is chosen at compilation time; in this paper, we use a chunk size of 64 vertices. Chunks are implemented as ring buffers with a `next` field that allows the creation of lists of chunks, and a `priority` field that records the priority level to which they correspond. Chunks can represent sets of vertices or the partial neighborhood of a single vertex. To support the latter mode, chunks have a `begin` and `end` field to represent the neighborhood range. All ring buffer operations have constant complexity and exhibit locality.

Thread-local Buckets. A single bucket is implemented as a linked list of chunks and is managed as a stack. Each thread has an unbounded vector of buckets, i.e. the thread-local buckets. Whenever a vertex is pushed into a bucket that has not been created yet, the vector is resized, rounding the size to the next power of two larger

than the bucket priority level. This avoids multiple resizes when vertices are inserted in subsequent buckets.

Current Bucket. The *curr* variable is a per-thread, shared variable that stores the priority level of the current bucket. It is initialized to zero and can change non-monotonically during execution. To allow work stealing, the current bucket is implemented as a Chase-Lev work-stealing deque that stores pointers to chunks [17]. The deque uses a ring buffer and two monotonic 64-bit integer indices for the *top* and *bottom* ends of the deque. Thief threads pop elements from the top index, while the owner of the deque pushes and pops elements from the bottom index. Contention between thieves and owners only happens when a single element is present. The deque is lock-free, and contention for an element is resolved through atomic CAS instructions. When the underlying ring buffer is full, the deque resizes by doubling its size and copying the elements. Notice that resizing is only triggered by the owner pushing a new element and does not affect steal attempts that can still be performed. The current bucket uses a thread-local chunk to buffer both pushes and pops of vertices. Once the chunk is full, it is pushed to the deque. Chunks popped from the deque are reused as thread-local chunks. Previous works use a similar approach, but with two chunks, one for pushing and one for popping vertices; in some cases, the chunks will have different sizes [62, 66]. In our case, experimental data show that a single chunk works better. Since the buckets and the current bucket are two different types of structures, i.e., list vs. array, advancing to the next bucket requires moving the chunks from the next bucket to the current bucket. This operation is shown in Algorithm 1 at Line 32. It is a linear-time operation that scans the bucket and copies the pointer to the chunks into the deque.

Termination Detection. Finally, we illustrate the termination detection protocol used in Algorithm 1 at Line 17. Termination should happen only when all threads have finished working. This means that the current bucket and all the thread-local buckets are empty. When a thread has finished working, it sets its current bucket's priority level to ∞. During the termination detection routine, the thread will check the priority level of every other thread's current bucket. The thread will stop if the priority level is ∞ for every thread. Otherwise, it will re-enter the work loop and try to steal work from other threads. Note that, since the thread's priority level is ∞, the stealing procedure will steal vertices at any priority level.

4.4 Optimizations

Neighborhood Decomposition. High-degree vertices in skewed-degree graphs cause load balancing issues because the neighborhood of a vertex is normally processed by a single thread. In synchronous implementations, it is possible to introduce load-balanced parallelism when the frontier of active vertices is very large, i.e., a large fraction of the edges will be visited in that synchronous step. This is achieved using a pull (or backward) step that only updates the shortest distance for all non-converged vertices and is highly effective when the path distance of a large number of vertices has converged [4]. This optimization does not apply in asynchronous implementations. Instead, Wasp introduces parallelism in the processing of high-degree vertices by decomposing their neighborhood into a number of contiguous intervals of neighbors. We decompose a neighborhood only if the degree is larger than a threshold θ, and each interval consists of a fixed-size range of θ neighbors, except for the last one. The decomposition is performed when a high-degree vertex is popped from a chunk (Line 19 in Algorithm 1). A thread processes the first range, while the other ranges are pushed into the buckets as single-vertex chunks with non-empty `begin` and `end` fields. We found $\theta = 2^{20}$ to be a large enough threshold to balance the trade-off between creating chunks and sharing the neighborhood processing.

Pruning Leaf Vertices. We implement an optimization to avoid useless work caused by the trivial leaves of the shortest path tree. These vertices are characterized by an in-degree equal to one and by not having any out-edge, except the one connecting them to the in-edge source. The shortest distance to leaf vertices does not impact the shortest distance to any other vertex. As such, it suffices to relax leaf vertices just once, namely after finalizing the distance to their parent in the shortest path tree. This optimization precomputes whether a vertex is a leaf in the shortest path tree and stores the value in a bitmap. After updating a vertex's distance, we only push the vertex into the buckets if it is not a leaf (Line 14 in Algorithm 1). Precomputing a bitmap proves more effective than on-the-fly checks, which cause more cache misses due to the random access pattern involved in retrieving the degree of an out-neighbor.

Bidirectional Relaxation. We implement the bidirectional relaxation optimization for undirected graphs [31]. Before relaxing a vertex u's out-neighbors (push-step), we try to update u's tentative distance through its in-neighbors (pull-step) to get a more effective relaxation later. Since in- and out-neighbor lists are the same in undirected graphs, this does not incur further cache misses. We limited this optimization to neighborhoods that fit into an L1 cache line, i.e., ≤ 8 weighted vertices, as this performed better in our case.

5 Experimental Evaluation

Our evaluation is performed on two machines that we refer to as Epyc and Xeon. Epyc uses a dual-socket AMD Zen 3 EPYC 7713 processor with 64 cores per socket, with simultaneous multi-threading disabled, 1TB DRAM, and 4 NUMA nodes per socket. Xeon uses a dual-socket Intel Sapphire Rapids Xeon Gold 6438Y+ processor with 32 cores per socket, with hyper-threading enabled, 256GB DRAM, and two sub-NUMA nodes per socket. Therefore, experiments on both systems use up to 128 threads. We compare Wasp against six baselines: the SSSP implementations from the GAP Benchmarking Suite [5], GBBS [29], and Galois [48]; Δ^*-stepping and ρ-stepping [31]; and parallel Dijkstra's algorithm using the MultiQueue [53]. The codebases are compiled using gcc-14.1.0 and gcc-10.2.1 respectively on Epyc and Xeon, using C++17 and the `-O3` and `-march=native` compilation options. We launch each execution of the programs using `numactl -i all` to interleave the memory allocations on all NUMA nodes.

Datasets. 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. The Road-USA, Twitter, sk-2005, Kron, and Urand graphs are the versions used

Table 1: Graph datasets used in the experimental evaluation. An overline on the graph abbreviation means the graph is undirected, an arrow means the graph is directed. $|V|$ is the number of vertices, $|E|$ is the number of directed edges – every edge is counted twice in undirected graphs.

