Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths

Xiaojun Dong
UC Riverside
xdong038@cs.ucr.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

Yunming Zhang
MIT
yunming@mit.edu

ABSTRACT

The single-source shortest-path (SSSP) problem is a notoriously hard problem in the parallel context. In practice, the Δ-stepping algorithm of Meyer and Sanders has been widely adopted. However, Δ-stepping has no known worst-case bounds for general graphs, and the performance highly relies on the parameter Δ, which requires exhaustive tuning. The parallel SSSP algorithms with provable bounds, such as Radius-stepping, either have no implementations available or are much slower than Δ-stepping in practice.

We propose the *stepping algorithm framework* that generalizes existing algorithms such as Δ-stepping and Radius-stepping. The framework allows for similar analysis and implementations for all stepping algorithms. We also propose a new abstract data type, lazy-batched priority queue (LaB-PQ) that abstracts the semantics of the priority queue needed by the stepping algorithms. We provide two data structures for LaB-PQ, focusing on theoretical and practical efficiency, respectively. Based on the new framework and LaB-PQ, we show two new stepping algorithms, ρ-stepping and Δ^*-stepping, that are simple, with non-trivial worst-case bounds, and fast in practice. We also show improved bounds for a list of existing algorithms such as Radius-Stepping.

Based on our framework, we implement three algorithms: Bellman-Ford, Δ^*-stepping, and ρ-stepping. We compare the performance with four state-of-the-art implementations. On five social and web graphs, ρ-stepping is 1.3–2.6x faster than all the existing implementations. On two road graphs, our Δ^*-stepping is at least 14% faster than existing ones, while ρ-stepping is also competitive. The almost identical implementations for stepping algorithms also allow for in-depth analyses among the stepping algorithms in practice.

CCS CONCEPTS

• **Theory of computation** → **Shortest paths**; **Shared memory algorithms**; • **Mathematics of computing** → **Graph algorithms**.

KEYWORDS

Single-source Shortest Paths; Parallel Algorithms; Shared-memory Algorithms; Stepping Algorithms; Parallel Priority Queue; Batch-dynamic Data Structures; ρ-stepping; Δ^*-stepping

This work is licensed under a Creative Commons Attribution International 4.0 License.

SPAA 2021, July 6–8, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8070-6/21/07.
https://doi.org/10.1145/3350755.3400227

ACM Reference Format:
Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *Proceedings of the 33nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3350755.3400227

1 INTRODUCTION

Given a weighted graph $G = (V, E, w)$ with $n = |V|$ vertices, $m = |E|$ edges, edge weight function $w : E \to \mathbb{R}^+$, and a source $s \in V$, the single-source shortest-path (SSSP) problem is to find the shortest paths from s to all other vertices in the graph. In this paper, we consider general positive edge weights. Sequentially, the best known bound for SSSP is $O(m + n \log n)$ using Dijkstra's algorithm [46] with Fibonacci heap [51]. However, SSSP is notoriously hard in parallel. Despite dozens of papers and implementations over the past decades, all existing solutions have some unsatisfactory aspects.

Practically, most existing parallel SSSP implementations [10, 43, 70, 90] are based on Δ-Stepping [68], which is a hybrid of Dijkstra's algorithm [46] and the Bellman-Ford algorithm [11, 50]. It determines the correct shortest distances in increments of Δ, in step i settling down all the vertices with distances in $[i\Delta, (i+1)\Delta]$. Within each step, the algorithm runs Bellman-Ford as substeps.

Although Δ-Stepping is the state-of-the-art practical parallel SSSP algorithm, two challenges still remain. Theoretically, Δ-Stepping has been analyzed on random graphs [38, 66], but no bounds has been shown for the general case. Practically, the parameter Δ can largely affect the algorithm's performance. The best choice of Δ depends on the graph structure, weight distribution, and the implementation itself. Fig. 1 shows the running time of three state-of-the-art Δ-Stepping implementations [43, 70, 91] and our own Δ^*-Stepping (a variant of Δ-Stepping, see Sec. 3) with different Δ values, on real-world graphs (more details in Sec. 7). A badly-chosen Δ can greatly affect the performance, and the best choices of Δ are very inconsistent for different graphs (even with the same weight distribution) and implementations. Hence, in practice, one needs exhausting searches for Δ in the parameter space as preprocessing.

Theoretically, there has been a rich literature of parallel SSSP algorithms [24, 34, 35, 57, 77, 80, 86] with $o(nm)$ work and $o(n)$ span (critical path length). Most of these algorithms rely on adding shortcuts to achieve the bounds. While these algorithms are inspiring, none of them have implementations or show practical advantages over Δ-Stepping on real-world graphs. We believe that one potential reason is the use of shortcuts and hopsets. To achieve $O(n^{1-\epsilon})$ span, these algorithms need to add $\Omega(n^{1+\epsilon})$ shortcuts. More shortcut edges contribute to more work, memory usage and footprint, hiding the advantages in the span improvement.

There has also been prior work on parallelizing the priority queue in Dijkstra's algorithm [4, 15, 28, 29, 42, 55, 61, 62, 73–76, 84, 92].

However, they do not provide interesting worst-case work and span bounds, or better performance than Δ-Stepping in practice.

We summarize existing work on parallel SSSP in Sec. 8.

Our approach. The three previous research directions on parallel SSSP (practical implementations, theoretical bounds, parallel priority queues) are mostly studied independently. We aim to design parallel SSSP algorithms combining the advantages—as simple as those using parallel priority queues, achieving worst-case guarantees that match the existing bounds, and as fast as (or faster than) Δ-Stepping in practice. Our key algorithmic insights include three components: a ***stepping algorithm framework***, which abstracts general ideas in some existing parallel SSSP algorithms, an abstract data type (ADT) ***Lazy-Batched Priority Queue (LaB-PQ)*** with efficient implementations, which extracts the semantics of the priority queue needed by stepping algorithms, and two new stepping algorithms ***ρ-Stepping and Δ*-Stepping***, which are efficient both in theory and practice.