Abbr.	Graph	$\|V\|$	$\|E\|$	Graph Type
$\overrightarrow{FT}$	Friendster [64]	68.3 M	2.58 B	Social Network
$\overline{KV}$	Kmer-v1r	214. M	465.4 M	Biological Network
$\overline{KR}$	Kron [40]	134.2 M	4.22 B	Synthetic Graph
$\overline{MW}$	Mawi	226.2 M	480 M	Network Traffic
$\overline{ML}$	Moliere	30.2 M	6.67 B	Semantic Network
$\overline{OK}$	Orkut [64]	3.1 M	234.4 M	Social Network
$\overline{EU}$	Road-EU [3]	54.1 M	108.1 M	Road Network
$\overline{USA}$	Road-USA [27]	23.9 M	57.7 M	Road Network
$\overrightarrow{SK}$	sk-2005	50.6 M	1.93 B	Web Crawl
$\overrightarrow{TW}$	Twitter [38]	61.6 M	1.46 B	Social Network
$\overline{UK7}$	uk-2007 [3]	104.3 M	6.6 B	Web Crawl
$\overrightarrow{UK6}$	uk-union-06	133.6 M	5.47 B	Web Crawl
$\overline{UR}$	Urand [32]	134.2 M	4.29 B	Synthetic Graph

in the GAP benchmarking suite [2, 5]. Friendster and uk-2007 are from the KONECT network repository [37]; Orkut, Road-EU, Kmer-v1r, Mawi, Moliere, and uk-union-06 are available in the SuiteSparse Matrix Collection [26] and at Laboratory for Web Algorithmics [10–12]. We use the Mawi graph `201512020330` from SuiteSparse. Road-USA, Mawi, and Moliere[2] are the only graphs with natural edge weights. All the other graphs have weights generated according to the GAP Benchmarking Suite and its reference implementation, i.e., uniformly distributed integers from 1 to 255.

Baselines Configuration. To help reproduce our results, we provide the baselines' configuration details where needed. We use the latest GAP version, which implements bucket fusion in SSSP [67]. GBBS uses the Julienne bucketing approach described in Section 2. We compare against the Δ-stepping implementation of GBBS using the default number of buckets, 32. GBBS is compiled using the ParlayLib scheduler for parallelism [6]. We use an optimized MultiQueue implementation that uses 8-ary heaps for priority queues, and *stickiness*, i.e., s consecutive pop operations will be performed on the same queue [62]. The implementation also uses insertion and deletion buffers of size b to increase locality, although accessing them still requires locks. The configuration of the MultiQueues in our experiments is $c = 2$, and $b = 16$, while we tuned the stickiness s for each input graph. Galois groups vertices in chunks, with the chunk size being a tuning parameter. The chunk size significantly impacts performance; we chose a single chunk size of 128 vertices for all graphs to provide a fair comparison.

Methodology. The measurement methodology is based on the GAP Benchmarking Suite [5]. Multiple trials are run for each graph. To reduce the variance of results, all trials have the same randomly selected starting vertex residing in the largest connected component. The Wasp codebase is based on the GAP reference implementation. Therefore, vertex identifiers and edge weights are 32-bit integers.

[2]The original edge weights were floats. However, not all implementations supported float weights. For this reason, we scaled the weights by a factor of 10^8 (with minimal loss of edge weights) and converted them to integers.

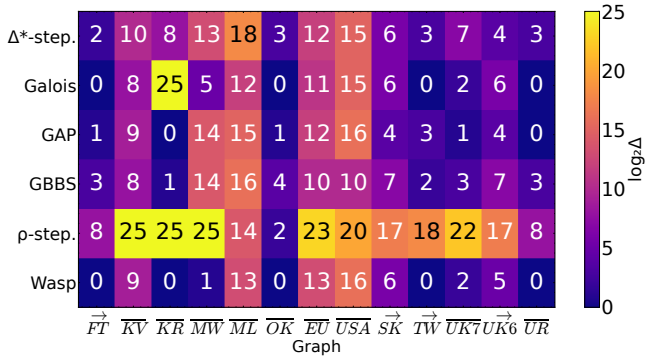


Figure 4: Optimal values of Δ used in the experimental evaluation with 128 threads for each graph and implementation.

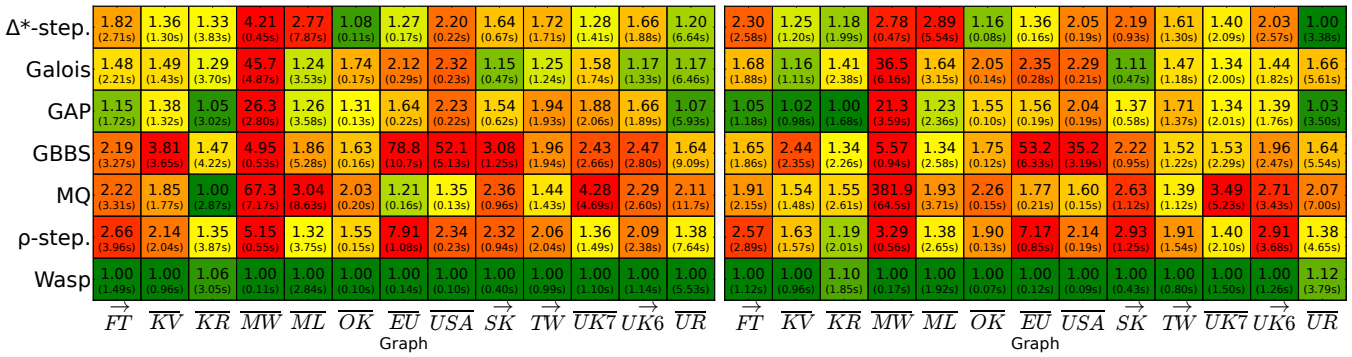
Tuning the Δ parameter in Δ-stepping appropriately is critical for good performance, as has been repeatedly observed [16, 31, 34]. We estimate the optimal Δ for each framework by sampling the space of possible choices using powers of two. The best values for Δ, which are used in the experimental evaluation with 128 threads, are listed in Figure 4. Notice that Wasp prefers low values of delta for 9 of the 13 graphs, with exceptions only for the low-degree graphs and the Moliere graph. This is an important result in itself, as it makes it much easier to estimate a good value for Δ. In general, it is impossible to know what a good Δ value is without executing the algorithm, as algorithms to estimate Δ are not accurate and also time-consuming [45]. This limits the practical applicability and constrains the performance benefits of Δ-stepping. 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 not the case for the examined baselines, where $\Delta = 1$ can be a very poor choice. We will explain later what causes this effect. The MultiQueue needs no tuning of Δ as it uses the parallel Dijkstra's algorithm (but still needs tuning of the stickiness parameter).

5.1 Performance Results

Figure 5 shows the color-coded performance of Wasp and the baselines we compare against. We show both execution time in seconds and the speedup of the best-performing implementation for each graph on both the Epyc and Xeon machines. Wasp is faster overall than the baselines, with just two cases in which it performs worse (≥ 10% difference in performance). In Table 2, we show the speedup of Wasp over each of the baselines. For these aggregated metrics, we use the geometric mean.

Road networks. Wasp excels in processing road graphs. Thanks to the asynchrony, it does not incur any barrier overhead, which makes Wasp have more than a 30-fold speedup over GBBS. It is faster than implementations that use optimizations to reduce the barrier overhead, outperforming the bucket fusion used in GAP, and the super sparse rounds of Δ*- and ρ-stepping. Work stealing is crucial to achieving this performance. It allows Wasp to process lower-priority vertices only when no high-priority vertices are available and, therefore, take advantage of the absence of barriers without incurring too much priority drift.

Skewed-degree graphs. Performance is generally good on skewed-degree graphs, although varied. Wasp generally outperforms the



(a) Performance of SSSP on Epyc with 128 cores and SMT disabled. **(b) Performance of SSSP on Xeon with 64 cores and SMT enabled.**

Figure 5: Heatmap of the performance of Wasp and the examined implementations on Epyc and Xeon using 128 threads. For each column, the best-performing implementation has a speedup of 1.0×. The other rows in the column show the speedup of the best-performing implementation over the corresponding implementation. Absolute execution times in seconds are shown under the speedup.

MultiQueue and GBBS on both systems. Contrary to expectations, ρ-stepping (with best ρ) is less performant than Δ^*-stepping [31], while Wasp has either better or competitive performance than both of them. Performance with GAP is often close; in particular, we see a higher performance improvement for those graphs that have a higher percentage of barrier time (see Figure 1). This suggests that Wasp compensates for the idle time at barriers by performing useful work that allows the algorithm to converge faster. Finally, Wasp outperforms Galois in most cases with at least 15% better performance. We would point out, however, that Galois reaches better performance thanks to the additional tuning of the chunk size, with larger chunks favoring skewed-degree graphs, with a difference in speedup of about 30% over the default chunk size. Conversely, the chunk size does not significantly impact Wasp's performance, making it easier to tune.

Mawi. The Mawi graph has a highly exceptional structure, leading to exceptional performance results. The graph contains a high-degree vertex connected to 93% of the vertices. However, 99% of these edges connect to degree-1 vertices, creating a star-like structure in the graph. Wasp is always the fastest when processing the Mawi graph, with large speedups over the other implementations. Thanks to the neighborhood decomposition and leaves pruning optimizations presented in Section 4.4, Wasp seamlessly distributes the large neighborhood across threads and avoids rescheduling leaf vertices. Galois, GAP, and the MultiQueue fail in parallelizing the workload as a single thread processes the whole neighborhood, while GBBS, Δ^*-stepping, and ρ-stepping exhibit better performance thanks to a direction-optimization pull-step [4]. Wasp outperforms Galois, GAP, and the MultiQueue by at least 20×, and up to 381×, while it has a 4× gmean speedup over the pull-enabled implementations across both systems.

5.2 Performance Scaling

We complement the results with 128 threads with a comparison of the scaling performance of Wasp and the baselines on the Epyc system. The number of threads is an additional parameter to take into account when selecting the value of Δ. In fact, the availability of fewer parallel resources usually calls for smaller values of Δ, as guaranteeing priority ordering is more important than high parallelism. Therefore, we tune Δ for each configuration of threads, graph, and implementation. Figure 6 shows our strong scaling analysis on four graphs. We show the execution time and speedup of the implementations with varying numbers of threads, from 1 to 128. We chose the MultiQueue as the baseline for these speedup figures, which allows us to compare the scaling performance of different implementations. For space reasons, we limit the number of figures to four graphs that we find representative of all results.

Table 2: Geometric mean speedup of Wasp over the baselines on Epyc and Xeon, across all graph datasets. Aggregated speedups across implementation and machine are shown in the last column and row.

	Δ*-step.	Galois	GAP	GBBS	MQ	ρ-step.	gmean
Epyc	1.67×	1.9×	1.83×	3.86×	2.54×	2.19×	2.23×
Xeon	1.66×	2.0×	1.61×	3.03×	2.94×	2.11×	2.16×
gmean	1.66×	1.94×	1.72×	3.42×	2.74×	2.15×	2.2×

Table 3: Self-speedup of each implementation with respect to the 1-thread execution time on Epyc. For each graph, the underlined number denotes the highest self-speedup.

Graph	Δ*-step.	Galois	GAP	GBBS	MQ	ρ-step.	Wasp
$\overrightarrow{FT}$	81.4	42.7	33.2	26.3	36.6	<u>145.6</u>	73.9
$\overline{KV}$	88.9	91.3	35.3	29.4	95.8	61.3	<u>100.3</u>
$\overline{KR}$	24.8	44.2	27.6	25.7	52.3	25.5	<u>52.4</u>
$\overline{MW}$	<u>74.2</u>	1.1	1.5	18.6	4.5	71.5	24.5
$\overline{ML}$	30.0	38.3	29.0	23.8	23.2	35.9	<u>72.3</u>
$\overline{OK}$	25.0	16.9	13.5	22.5	<u>38.3</u>	26.8	27.1
$\overline{EU}$	28.7	22.2	20.9	0.7	<u>73.9</u>	6.3	39.0
$\overline{USA}$	12.0	12.3	9.3	0.7	<u>44.3</u>	12.7	25.2
$\overrightarrow{SK}$	53.3	52.8	31.1	26.3	52	50.9	<u>67.9</u>
$\overrightarrow{TW}$	49.6	39.8	15.8	22.5	45.6	50.7	<u>51.3</u>
$\overline{UK7}$	57.5	44.0	26.0	25.7	37.1	53.2	<u>74.6</u>
$\overrightarrow{UK6}$	54.6	45.7	30.1	29.4	54.2	57.5	<u>83.6</u>
$\overline{UR}$	21.2	35.0	23.4	22.0	23.4	<u>121.8</u>	48.9

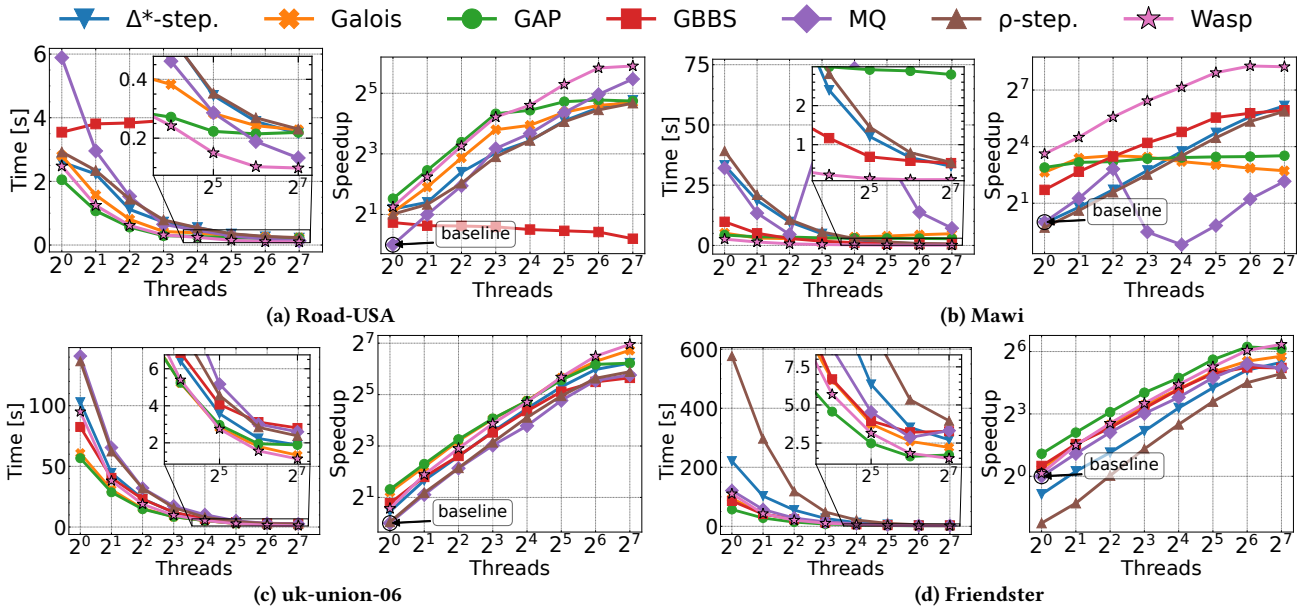


Figure 6: Strong scaling analysis of the examined implementations on the Epyc system. For each graph, the execution time and speedup of each implementation are shown. The speedups are relative to the execution of the MultiQueue with one thread.