Our stepping algorithm framework (Algorithm 1) abstracts the common idea in some existing "stepping" algorithms (e.g., Radius-Stepping [24] and Δ-Stepping [68]): in each *step*, the algorithm relaxes all vertices with tentative distances within a certain threshold, as a batch and in parallel. The two extreme cases are the two textbook algorithms: Dijkstra's algorithm with batch size 1, and Bellman-Ford algorithm with batch size n. We formalize several algorithms in this framework (Tab. 2). Interestingly, some variants of parallel Dijkstra [4, 15, 92] also use a similar high-level idea.

The proposed ADT LaB-PQ abstracts the priority queue needed by the stepping algorithms. It supports Update to commit an update to the data structure, which can be lazily batched and executed in parallel. It also supports Extract to return all records with keys within a certain threshold in parallel. The LaB-PQ is inspired by the recent work on *batch-dynamic data structures* [2, 5, 17, 79, 83, 85], where multiple updates or queries are applied to the data structure in batches in parallel. One advantage of LaB-PQ is that we do not explicitly generate the batches, but do it *lazily*. On top of the ADT, all stepping algorithms can easily use LaB-PQ's interface as a black box. Underneath it, we provide efficient data structures to support LaB-PQ. We show a theoretically efficient implementation of LaB-PQ based on the tournament tree (Sec. 4.2). It improves the cost bounds for existing parallel SSSP algorithms such as Radius-Stepping [24] and Shi-Spencer [77]. In practice, we show simple implementations based on flat arrays, which makes our stepping algorithms outperform state-of-the-art software [10, 43, 70, 90].

Based on the stepping algorithm framework and LaB-PQ, we also propose a new parallel SSSP algorithm, referred to as *ρ-Stepping*, which is simple and efficient both in theory and in practice. The high-level idea of ρ-Stepping is to relax a fixed number of unsettled vertices with small tentative distances in each step. While a similar (but not the same) idea have been used in some parallel Dijkstra's algorithms [4, 15, 92], none of them have interesting bounds or practical performance comparable to Δ-Stepping. In this paper, we formally analyze ρ-Stepping and show work and span bounds. ρ-Stepping achieves a better span bound than Radius-Stepping with a slightly higher work bound (Thm. 3.1). The work bound also applies to directed graphs (the bounds for Radius-Stepping only holds for undirected graphs). Practically, our ρ-Stepping is 1.3-2.6× faster than previous implementations on social and web graphs, and is competitive on road graphs (Fig. 3).

In addition to theoretical guarantees and practical performance, another advantage of ρ-Stepping is that, it needs no preprocessing (e.g., adding shortcuts in Radius-Stepping) or time-consuming parameter searching (e.g., finding best Δ in Δ-Stepping). Our experiments (Fig. 2) show that, the best choice of ρ is consistent and insensitive across the real-world graphs we tested.

Inspired by the stepping algorithms and LaB-PQ, we also show Δ*-Stepping, a variant of Δ-Stepping, which is simple, has non-trivial worst-case bounds (Tab. 3), and fast in practice (Fig. 3).

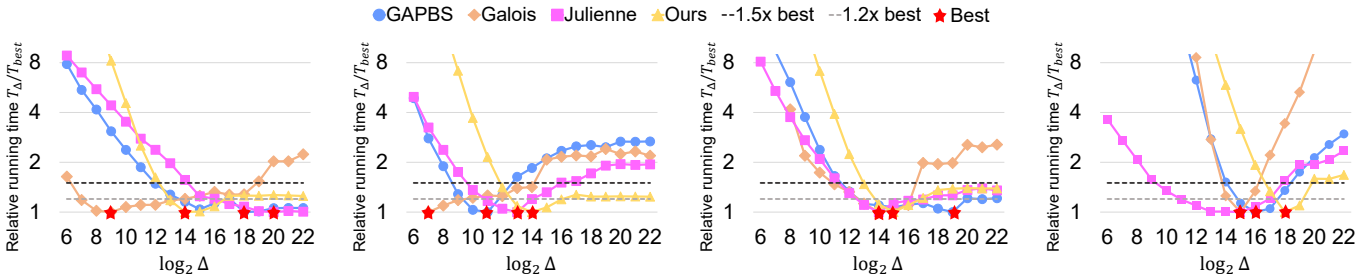
Our Contributions. Combining our LaB-PQ with existing algorithms and our new algorithms, we achieve new bounds and efficient implementations for parallel SSSP. These results are due to the abstraction of stepping algorithm framework and LaB-PQ, which greatly simplifies algorithm design, analysis, and implementation.

In theory, we show new bounds for Radius-Stepping [24], Shi-Spencer [77], Δ*-Stepping, and ρ-Stepping. We note that, with no shortcuts or hopsets, it seems unlikely to show $o(n)$ worst-case span (consider a chain). However, tighter bounds can depend on certain graph parameters, which may exhibit a good property on real-world graphs. For example, although parallel Bellman-Ford has worst-case span of $\tilde{O}(n)$, a more precise bound is $\tilde{O}(d)$, where d is the shortest-path tree depth. Indeed, on social networks with small d, parallel Bellman-Ford is reasonably fast (Table 4). To capture this, Blelloch et al. [24] proposed a graph invariant (k, ρ)-graph that indicates how "parallel" a graph is. Intuitively, a graph is a (k, ρ)-graph if every vertex reaches ρ nearest vertices in k hops. We extend this concept to analyze multiple stepping algorithms. Our experiments show that the real-world social or web graphs we tested are $(\log n, O(\sqrt{n}))$-graphs, and road graphs we tested are $(\sqrt{n}, O(n))$-graphs (Fig. 6). Under our framework, the stepping algorithms share common subroutines in analyses, such as the extraction lemma (Lem. 5.1) and the distribution lemma (Lem. 5.2).