In general, with a few threads, Wasp starts slow. However, as soon as the number of threads is more significant, e.g., ≥ 16 threads, Wasp catches up with GAP or surpasses it. This is expected because Wasp's overhead for managing concurrent data structures is spread across threads as the workload is increasingly distributed among parallel threads. Moreover, Wasp usually scales better than the other implementations. Figure 6a shows how Wasp continues scaling after 16 threads on the Road-USA graphs, whereas GAP stops scaling at 32 threads. We also see that GBBS does not scale well on road graphs, being slower than the 1-thread execution. This is not an error and aligns with previous results [31]. In Figure 6b, the optimizations of Wasp give it a large advantage over all other implementations on the Mawi graph, showing optimal scaling. Figure 6d and 6c represent the scaling on skewed-degree graphs. Wasp scales well and is usually better than the other implementations. On some of these graphs, the scaling curve of GAP flattens or performs worse when reaching 128 threads, but Wasp is consistently faster or on par with GAP with both 64 and 128 threads.

Finally, in Table 3, we present the self-speedup of each implementation. In general, we notice that no implementation has the best self-speedup on all graphs. Wasp has good self-speedup, although it is not always the best. We note that some abnormal values for ρ-stepping are due to the very slow 1-thread execution time, as evidenced by Friendster in Figure 6d. The same happens for the Urand graph.

5.3 Optimizations Ablation Study

It is common practice to introduce additional optimizations on top of a key algorithmic idea to improve performance in practice on specific types of datasets. For instance, direction optimization (used in Δ^*-stepping, ρ-stepping, and GBBS) and bucket fusion (used in Δ^*-stepping, ρ-stepping, and GAP) are used to deal with, respectively, skewed-degree and large-diameter graphs. Figure 7 shows

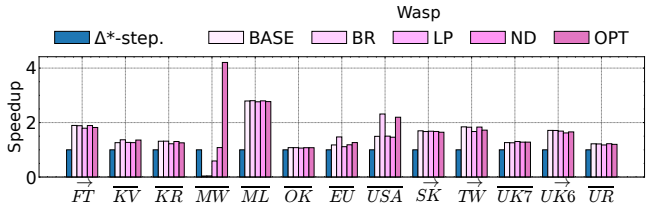
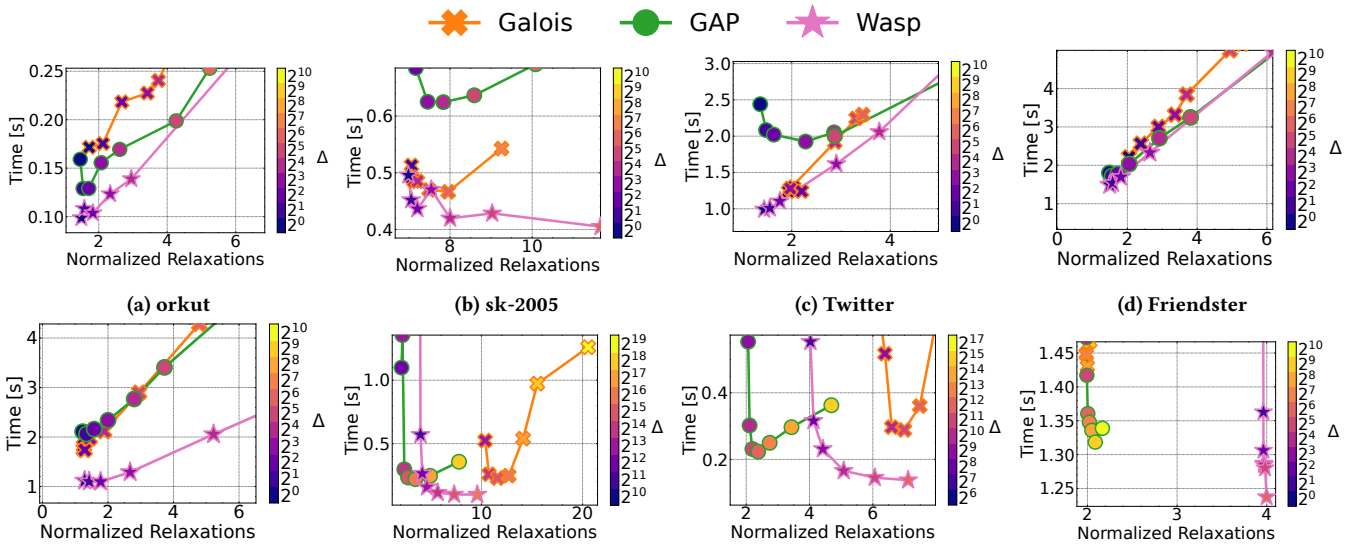


Figure 7: Ablation study of the optimizations used in Wasp. Δ^*-stepping is the baseline. BASE is the version without optimizations, BR uses bidirectional relaxation only, LP uses leaves pruning only, ND uses neighborhood decomposition only, and OPT is the version with all optimizations enabled.

the impact of each single optimization presented in Section 4.4 over the base implementation of Wasp's priority-aware work stealing, without any optimization. Moreover, we also compare to the best-performing baseline implementation, Δ^*-stepping. With only one exception, the non-optimized version of Wasp outperforms Δ^*-stepping, with a 14% performance improvement. We note that this is an unfair comparison, as the baseline has all optimizations enabled. In general, we see that not all graphs benefit from all optimizations, e.g., bidirectional relaxation increases performance in road networks, while neighborhood decomposition is more useful for denser graphs with large neighborhoods. The leaves pruning introduces some overhead for checking whether a vertex is a leaf; however, we notice that this could be avoided through a fast preprocessing step on the graph [21]. Finally, enabling all optimizations gives the best performance overall across the graphs, and in particular for the Mawi graph, in which neighborhood decompositions and leaves pruning are crucial to increase parallelism and reduce useless work.



(a) orkut (b) sk-2005 (c) Twitter (d) Friendster

(e) uk-2007 (f) Road-USA (g) Road-EU (h) Kmer-v1r

Figure 8: Comparison of the number of edge relaxations and the execution time of Galois, GAP, and Wasp for different values of Δ. The number of edge relaxations is normalized by the number of relaxations in Dijkstra's algorithm.

5.4 Priority Drift Analysis

The value of Δ determines both parallelism and priority drifting in Δ-stepping. We demonstrate that Wasp achieves a decoupling of these two aspects, which explains its high performance. Figure 8 shows the impact of Δ on the execution time and the number of relaxations. The number of relaxations is normalized to those performed by Dijkstra's algorithm, which is the theoretical minimum number of relaxations. Results are shown for eight graphs. These are representative of the remaining graphs.