In practice, our framework and array-based LaB-PQ give unified implementations for Bellman-Ford, Δ*-Stepping and ρ-Stepping. Our implementations achieve the best performance on all graphs (see Fig. 3). On the social and web graphs, ρ-Stepping is 1.3-2.6× faster than existing implementations. On road graphs, our Δ*-Stepping is consistently the fastest and ρ-Stepping is competitive to previous ones. This indicates the effectiveness of our framework since all optimizations are easily applicable to all algorithms. We also provide an in-depth experimental study based on our framework, especially to understand the tradeoff between work and parallelism. We show how different stepping algorithms explore the frontier in steps (Figs. 5 and 7), the parameter space (Figs. 1 and 2), and eventually draw interesting conclusions in Sec. 7.

Due to page limit, we postpone some analysis and full experimental results to the full version of the paper [48]. We summarize our contributions of this paper as follows.

- A stepping algorithm framework, which unifies multiple parallel SSSP algorithms.
- A new ADT LaB-PQ and two implementations, which are used in our analysis and implementations, respectively.
- A new parallel SSSP algorithm ρ-Stepping, which is preprocessing-free, simple and efficient both in theory and in practice.



(a). Twitter (TW) (b). Friendster (FT) (c). WebGraph (WB) (d). Road USA (USA)

Figure 1: Δ-stepping relative running time with varying Δ, including social networks (Twitter and Friendster), web graph (WebGraph), and road network (Road USA). A complete version with seven graphs is presented in the full version of this paper [48]. We use 96 cores (192 hyperthreads). We vary Δ and report the running time divided by the best running time across all Δ values. The best choice of Δ for each implementation is marked as a red star.

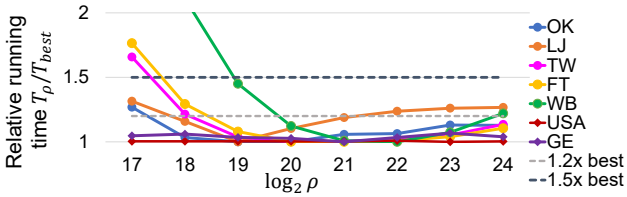


Figure 2: Relative running time of ρ-Stepping with varied ρ. We use 96 cores (192 hyperthreads). We vary ρ and tested the average running time on 100 random sources, and divided by the time with the best ρ.

		Social and Web Graphs						Road Graphs		
		OK	LJ	TW	FT	WB	Ave.	GE	USA	Ave.
Δ-step.	GAPBS	1.96	1.29	2.61	1.46	1.81	1.83	1.22	1.30	1.26
	Julienne	2.18	1.75	1.96	1.36	1.92	1.83	36.74	39.61	38.18
	Galois	1.58	1.42	1.33	1.37	1.36	1.41	1.22	1.14	1.18
	*PQ-Δ	1.00	1.03	1.15	1.26	1.19	1.13	1.00	1.00	1.00
BF	Ligra	2.02	1.45	1.67	2.53	2.01	1.93	-	-	-
	*PQ-BF	1.09	1.19	1.28	1.34	1.60	1.30	1.69	1.60	1.64
ρ-step.	*PQ-ρ-fix	1.08	1.09	1.00	1.00	1.01	1.03	1.14	1.18	1.16
	*PQ-ρ-best	1.02	1.00	1.00	1.00	1.00	1.00	1.14	1.18	1.16

Figure 3: The heat map of the parallel running time relative to the fastest on each graph. We use 96 cores (192 hyperthreads). Each column is a graph instance. "Ave." gives the average numbers over five social/web graphs and two road graphs, respectively. "*" denotes our implementations. *PQ-ρ-fix* means to use a fixed parameter ρ across all graphs in ρ-Stepping, and *PQ-ρ-best* denotes the best running time using all values of ρ.

- A new variant of Δ-Stepping (Δ*-Stepping), which is simple, with theoretical guarantee, and fast in practice.
- New analyses for stepping algorithms based on (k, ρ)-graph, which include parameterized work and span bounds for ρ-Stepping (Thm. 3.1) and Δ*-Stepping (Thm. 5.6), and improved work bounds for Radius-Stepping (Col. 5.4) and Shi-Spencer (Col. 5.5).
- Efficient parallel implementations of Bellman-Ford, Δ*-Stepping and ρ-Stepping, which outperform existing ones (Tab. 4).
- In-depth experimental study of parallel SSSP algorithms.

2 PRELIMINARIES

Computational Model. We use the work-span model for fork-join parallelism with binary forking to analyze parallel algorithms [19, 37], which is recently used in many papers on parallel algorithms [3, 12, 13, 16, 18, 20–23, 25–27, 32, 33, 36, 44, 45, 47, 53]. We assume a set of threads that share a common memory. Each thread supports standard RAM instructions, and a `fork` instruction that forks two new child threads. When a thread performs a `fork`, the two child threads all start by running the next instruction, and the original thread is suspended until all children terminate. A computation starts with a single root thread and finishes when that root thread finishes. An algorithm's ***work*** is the total number of instructions and the ***span*** (depth) is the length of the longest sequence of dependent instructions in the computation. We can execute the computation using a randomized work-stealing scheduler in practice. We assume unit-cost atomic operation WriteMin(p, v)[1], which reads the memory location pointed to by p, and write value v to it if v is smaller than the current value. We also use atomic operation TestAndSet(p), which reads and attempts to set the boolean value pointed to by p to *true*. It returns *true* if successful and *false* otherwise.

Graph Notations. We consider a weighted graph $G = (V, E, w)$. WLOG, we assume G is a connected, simple graph, with minimum edge weight $\min_{e \in E} w(e) = 1$, and no parallel edges) We use $L = \max_{e \in E} w(e)$. For $v \in V$, define $N(v) = \{u \mid (v, u) \in E\}$ as the ***neighbor set*** of v. We use $d(u, v)$ as the shortest-path distance in G between two vertices u and v. A ***shortest-path tree*** rooted at vertex u is a spanning tree T of G such that the path distance in T from u to any other $v \in V$ is $d(u, v)$.

(k, ρ)-graph. We use the concept of (k, ρ)-graph in [24] to analyze stepping algorithms. (k, ρ)-graph is a graph invariant highly related to the analysis of parallel SSSP algorithms. Intuitively, a graph is a (k, ρ)-graph if any vertex can reach its ρ nearest neighbors in k hops. More formally, we define the ***hop distance*** $\hat{d}(u, v)$ from a vertex v to u as the number of edges on the shortest (weighted) path from v to u using the fewest edges. Let $r_\rho(v)$ be the ρ-th closest distance from v, and $\bar{r}_k(v)$ the shortest distance from v to another vertex more than k-hops away.

Definition 1 ((k, ρ)-graph [24]). We say a graph $G = (V, E, w)$ is a (k, ρ)-graph if for all $v \in V$, $r_\rho(v) \le \bar{r}_k(v)$.

For a given graph $G = (V, E)$, we denote k_ρ^G to be the smallest value for k to make G a (k, ρ)-graph. With clear context, we omit the superscription. k_n is the shortest-path tree depth.

Others. We use $\log n$ as a short form of $1 + \log_2(n + 1)$. We say $O(f(n))$ ***with high probability*** (*whp*) to indicate $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \ge 1$, where n is the input size.

[1] a more practical assumption is to charge $O(t)$ work and $O(\log t)$ span when t operations priority update to a memory location. It does not change the overall bound since forking t parallel tasks requires $\Omega(\log t)$ span, which is already captured.

3 FRAMEWORKS

3.1 The LaB-PQ Abstraction

An abstract data type ***Lazy-Batched Priority Queue***, or ***LaB-PQ***, denoted as $\mathbb{PQ}$, maintains a universe of records (id, k), where $id \in I$ is the unique ***identifier*** for this record and $k \in K$ is the ***key***. In some applications, each record also has a ***value*** $v \in V$. In this paper, if not specified, we assume an empty value type for simplicity. In all SSSP algorithms in this paper, the identifiers are vertex labels from 1 to n. The total ordering of all keys is determined by a comparison function $<_K: K \times K \mapsto Bool$. A LaB-PQ $Q \in \mathbb{PQ}$ is associated with a ***mapping function*** $\delta_Q : I \mapsto K$, which maps an identifier to its corresponding key (or key-value) that can change dynamically over time. With clear context, we omit the subscription Q, and use $\delta[id]$ to denote the mapping from id to key. In the SSSP algorithms of this paper, this mapping function maps each vertex label to its (tentative) distance. In our implementation, this mapping function is passed to LaB-PQ by a pointer to the tentative distance array. More formally, a LaB-PQ $\mathbb{PQ}$ is parameterized on the following:

$\boldsymbol{I}$	Unique identifier type
$\boldsymbol{K}$	Key type
$\boldsymbol{V}$	(Optional) Value type
$\boldsymbol{<_K}: K \times K \mapsto Bool$	Comparison function on K
$\boldsymbol{\delta_Q} : I \mapsto K \times V$	A mapping from an id to its key (or key-value)

A LaB-PQ maintains a subset of identifiers in the universe. It can extract records with (relatively) small keys in parallel based on $\delta[\cdot]$. The interface of the LaB-PQ includes two functions: Update and Extract (see Table 1).We note that these two functions are sufficient for SSSP application. We discuss more functionalities of LaB-PQ in the full version of this paper [48].

Update(id) function commits an update to Q regarding the record with identifier id. It "notifies" Q that the new key for this record is now in $\delta[id]$. If id is not in Q yet, Update inserts it to Q. Multiple Update can be executed concurrently. We note that the change of the record is embodied in the change of $\delta[id]$, and thus the data structure only needs to know the record's id to address the modification. An important observation is that, we do not have to modify Q immediately, but can execute them *lazily*. These changes make no difference to any other operations on Q before the next Extract. Compared to the classic "batch-dynamic" setting, our interface avoids explicitly generating the batch, which simplifies the algorithm and improves performance.

Extract(θ) returns all identifiers in Q with key $\leq \theta$, and then deletes them from Q. Note that the result of Extract reflects all previous *modifications* to Q, including Update functions and deletions from the previous Extract. It then extracts the corresponding records based on the latest view of Q. An Extract function *cannot* be executed concurrently with other functions (Update or another Extract). This is required for LaB-PQ's correctness.

Augmenting LaB-PQ. In some applications, we need a "sum" (the *augmented value* of type A) of all records (keys and possible values) in the LaB-PQ . We refer to this as Q.Reduce(). This function first map each record in Q to a value of type A, and use a binary commutative and associative operator $\oplus$ ($(A, \oplus)$ is a commutative monoid) to compute abstract sum of all records in Q using $\oplus$.

Q.Update(id): $\mathbb{PQ} \times I \mapsto \square$	Modify the record with identifier id in Q to $\delta[id]$. If $id \notin Q$, first add it to Q.
$s = Q$.Extract(θ): $\mathbb{PQ} \times K \mapsto seq$	Return identifiers in Q with keys no more than θ and delete them from Q.

Table 1: Interface of LaB-PQ.

3.2 The Stepping-Algorithm Framework

Algorithm 1: The Stepping Algorithm Framework.

Input: A graph $G = (V, E, w)$ and a source node s.
Output: The graph distances $d(\cdot)$ from s.

```
1 δ[·] ← +∞, associate δ to a LaB-PQ Q
2 δ[s] ← 0, Q.Update(s)
3 while |Q| > 0 do
4     ParallelForEach u ∈ Q.Extract(ExtDist) do
5         ParallelForEach v ∈ N(u) do
6             if WriteMin(δ[v], δ[u] + w(u,v)) then
7                 Q.Update(v)
8     Execute FinishCheck
9 return δ[·]
```

On top of the LaB-PQ interface, we also propose a simple ***stepping algorithm framework***, in order to reveal the internal connection of the existing SSSP algorithms. Recall the two sequential textbook algorithms, Dijkstra's algorithm [46] and Bellman-Ford algorithm [11, 50]. Dijkstra only visits one vertex at a time and thus is work-efficient, but it is inherently sequential. Bellman-Ford visits all vertices in a step so it requires redundant work, but can be easily parallelized. Many parallel SSSP algorithms integrate the idea in both algorithms, and visit a subset of unsettled vertices close to the source node. Hence, they require less work than Bellman-Ford, and have better parallelism than Dijkstra. These algorithms are referred to as stepping algorithms (e.g., Δ-Stepping and Radius-Stepping) since they process a batch of vertices in a step. This is captured by LaB-PQ in the stepping algorithm framework.