In this analysis, we compare Wasp against GAP and Galois, as the former represents the typical behavior of synchronous Δ-stepping, while the latter is the principal asynchronous competitor of Wasp. For each implementation, we show the top seven values of Δ. Not all data points are shown due to scale reasons.

We observe that increasing the value of Δ generally increases the amount of relaxations, as expected when coarsening priorities. In general, across all graphs, Galois has a higher number of relaxations than Wasp with equal Δ, showing slower performance. GAP, on the other hand, is more conservative with the number of relaxations. However, it often needs to increase Δ not only in low-degree graphs – which is expected due to the low parallelism – but also in skewed-degree graphs, where more work is generally available. This happens more prominently on Orkut, sk-2005 and Twitter, as shown in Figures 8a, 8b and 8c.

Wasp shows a behavior that is a mix of the two. On skewed-degree graphs in Figures 8a to 8e, Wasp achieves the minimal amount of relaxations with $\Delta = 1$. Moreover, increasing Δ increases both the number of relaxations and execution time, with the only exception for sk-2005. Minimal Δ and relaxations allow Wasp to perform generally better than the other implementations, as shown in Figure 5. This analysis shows that Wasp efficiently exploits the inherent parallelism of skewed-degree graphs through its work-stealing-based load balancing mechanism.

The second behavior occurs in low-degree graphs. These are Road-USA, Road-EU, and Kmer-v1r, and we show them in Figures 8f to 8h. Small values of Δ do not work in these graphs: as they have a low average degree, little parallelism is available. However, coarsening causes priority drifting. We see this in both GAP and Galois. In Wasp, however, we do not see the same behavior of Galois because of our work-stealing strategy: stealing vertices based on global priorities dampens the impact of asynchrony. Although we cannot estimate a "good enough" value of Δ for every graph, the analysis shows that Wasp can better exploit the parallelism provided by Δ coarsening.

Therefore, Wasp can get the best of two worlds: it does not need priority coarsening when parallelism is already available, and it can exploit coarsening to create parallelism like GAP, but at the same time, overcome the overheads of the synchronous model by exploiting asynchrony using lightweight and priority-aware work stealing. Moreover, unlike GAP and Galois, Wasp allows priority drifting only when needed, as it happens only if no high-priority work is available. This way, Wasp decouples the generation of parallelism from priority drifting.

6 Related Work

Other than the approaches mentioned in Section 2, there is a vast literature on parallel methods for solving SSSP. KLA [34] presents a paradigm to relax synchronization, allowing up to k rounds to be executed asynchronously. Building on top of Δ-stepping, radius-stepping [7] provides work and depth guarantees for arbitrary undirected graphs. The bounds are first shown on (k, ρ)-graphs and then extended to arbitrary graphs by introducing shortcut edges to turn them into $(1, \rho)$-graphs. Chakaravarthy et al. hybridize Δ-stepping with Bellman-Ford, switching to it once the number of settled vertices reaches a local maximum after a bucket is processed [16]. Δ-stepping has been translated to an algebraic formulation [54] for use in the GraphBLAS framework [15, 24, 25, 36]. The Multi

Bucket Queue (MBQ) [66] uses a bucketing strategy adapted to the push/pop queue selection method of the MultiQueue to implement Δ-stepping. MBQ uses lock-protected queues, like the MultiQueue. Each queue uses a bucketing structure with a bounded range of buffers and an overflow and underflow bucket used to store tasks outside the current range. Using buckets reduces the sequential overhead of heaps, but the implementation still uses locking. In Wasp, we use a lock-free deque to allow work-stealing and balance the load among threads. DSMR [43] introduces a parallel algorithm for SSSP based on applying Dijkstra's algorithm on subgraphs of the starting graph. Other approaches for parallelizing Dijkstra's algorithms have also been proposed [20, 50]. The Stealing MultiQueue [52] is a relaxed priority queue that uses thread-local priority queues, implemented as d-ary heaps, and work stealing with batching. Each thread has a stealing buffer that is filled when local push or top heap operations occur. However, filling a stealing buffer of size b requires b pop-top operations from the heap, each with $O(d \log_d n)$ time complexity. Stealing is performed when a local heap is empty or with probability p when a pop-top is issued.

GPU-based SSSP. On GPUs, existing graph algorithms and optimization techniques must be adapted to the massively parallel nature of the architecture in order to utilize GPU resources efficiently. The Near-Far algorithm [23] adapts Δ-stepping by simplifying the bucket structure and using only the *Near* and *Far* buckets while keeping the synchronous execution model. ADDS [60] introduces a new GPU implementation that uses multiple buckets and operates asynchronously through a scheduler thread block that distributes work to worker thread blocks. Zhang et al. [68] optimize SSSP on GPU through a vertex reordering preprocessing step, adaptive load balancing that switches between static and dynamic work allocation to threads, and allows dynamic change of the buckets' size based on the GPU utilization and the number of converged vertices.

7 Conclusion

We have presented the Wasp algorithm for efficient parallel Single-Source Shortest Path computation. Wasp uses a thread-local bucket queue for low-priority vertices and work-stealing queues for the threads' highest-priority vertices. Its design is based on a novel priority-based and NUMA-aware work stealing protocol that enables load balancing across threads and judicious priority drifting triggered by the lack of higher-priority work.

Experimental performance evaluation on 13 graph datasets and two systems demonstrates 2-fold mean speedups over state-of-the-art parallel algorithms for SSSP, giving each implementation the benefit of custom tuned parameters. Wasp efficiently exploits the inherent parallelism of skewed-degree graphs, and contrary to prior work, selecting $\Delta = 1$ for Wasp does not result in major performance loss compared to the optimal Δ. This signifies major progress towards the practical applicability of our algorithm to a variety of graphs with minimal parameter tuning.

Acknowledgments

This work was partially funded by the European Union, Horizon Europe 2021-2027 Framework Programme, Grant No. 101072456, and the UK Research and Innovation, Engineering and Physical Sciences Research Council, Grant No. EP/X029174/1.

A Additional Experiments

The review committee proposed 14 additional graphs to further evaluate the Wasp algorithm. Our system could not generate/convert three due to memory limits, and two were too small for parallel SSSP, yielding similar performance to Dijkstra's algorithm. Several graphs had edge weights, but included negative values, making them unsuitable for positive-weight SSSP. Consequently, we assigned all weights using the reviewers' suggested scheme: a normal distribution with mean $\mu = 1$ and standard deviation $\sigma = \sqrt{|V|/|E|}$, truncated to exclude negatives.

Table 4 shows the list of additional datasets, while Figure 9 shows the performance of Wasp and the selected baselines on the Epyc system. Performance is varied on this dataset, and in this case, Wasp is not always the best-performing algorithm, with up to a 47% slowdown against other implementations. However, it is the implementation that performs best across all graphs, with a gmean 1.15× speedup against Δ*-stepping (lowest), a gmean 3.88× speedup against GBBS (highest), and an overall gmean 1.66× speedup across all implementations. These results raise a broader question about evaluating SSSP performance when edge weights are absent and must be generated. While we used the GAP Benchmarking Suite weighting scheme, others employ wider intervals [28, 31] or floating-point weights, as in Graph500 [46, 47]. Weight distribution also impacts results, with non-uniform distributions potentially altering conclusions. Further research is needed to assess how these factors influence benchmarking outcomes.