We present this stepping algorithm framework in Algorithm 1. This framework requires two user-defined functions, ExtDist and FinishCheck. Many SSSP algorithms can be instantiated by plugging in different ExtDist and FinishCheck functions (see Tab. 2). Algorithm 1 starts with associate the distance array δ to a LaB-PQ Q. It then runs in ***steps***. In each step, we process vertices with distances within a threshold θ, which is computed by ExtDist and used as the parameter of Extract. The extracted vertices will relax their neighbors using WriteMin (Line 6). If successful, we call Update on the corresponding neighbor. Some algorithms (e.g., Δ-Stepping) contain substeps in each step. This is captured by FinishCheck—if the condition is not true, the threshold θ will not be recomputed. We say a vertex v is ***settled*** the last time it is extracted from the LaB-PQ and relaxes all its neighbors (and thus its distance does not change thereafter). We define the ***frontier*** as all vertices in Q, which are those waiting to be explored to relax their neighbors.

The stepping algorithm framework applies to various algorithms as shown in Tab. 2. We now briefly introduce them.

Dijkstra and Bellman-Ford. Dijkstra's algorithm visits and settles the vertex with the closest distance in the frontier. By setting θ as $\min_{v \in Q}(\delta[v])$, Algorithm 1 works the same as Dijkstra's algorithm, with the exception that multiple vertices with the same

Algorithm	ExtDist	FinishCheck	Work	Span
Dijkstra [46]	$\theta \leftarrow \min_{v \in Q}(\delta[v])$	-	$\tilde{O}(m)$	$\tilde{O}(n)$
Bellman-Ford [11, 50]	$\theta \leftarrow +\infty$	-	$\tilde{O}(k_n m)$	$\tilde{O}(k_n)$
Δ-Stepping [68]	$\theta \leftarrow i\Delta$	if no new $\delta[v] < i\Delta$, $i \leftarrow i+1$	-	-
Δ^*-Stepping (new)	$\theta \leftarrow i\Delta$	-	$\tilde{O}(k_n m)$	$\tilde{O}\left(\frac{k_n(\Delta+L)}{\Delta}\right)$
Radius-Stepping [24]	$\theta \leftarrow \min_{v \in Q}(\delta[v] + r_\rho(v))$	if there exists $\delta[v] < \theta$, do not recompute ExtDist	$\tilde{O}(k_\rho m)$	$\tilde{O}\left(\frac{k_\rho n}{\rho} \cdot \log L\right)$
ρ-Stepping (new)	$\theta \leftarrow \rho$-th smallest $\delta[v]$ in Q	-	$\tilde{O}(k_n m)$	$\tilde{O}\left(\frac{k_\rho n}{\rho}\right)$ (undirected)

Table 2: SSSP Algorithms in the stepping algorithm framework, their ExtDist and FinishCheck, and the work and span bounds based on the LaB-PQ implementation in Sec. 4. Here L is the longest edge in the graph (assuming the shortest has length 1). ρ, k_ρ and k_n are related to (k, ρ)-graph defined in Sec. 2. $\tilde{O}()$ omits $\log n$ and lower-order terms for simplicity, and the full bounds are shown in Tab. 3.

distances will be processed together, which does not affect correctness and efficiency. Finding the closest vertex can be supported using Reduce() and taking min on keys. Bellman-Ford visits all vertices in the frontier in each step, so we set θ as infinity, and in each step Algorithm 1 relaxes the neighbors of all vertices in Q.

Δ-Stepping. As a hybrid of Dijkstra and Bellman-Ford, Δ-Stepping visits and settles all the vertices with shortest-path distances between $i\Delta$ and $(i+1)\Delta$ in step i. Within each step, the algorithm runs Bellman-Ford as substeps. Hence we can set θ to $i\Delta$, and use FinishCheck to check if any newly relaxed vertex still has distance within $i\Delta$. If not, we increment i and proceed to the next step.

Δ^*-Stepping. We note that FinishCheck is not necessary for Δ-Stepping, just like other stepping algorithms. In fact, all existing implementations [10, 43, 70, 90] relaxed FinishCheck in different ways. In this paper, we show that removing FinishCheck in Δ-Stepping (referred to as ***Δ^*-Stepping***) can lead to better bounds (Thm. 5.6) and good practical performance (Sec. 7).

Radius-Stepping. In Radius-Stepping, we precompute $r_\rho(v)$, the distance from each vertex v to the ρ-th closest vertex, for all vertices. Then in each step, Radius-Stepping sets the threshold θ as $\min_{v \in Q}(\delta[v] + r_\rho(v))$, and then uses Bellman-Ford as substeps to compute the distances for vertices no more than the threshold. FinishCheck is needed by the theoretical analysis, which bounds the number of total substeps to be $O((k_\rho n/\rho) \cdot \log \rho L)$.

To implement Radius-Stepping in our framework, we need an augmented LaB-PQ. We set $r_\rho(u)$ of a vertex u as the value of each record. We map each record to $k + v$ for a record with key k (distance) and value v (vertex radius), and set the operator $\oplus$ as min. The threshold in Extract is $\theta = \min_{v \in Q}(\delta[v] + r_\rho(v))$, computed by Q.Reduce(). In Sec. 4, we show that maintaining the augmented values does not affect the asymptotical cost bounds.

ρ-Stepping. In this paper, we propose a new algorithm ρ-Stepping in the stepping algorithm framework. ρ-Stepping extracts the ρ nearest vertices in the frontier, and relaxes their neighbors. The threshold θ is the ρ-th smallest element in Q. We overload the notation of ρ from Radius-Stepping because they share high-level similarities in the theoretical analysis. The only step for ρ-Stepping in addition to the stepping algorithm framework is finding the ρ-th closest distance among all vertices in the frontier (the ExtDist). In our implementation, we simply use a sampling scheme that randomly pick $s = O(n/\rho + \log n)$ elements, sort them and pick the $(\rho s/n)$-th one. More details on how to find the ρ-th element is in the full paper, and an efficient implementation is in Sec. 6.

Picking the a subset of vertices with closest distances and relaxing their neighbors is not a groundbreaking idea, and has been used in the literature (e.g., [4, 15, 92]). However, the extracting process in previous work is either sequential or concurrent, so none of the existing algorithms support non-trivial work and span bounds, or practical efficiency as compared to Δ-Stepping. In this paper, we argue that this simple solution can achieve both theoretical and practical efficiency. Theoretically, we show that:

Theorem 3.1 (Cost for ρ-Stepping). *On a (k_ρ, ρ)-graph G, the ρ-Stepping algorithm has in $O\left(k_n m \log \frac{n^2}{m\rho}\right)$ work and $O\left(\frac{k_n n \log n}{\rho}\right)$ span. If G is undirected, the span is $O\left(\frac{k_\rho n \log n}{\rho}\right)$.*

We will first show implementations of LaB-PQ and the cost, and then formally prove this result in Sec. 5.4. ρ-Stepping also has good practical performance, which is shown in Sec. 7.

4 LAB-PQ IMPLEMENTATION

We now discuss how to efficiently support LaB-PQ in Algorithm 1. We present two data structures for LaB-PQ with the goal of theoretical and practical efficiency, respectively. The obliviousness for data structures from the algorithm's perspective is an advantage of the LaB-PQ ADT.

In our analysis, we define a ***batch of modifications*** as all Update operations between two invocations of Extract functions. The ***modification work*** on a batch B is all work paid to Update all records in B, as well as any later work (done by a later Extract) to actually apply the updates. We define a ***batch of extraction*** as all records returned by an Extract function. The ***extraction work*** on a batch B is all work paid to output the batch from the Extract function, as well as any later work (done by the next Extract) to actually remove them from Q.

4.1 Related Work

Early PRAM and BSP algorithms had explored parallel priority queues in a variety of approaches [9, 28, 31, 39, 40, 71, 72], and heavily rely on synchronization-based techniques such as pipelining. These algorithms do not have better bounds than recent batch-dynamic search trees [17, 19, 81–83] when mapping to the fork-join model. Other previous papers considered the concurrent, external-memory, and other settings [4, 15, 29, 55, 61, 62, 73–76, 84, 92]. These data structures also do not have better bounds than batch-dynamic search trees since they do not focus on optimizing work or span. However, existing batch-dynamic search trees or other

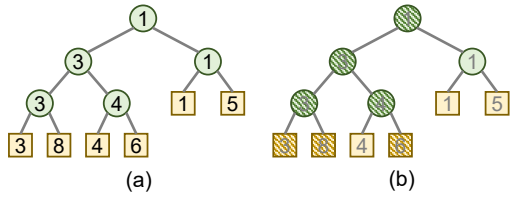


Figure 4: A tournament tree. Square leaf nodes store the records and round interior nodes keep the smallest key in their subtrees. (a) is a tournament tree containing 6 records 3, 8, 4, 6, 1 and 5. (b) shows an update on a batch of 3, 8 and 6. The shaded nodes are marked as *renewed*.

data structures (e.g., skiplists) maintaining the total ordering of the records, incur an $\Omega(\log(n))$ work lower bound per record update (more details are in the full paper). Our key observation is that maintaining total ordering, which incurs overhead both theoretically and practically, is not necessary for a parallel priority queue.

To the best of our knowledge, the only parallel data structure that has similar bounds to our new data structure is the batch-dynamic binary heap [88]. However, it has a few disadvantages: it does not support efficient batch-extract, is very complicated (no implementation available), and the span is suboptimal ($O(\log^2 n)$ in the binary fork-join model). Our new tournament-tree based LaB-PQ supports full features in the LaB-PQ, has $O(\log n)$ span, and is arguably much simpler.

4.2 Tournament-Tree-Based Implementation

We start with introducing the tournament tree (aka. winner tree). It is a complete binary tree with n external nodes (leaves) and $n-1$ interior nodes. A tournament tree stores the records in the leaves. In our use case, we only need to store the record *id* in the leaves using the LaB-PQ interface. Each interior node stores $k \in K$ (K is in key type for the records) that takes the smaller key (defined by $<_K$) from its children. Fig. 4(a) illustrates a tournament tree when keys are integers and $<_K$ is $<_{\mathbb{Z}}$.

We now discuss how to use a tournament tree to implement LaB-PQ. We will use $t.left$, $t.right$ and $t.parent$ to denote the left child, right child and parent of a node t. For simplicity, we assume the universe of the records has a fixed size n (for SSSP $n = |V|$), and the tournament tree has n leaf nodes each with a boolean flag *inQ* indicating if this record is in (has been inserted to) the LaB-PQ Q. We note that this is sufficient for the SSSP algorithms. The dynamic version (where the size of the tournament tree changes with the size of LaB-PQ) will be described in the full paper.

A tournament tree T on n records contains $2n-1$ nodes in total. The first $n-1$ nodes are interior nodes. For an interior node t, we use $t.k$ to denote the key stored in node t. To support the LaB-PQ interface, each interior node contains a bit flag *renew* indicating if any key in its subtree has been modified after the last update of $t.k$. This flag is initially set to 0 (*false*).

Constructing such a tree with given initial values simply takes linear work and $O(\log n)$ span using divide-and-conquer: construct both subtrees recursively in parallel, and update the root's key based on the two children's keys.