Table 4: Additional graph datasets.

Abbr.	Graph	$\|V\|$	$\|E\|$	Graph Type
$\overrightarrow{CR}$	Circuit5M	5.5 M	53.9 M	Circuit Sim.
$\overline{DL}$	Delaunay-n24	16 M	100.6 M	Delaunay Triangulation
$\overrightarrow{HC}$	Hypercube	8.3 M	192.9 M	Synthetic Graph
$\overline{KP}$	Kkt-power	2 M	12.9 M	KKT Graph
$\overline{NL}$	Nlpkkt240	17.9 M	746.4 M	KKT Graph
$\overrightarrow{RR}$	Random-regular	33 M	536.8 M	Synthetic Graph
$\overline{SM}$	Spielman-k600	72.1 M	144.7 M	Laplacian Matrix
$\overrightarrow{ST}$	Stokes	11 M	337.8 M	Semiconductor Sim.
$\overrightarrow{WB}$	Webbase-2001	113.1 M	979.7 M	Web Crawl

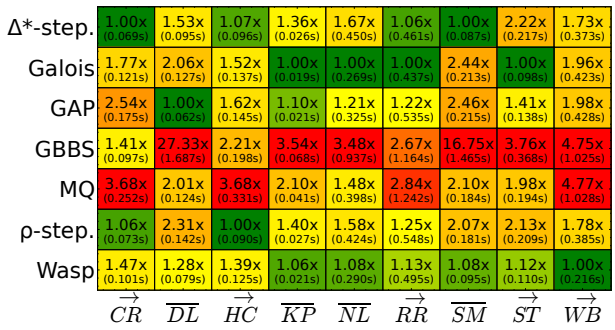


Figure 9: Performance heatmap of the baselines and Wasp on the additional graph datasets using the Epyc system.

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] 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

[3] 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

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

[6] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20)*. Association for Computing Machinery, New York, NY, USA, 507–509. doi:10.1145/3350755.3400254

[7] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel Shortest Paths Using Radius Stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. Association for Computing Machinery, New York, NY, USA, 443–454. doi:10.1145/2935764.2935765

[8] 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

[9] 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

[10] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice and Experience* 34, 8 (2004), 711–726. doi:10.1002/spe.587

[11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*. Association for Computing Machinery, New York, NY, USA, 587–596. doi:10.1145/1963405.1963488

[12] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. doi:10.1145/988672.988752

[13] 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

[14] 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.

[15] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. 2017. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 643–652. doi:10.1109/IPDPSW.2017.117

[16] 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

[17] David Chase and Yossi Lev. 2005. Dynamic Circular Work-Stealing Deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 21–28. doi:10.1145/1073970.1073974

[18] Fabio Checconi and Fabrizio Petrini. 2014. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 425–434. doi:10.1109/IPDPS.2014.52

[19] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluç, Katherine Yelick, and John D. Owens. 2022. Scalable Irregular Parallelism with GPUs: Getting CPUs Out of the Way. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–16. doi:10.1109/SC41404.2022.00055

[20] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. 1998. A Parallelization of Dijkstra's Shortest Path Algorithm. In *Mathematical Foundations of Computer Science 1998*, Luboš Brim, Jozef Gruska, and Jiří Zlatuška (Eds.). Springer, Berlin, Heidelberg, 722–731. doi:10.1007/BFb0055823

[21] 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

[22] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V. Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. 2019. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 15–28. doi:10.1109/PACT.2019.00010

[23] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 349–359. doi:10.1109/IPDPS.2014.45

[24] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4 (Dec. 2019), 44:1–44:25. doi:10.1145/3322125

[25] Timothy A. Davis. 2023. Algorithm 1037: SuiteSparse:GraphBLAS: Parallel Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 49, 3 (Sept. 2023), 28:1–28:30. doi:10.1145/3577195

[26] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (Dec. 2011), 1:1–1:25. doi:10.1145/2049662.2049663

[27] 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/

[28] 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

[29] 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

[30] 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

[31] 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

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

[33] 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

[34] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2014. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/2628071.2628091

[35] 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

[36] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical Foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–9. doi:10.1109/HPEC.2016.7761646

[37] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. doi:10.1145/2487788.2488173

[38] 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

[39] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority Queues Are Not Good Concurrent Priority Schedulers. In *Euro-Par 2015: Parallel Processing (Lecture Notes in Computer Science)*, Jesper Larsson Träff, Sascha Hunold, and

Francesco Versaci (Eds.). Springer, Berlin, Heidelberg, 209–221. doi:10.1007/978-3-662-48096-0_17

[40] 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

[41] Xue Li, Ke Meng, Lu Qin, Longbin Lai, Wenyuan Yu, Zhengping Qian, Xuemin Lin, and Jingren Zhou. 2023. Flash: A Framework for Programming Distributed Graph Processing Algorithms. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 232–244. doi:10.1109/ICDE55515.2023.00025

[42] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. 2019. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. IEEE Press, Dallas, Texas, 1–11. doi:10.1109/SC.2018.00059

[43] Saeed Maleki, Donald Nguyen, Andrew Lenharth, María Garzarán, David Padua, and Keshav Pingali. 2016. DSMR: A Parallel Algorithm for Single-Source Shortest Path Problem. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/2925426.2926287

[44] Ke Meng, Liang Geng, Xue Li, Qian Tao, Wenyuan Yu, and Jingren Zhou. 2023. Efficient Multi-GPU Graph Processing with Remote Work Stealing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 191–204. doi:10.1109/ICDE55515.2023.00022

[45] 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

[46] Richard Murphy, Jonathan Berry, William McLendon, Bruce Hendrickson, Douglas Gregor, and Andrew Lumsdaine. 2006. DFS: A Simple to Write Yet Difficult to Execute Benchmark. In *2006 IEEE International Symposium on Workload Characterization*. 175–177. doi:10.1109/IISWC.2006.302741

[47] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the Graph 500. *Cray Users Group (CUG)* 19, 45-74 (2010), 22.

[48] 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

[49] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. *ACM SIGPLAN Notices* 53, 2 (March 2018), 622–636. doi:10.1145/3296957.3173180

[50] James B. Orlin, Kamesh Madduri, K. Subramani, and M. Williamson. 2010. A Faster Algorithm for the Single Source Shortest Path Problem with Few Distinct Positive Lengths. *Journal of Discrete Algorithms* 8, 2 (June 2010), 189–198. doi:10.1016/j.jda.2009.03.001

[51] 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

[52] 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]

[53] 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

[54] 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)*. 241–250. doi:10.1109/IPDPSW.2019.00047

[55] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: Addressing Load Imbalance of Graph Partitioning. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3079079.3079097

[56] 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

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

[58] Hans Vandierendonck. 2020. Graptor: Efficient Pull and Push Style Vectorized Graph Processing. In *Proceedings of the 34th ACM International Conference on Supercomputing (ICS '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3392717.3392753

[59] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 38–52. doi:10.1145/3293883.3295733

[60] Kai Wang, Don Fussell, and Calvin Lin. 2021. A Fast Work-Efficient SSSP Algorithm for GPUs. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 133–146. doi:10.1145/3437801.3441605

[61] 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

[62] 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

[63] 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

[64] 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

[65] 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

[66] 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

[67] 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

[68] 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 (ICPP Workshops '23)*. Association for Computing Machinery, New York, NY, USA, 50–60. doi:10.1145/3605731.3605746

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 We identify the shortcomings of state-of-the-art parallel SSSP algorithms, analyzing and identifying when to usefully introduce priority drifting.

C_2 We design Wasp, a novel parallel SSSP algorithm that enables on-demand priority relaxation to increase available parallelism and thread occupancy.

C_3 We design a novel, lightweight, and efficient NUMA-aware work stealing protocol for asynchronously retrieving high-priority work in parallel SSSP.

C_4 We provide an experimental evaluation comparing against six state-of-the-art parallel SSSP algorithms on 13 heterogeneous graph datasets on two modern multicore machines.

A.2 Computational Artifacts

A_1 The artifact is available for download at this DOI: 10.5281/zenodo.15872863.

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3, C_4	Figures 1-2, and 4-7

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

The artifact provides the source code for the Work-Stealing Shortest Path (Wasp) algorithm and the work-stealing protocol (C_2, C_3). It provides the datasets and the source code of the implementations used in the experimental evaluation (C_1, C_4).

Expected Results

The first part of A_1 should show that the GAP Benchmarking Suite (GAP) has high barrier overheads (> 20% execution time) on at least six out of 13 graphs, and that the MultiQueue (MQ) has high queue operations overheads (> 20% execution time) on all graphs (C_1). The second part of A_1 should show the better performance results of Wasp when compared to other implementations with the best tuning parameters. The results are shown in terms of speedup and strong scaling on a single multicore server (C_2, C_3, C_4).

Expected Reproduction Time (in Minutes)

Expected artifact setup time:

- Compilation of source code of SSSP implementations and utilities: 10 minutes.
- Downloading graph datasets in Matrix Market/edge-list format: 300 minutes (with 50MB/s download speed).
- Converting the graph datasets into binary formats: 200 minutes.

Expected artifact execution time:

- Scaling experiments and tuning of each implementation's parameters: 36000 minutes (sequential time). Approximate parallel time, using five compute nodes: 7200 minutes.
- Execution breakdown of the GAP Benchmarking Suite and the MultiQueue on all graphs: 45 minutes (sequential time).
- Experiments to count the number of edge relaxations in Galois, GAP, and Wasp: 1500 minutes (sequential time).

Sequential times can be reduced by running on multiple homogeneous nodes or by reducing the number of runs on lower thread counts.

Expected artifact analysis time:

- Parsing and plotting of results: 10 minutes.

Artifact Setup (incl. Inputs)

Hardware. Any x86 processor, preferably with more than 16 cores. At least 128 GB of DRAM. At least 1 TB of free disk storage.

Software. The source code for the Wasp algorithm, GAP Benchmarking Suite, Galois, the MultiQueue, Δ- and ρ-stepping are included in artifact A_1. A modern GNU/Linux distribution, e.g., in our evaluation, our systems used Debian and Rocky Linux. The library and utility `libnuma` and `numactl` are required. Python version ≥ 3.10 with packages: `requests`, `pyyaml`, `numpy`, `pandas`, `matplotlib`, `scienceplots`, `openpyxl` is required to launch the experiments and to reproduce the figures of the paper.

Datasets / Inputs. The artifact requires 13 graph datasets to reproduce the results. These are publicly available, with direct URLs given below. We provide conversion utilities and instructions to convert the graphs from the textual format to the binary format of each implementation of the SSSP algorithm.

- Friendster – 10.5281/zenodo.15697359
- Kmer-v1r – 10.5281/zenodo.15697281
- Kron – 10.5281/zenodo.15698028
- Mawi – 10.5281/zenodo.15697326
- Moliere – 10.5281/zenodo.15698349
- Orkut – 10.5281/zenodo.15697210
- Road-EU – 10.5281/zenodo.15697670
- Road-USA – 10.5281/zenodo.15698160
- sk-2005 – 10.5281/zenodo.15697445
- Twitter – 10.5281/zenodo.15697688
- uk-2007 – 10.5281/zenodo.15698175
- uk-union-06 – 10.5281/zenodo.15697486
- Urand – 10.5281/zenodo.15697819

Installation and Deployment. A C++ compiler supporting C++17 is required for compiling the source code. GNU Make and CMake are also required for compilation. The artifact provides scripts for launching the experiments on a SLURM-managed cluster or supercomputer.

Artifact Execution

The tasks for the artifact execution require all implementations to be compiled and the software requirements above to be met. The artifact execution tasks are:

T_1 Optionally run the parameter tuning experiments for all thread counts and implementations. This step will also provide the data for the scaling plots.

$T_{1'}$ Alternatively, use the tuning parameters used in the paper, which are provided with the artifact, and run the scaling experiments for all implementations. These might not be optimal for the local system where the experiments are run. Therefore, the outcome might differ.

T_2 Run the experiments for the breakdown of the execution time of the GAP Benchmarking Suite and the MultiQueue.

T_3 Run the experiments for counting the number of edge relaxations in Galois, GAP, and Wasp.

If T_1 is executed, the optimal parameters will first be found for the higher thread count, and then the best parameters will be used to reduce the search space of the parameter search with lower thread counts. T_1 or the alternative T_1' must precede T_2 and T_3 that can be run in parallel

Artifact Analysis (incl. Outputs)

The artifact analysis depends on the artifact execution tasks. Python scripts and instructions are provided to parse the execution logs and replicate the figures and tables of the paper. In particular, we provide scripts for: the execution breakdown of GAP and the MultiQueue, the speedup with the highest thread count, the scaling plots (both time and speedup), the Δ parameter table, the self-speedup table, and the priority drifting analysis plot.

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Below, we present the workflow to reproduce our results on a system that fulfills all the setup requirements. The artifact provides more detailed instructions in order to operate with limited setup requirements, or to reproduce a subset of the results, through some `README` files. The top-level `README` in the artifact is the entry point for the detailed instructions. We strongly suggest reading the instructions in the artifact in order to reproduce our results successfully, even if the system fulfills all requirements.

Artifact Setup (incl. Inputs)

The following software requirements are usually pre-installed and made available in any managed cluster; therefore, we do not provide additional scripts for setup:

- C++17 compiler
- CMake
- GNU Make
- `numalib` and `numactl`

For non-cluster Linux environments, these software requirements are usually available in the distribution package manager.

We provide scripts to build the conversion utilities and SSSP algorithm implementations from their source code. This task also automatically fetches and builds required dependencies (LLVM with RTTI, Boost) if not available in your environment.

(1) Configure your environment by modifying `set_env.sh` to match your system configuration, particularly setting `SENV_CPUS` to your maximum number of cores/hyperthreads; this was set to 128 for our experiments.
(2) Load the environment variables. This is required for every new terminal session:

```
$ source set_env.sh
```

(3) Navigate to the `impl/` directory and execute the setup script:

```
$ cd impl
$ $./setup.sh all
```

The graph datasets can be acquired from the Zenodo repositories. We provide automated scripts for downloading and decompressing all or a subset of the graphs.