We next present the implementation of LaB-PQ's interface using tournament tree. Due to page limit, we only show the pseudocode (Algorithm 2) and a high-level overview here. The analysis

Algorithm 2: The Tournament-tree based LaB-PQ.

```
1  Maintains A tournament tree T with n leaf nodes each
     corresponding to a record.
2  Function Mark(record id id, boolean flag newflag)
3      Let t be the tree leaf corresponding to id
4      t.inQ ← newflag
5      while t ≠ T.root and TestAndSet(t.parent.renew) do
6          t ← t.parent
7  Function Sync(node t) → k ∈ K
8      if t is leaf then
9          if t.inQ then return δ[t.id]
10         else return +∞
11     if t.renew = 0 then return t.k
12     t.renew ← 0
13     In Parallel:
14         leftKey ← Sync(t.left)
15         rightKey ← Sync(t.right)
16     return t.k ← min(leftKey, rightKey)
17 Function ExtractFrom(threshold θ, node t) → seq
18     if t is a leaf then
19         if (δ[t.id] ≤ θ) then
20             Mark(t.id, 0)          // Marked as not in Q
21             return {t.id}
22         else return {}
23     if θ < t.k then return {}                // empty seq
24     In Parallel:
25         leftseq ← ExtractFrom(θ, t.left)
26         rightseq ← ExtractFrom(θ, t.right)
27     return leftseq + rightseq
28 Function Extract(threshold θ)
29     Sync(T.root)
30     return ExtractFrom(θ, T.root)
31 Function Update(id)
32     Mark(id, 1)
```

are given in the full paper. We first introduce a helper function Mark(id, *newflag*).

Mark(id, *newflag*) first sets the record id's *inQ* flag to be *newflag*—0 means the record should be deleted, and 1 means inserted. Whichever value *newflag* is, this means the record of id has been updated. Then, the algorithm marks the *renew* flags of the nodes on the tree path from the updated leaf to the root. This process is executed using TestAndSet. If the TestAndSet fails, we know that another Mark has marked the rest of the path, so the current Mark terminates immediately. An example is shown in Fig. 4(b). Updating the nodes' keys is postponed to the next Extract function.

Update. The Update algorithm simply calls Mark(id, 1).

Extract. Extract first uses a function Sync to update the keys for all nodes with *renew* flag as 1. It then calls ExtractFrom to output all records with keys no more than θ. Those output keys are also marked as deleted from T (Line 20).

The Sync(t) function recursively restores the keys in the interior nodes using a divide-and-conquer approach (Line 7–16), and returns the key at the current node t. The return value of a leaf node is either the record's key or infinity, depending on the *inQ* flag. For an

interior node, Sync will update the key to be the smaller one of its two children and return this key. After all interior tree nodes have been updated, we use ExtractFrom to acquire all records with keys no more than θ. This step can be parallelized similarly using divide-and-conquer (Line 17–27): we can traverse the left and right subtrees respectively and concatenate the two results. A subtree is skipped when the key at the subtree root (minimum key in the subtree) is larger than θ. Note that if we want the output sequence in a consecutive array, we can traverse for two rounds—the first round computes the number of extracted records, and the second round writes them to the corresponding slots.

We can implement Reduce similarly. We keep a collective status $t.a \in A$ (A is the augmented value type) for each interior node t, and it is updated in the Update function in Line 7–16 similar to the update for k. We do not need to update $t.a$ for node t if the subtree rooted at t remains unchanged, which is captured by $t.renew$.

Theorem 4.1. *Consider a tournament tree on a universe of n records, implemented with algorithms in Algorithm 2. The modification work on a size-b batch is $O(b \log(n/b))$. The extraction work on a size-b batch is $O(b \log(n/b))$. The span of* Extract *and* Update *is $O(\log n)$.*

The formal analysis of Thm. 4.1 is given in the full paper. We will see how to use Thm. 4.1 in Sec. 5.

4.3 Array-Based Implementation

Algorithm 2 uses a tree-based structure to provide tight work bounds for applying a batch of modifications or extractions. This is asymptotically better than batch-dynamic search trees [17, 19, 83]. However, in practice, maintaining a tree-based data structure can be expensive because of larger memory footprint and random access. Even though we can implement a tournament tree in a flat array (no pointers), it still requires extra storage for interior nodes and incurs frequent random accesses (following tree path). When the batch size b approaches n and $O(\log(n/b))$ becomes small, the theoretical advantage of tournament trees becomes insignificant, and is asymptotically the same as just loop over all records.

This is observed by the practitioners. Most (if not all) practical SSSP implementations just keep an array for all records without maintaining sophisticated structures. This is because parallel SSSP algorithms usually use a very large value of b to get sufficient parallelism. To implement Update on an array, we can just set a flag to indicate a record is added to Q. For Extract, we loop over the entire array and pack all records with keys within θ in parallel, which takes linear work and $O(\log n)$ span. While efficiently implementing the array requires many subtle details (shown in Sec. 6), asymptotically, the following bound is easy to see.

Theorem 4.2. *The array-based LaB-PQ requires $O(b)$ modification work on a size-b batch. The extraction work on a size-b batch is $O(n)$. The span of* Extract *and* Update *is $O(\log n)$.*

5 ANALYSIS FOR STEPPING ALGORITHMS

With the stepping algorithm framework (Algorithm 1) and LaB-PQ's implementation, we can now formally analyze the cost bounds for the stepping algorithms, which are summarized in Tab. 3. Our new bounds are parameterized by the definition of (k, ρ)-graph shown in [24]. We first show some useful results for all stepping algorithms in Sec. 5.1, and use them to prove the results in Sec. 5.2. We later show the span for ρ-Stepping on undirected graphs in Sec. 5.3, and compare with existing algorithms in Sec. 5.4.

5.1 Useful Results for All Stepping Algorithms

We first show two useful lemmas for all stepping algorithms.