(1) Navigate to the `graphs/` directory
(2) Download and decompress all graphs:

```
$ cd graphs
$ ./download.sh all
```

Downloaded graphs must then be converted into the binary formats required by different SSSP implementations.

(1) Convert from Matrix Market format to GAP and Galois binary formats:

```
$ ./convert_mtx.sh both all
```

(2) After all Matrix Market conversions complete, convert GAP format to GBBS binary format:

```
$ ./convert_gap.sh all
```

Artifact Execution

The execution phase consists of four main tasks that reproduce the paper's experimental results. Tasks T_1 and T_1' are mutually exclusive alternatives for the main scaling experiments. Tasks T_2 and T_3 are independent of each other and can be executed in any order after the scaling experiments; however, they should use the same workflow (FAST or SLOW) as the chosen scaling experiment task for consistency.

All execution tasks are performed within the `experiments/` directory:

```
$ cd experiments/
```

Task T_1: SLOW Workflow. Find optimal Δ parameters for each implementation, graph, and thread count combination while generating scaling analysis data. This task implements a comprehensive parameter search that identifies the optimal Δ values for Δ-stepping algorithm implementations. The process starts with the highest thread count and progressively halves the thread count until reaching single-threaded execution.

(1) Start with the maximum thread count available on your system (e.g., 128 threads):

```
$ python3 launch_scaling.py -w slow -t 128
```

(2) After all jobs complete, run the following script to update the optimal Δ parameters:

```
$ python3 best_deltas.py
```

(3) Repeat for each thread count, halving each time, e.g., 64, 32, 16, 8, 4, 2, 1:

```
$ python3 launch_scaling.py -w slow -t 64
$ python3 best_deltas.py
# Continue this pattern until reaching 1 thread
```

(4) Execute the final `best_deltas.py` script after the single-threaded run.

Task T_1': FAST Workflow. Execute scaling experiments using the Δ parameters from the paper, providing faster reproduction with potentially suboptimal results for a different hardware configuration.

(1) Run the complete scaling analysis:

```
$ python3 launch_scaling.py -w fast
```

(2) Prepare the results for analysis:

```
$ python3 copy_best.py
```

In the rest of the document, we assume the SLOW workflow has been used. For the FAST workflow, replace any `-w slow` with `-w fast`.

Task T_2. Generate timing breakdowns for the GAP Benchmarking Suite and MultiQueue. This task provides the experimental data for understanding the execution time components and bottlenecks in different SSSP implementations.

(1) Using the same workflow choice as the scaling experiments:

```
$ python3 launch_timing.py -w slow
```

Task T_3. Count and analyze edge relaxations in Galois, GAP, and Wasp implementations to evaluate priority drifting.

(1) Using the same workflow choice as the scaling experiments:

```
$ python3 launch_relax.py -w slow
```

Artifact Analysis (incl. Outputs)

The artifact analysis phase processes the execution logs to reproduce the paper's figures. All analysis tasks depend on the completion of the artifact execution tasks and are performed within the `plots/` directory:

```
$ cd plots/
```

Speedup Analysis and Scaling Performance. Generate the primary performance evaluation figures (Figures 5-6) that demonstrate Wasp's scalability advantages over existing SSSP implementations.

Fig. 5: Generate the speedup heatmap showing comparative performance across all implementations:

```
$ python3 heatmap.py -w slow
```

Fig. 6: Create scaling plots for detailed performance analysis:

```
$ python3 scaling.py -w slow
```

Priority Drift Analysis. Analyze the edge relaxations to demonstrate how Wasp's work-stealing approach efficiently exploits priority drift to increase performance.

Fig. 7: Generate priority drift analysis plots:

```
$ python3 count-relax.py -w slow
```

Execution Time Breakdown. Provide a detailed analysis of algorithmic overhead components in the GAP Benchmarking Suite Δ-stepping implementation (barriers) and the MultiQueue-based parallel Dijkstra's algorithm (push and pop operations) implementation.

Fig. 1-2: Generate time breakdown analysis:

```
$ python3 time-breakdown.py
```

Optimal Δ Parameters. Visualize the optimal Δ parameter selection across different implementations and graph types.

Fig. 4: Generate the optimal Delta parameter visualization:

```
$ python3 deltas-heatmap.py -w slow
```

Discussion. Although absolute values might differ due to hardware differences, we expect evaluators to see consistent trends with those of the paper, confirming the speedup over the baselines and the scaling behavior shown in the paper.

Due to hardware differences, evaluators should expect:

- Performance results may differ if using the FAST workflow due to non-optimal choice of Δ.
- Optimal Δ parameters may differ from those reported in the paper when using the SLOW workflow.
- Scaling curves may show different slopes while maintaining relative performance ordering.

The key validation criterion is that Wasp consistently outperforms competing implementations across the majority of test configurations, supporting the paper's primary contributions regarding work-stealing effectiveness in SSSP algorithms.

Reproducibility Report

D Overview of Reproduction of Artifacts

The following table provides an overview of each computational artifact's reproducibility status. Artifact IDs correspond to those in the AD/AE Appendices.

Artifact ID	Available	Functional	Reproduced
A_1	•	•	•
Badge awarded	yes	yes	yes

E Reproduction of Computational Artifacts

E.1 Timeline

The artifact evaluation was conducted on August 9, 2025.

E.2 Computational Environment and Resources

The artifact evaluation was conducted on one x86 server with the following hardware and software:

- Two AMD EPYC 9664 64-core processor
- 2304 GB DDR5-4800 memory
- Ubuntu 22.04 LTS
- GCC 11.4.0
- Python 3.12

E.3 Details on Artifact Reproduction

The author followed the operation steps provided in Section "Artifact Setup" in the Artifact Evaluation part and was able to successfully set up the reproduction environment.

The author evaluated a subset of the graphs provided by the Artifact Evaluation:

- Kmer-v1r;
- Mawi;
- Orkut;
- Road-EU;
- Road-USA;
- sk-2005;
- Twitter.

The following graphs were not evaluated due to the time limitation:

- Friendster;
- Kron;
- Moliere;
- uk-2007;
- uk-union-06;
- Urand.

The author followed the "*Task T_1': Fast Workflow*" and "Task T_2" to evaluate the artifacts, then followed Section "Artifact Analysis" to visualize and analyze experiment results. Although the hardware platform is different from the one used in the paper, the author can reproduce results similar to Figures 5 and 6 in pap734. Considering that the results in these two figures are the most important parts of pap734, the author suggests that all three artifact badges should be awarded.

Disclaimer: This Reproducibility Report was crafted by volunteers with the goal of enhancing reproducibility in our research domain. The time period allocated for the reproducibility analysis was constrained by paper notification deadlines and camera-ready submission dates. Furthermore, the compute hours in the shared infrastructure (e.g., Chameleon Cloud) available to the authors of this report were limited and restricted the scope and quantity of experiments in the review phase. Consequently, the inability to reproduce certain artifacts within this evaluation should not be interpreted as definitive evidence of their irreproducibility. Limitations in the time allocated to this review and the compute resources available to the reviewers may have prevented a positive outcome. Furthermore, reviewers assess the reproducibility of the artifacts provided by the authors; however, they are not accountable for verifying that the artifacts support the main claims of the paper.