Lemma 5.1 (Number of extractions). *In a stepping algorithm, a vertex $v \in V$ will not be extracted from the priority queue (Line 4 in Algorithm 1) more than k_n times.*

Proof. Consider the shortest path $P = \{v_0 = s, v_1, v_2, \ldots, v_l = v\}$ from the source s to v with fewest hops. Since we assume the edge weights are positive, we know that $d(s, v_i) < d(s, v_j)$ for $i < j$. Hence, whenever v is extracted from the priority queue, the earliest unsettled vertex v_i in P must also be extracted and settled. This is because v_{i-1} is already settled and have relaxed v_i in previous rounds, and $d(s, v_i) \le d(s, v)$. Based on the definition of the (k, ρ)-graph, we have $l \le k_n$, which proves the lemma. □

Lemma 5.2 (Distribution). *If a stepping algorithm has S steps, and incurs U updates (relaxations), the total work is $O(U \log(nS/U))$ using tournament-tree-based LaB-PQ.*

Proof. The work of a stepping algorithm consists of modification work for relaxations (updates) and extraction work applied to the LaB-PQ. Each extracted vertex corresponds to a previous successful relaxation, and an update and an extraction have the same cost per vertex. Hence, we only need to analyze modification costs since extraction costs are asymptotically bounded.

The U updates are distributed in S steps. Let u_i be the number of relaxations applied in step i ($\sum_i u_i = U$). The overall work across all steps is $W = O(\sum_i u_i \log(n/u_i))$. Since $u_i \log(n/u_i)$ is concave, $\sum_i u_i \log(n/u_i) \le S \cdot ((\sum_i u_i/S) \log(n/(\sum_i u_i/S))) = U \log(nS/U)$, which proves the lemma. □

5.2 Cost Bounds for Stepping Algorithms

With Lem. 5.1 and 5.2 for the stepping algorithms and Thm. 4.1 and 4.2 for LaB-PQ's cost, we can now show the cost bounds shown in Tab. 3 except for one given in Sec. 5.3.

Dijkstra's algorithm has $O(n)$ steps and $O(m)$ relaxations, Lem. 5.2 gives $O(m \log(n^2/m))$ work which is essentially better than Brodal et al.'s algorithm [28] (their span is also $O(n \log n)$ on the fork-join model). Bellman-Ford has $O(k_n)$ steps and $O(k_n m)$ relaxations, so the work is $O(k_n m)$ and the span is $O(k_n \log n)$. The following theorem shows the number of steps for ρ-Stepping.

Theorem 5.3 (Number of steps for ρ-Stepping). *On a (k_ρ, ρ)-graph, the ρ-Stepping algorithm finishes in $O(k_n n/\rho)$ steps.*

Proof. In ρ-Stepping, each step can either be a *full-extract*, where $|Q| \ge \rho$ so we extract ρ vertices with closest tentative distances, or a *partial-extract*, where $|Q| < \rho$ so we extract all but fewer than ρ vertices. There can be at most $O(k_n n/\rho)$ full-extracts, since Lem. 5.1 shows that each vertex can only be extracted for k_n times. Given that we have n vertices in total, there can be at most $O(k_n n/\rho)$ full-extracts. We now show that at most k_n partial-extracts can occur. Similar to the analysis for Lem. 5.1, once a partial-extract occurs, at least one vertex on the shortest path P from source s to

Algorithm	Work		Span	Previous Best	
	Tournament-tree-based	Array-based		Work	Span
Dijkstra [28, 46]	$O\left(m \log \frac{n^2}{m}\right)$	$O(m + n^2)$	$O(n \log n)$	$O(m \log n)$	same
Bellman-Ford [11, 50]	$O(k_n m)$	$O(k_n m)$	$O(k_n \log n)$	same	same
Δ^*-Stepping	$O\left(k_n m \log \frac{nL}{m\Delta}\right)$	$O\left(k_n m + \frac{k_n n(\Delta+L)}{\Delta}\right)$	$O\left(\left(\frac{k_n(\Delta+L)}{\Delta}\right) \log n\right)$	-	-
Radius-Stepping† [24]	$O\left(k_\rho m \log \frac{n^2 \log \rho L}{m\rho}\right)$ (U)	$O\left(k_\rho m + \frac{k_\rho n^2}{\rho} \cdot \log \rho L\right)$ (U)	$O\left(\frac{k_\rho n}{\rho} \cdot \log \rho L \log n\right)$ (U)	$O(k_\rho m \log n)$ (U)	same
Shi-Spencer† [77]	$O\left((m + n\rho) \log \frac{n^2}{m+n\rho}\right)$ (U)	$O\left(m + n\rho + \frac{n^2}{\rho}\right)$ (U)	$O\left(\frac{n \log n}{\rho}\right)$ (U)	$O((m + n\rho) \log n)$ (U)	same
ρ-Stepping	$O\left(k_n m \log \frac{n^2}{m\rho}\right)$	$O\left(k_n m + \frac{n^2 k_\rho}{\rho}\right)$ (U) $O\left(k_n m + \frac{n^2 k_n}{\rho}\right)$	$O\left(\frac{k_\rho n \log n}{\rho}\right)$ (U) $O\left(\frac{k_n n \log n}{\rho}\right)$	-	-

Table 3: New work and span bounds for the stepping algorithms and comparison to previous results. (U) indicates the bound only works for undirected graphs. (-) indicates no non-trivial bound is known to the best of our knowledge. (same) indicates the previous bound matches the tournament-tree-based work or the span. All new work bounds for Δ^*-Stepping, Radius-Stepping, Shi-Spencer, and ρ-Stepping are based on the distribution lemma (Lem. 5.2) and the LaB-PQ bounds. Radius-Stepping and Shi-Spencer (noted with †) require preprocessing.

any vertex v is settled. Based on the definition of the (k, ρ)-graph, we have $|P| \leq k_n$, so in total, at most k_n partial-extracts can occur. Putting both parts together proves the theorem. □

Combining the result with Lem. 5.2 gives the work bound of ρ-Stepping in Tab. 3.

We now show that we can get better work bounds for Radius-Stepping using LaB-PQ. Radius-Stepping extracts all vertices with distance within $\min_{v \in Q}(\delta[v] + r_\rho(v))$ in each step. The original papers uses a search tree to support this operation. We note that our LaB-PQ fully captures the need in Radius-Stepping. By replacing the search tree with our tournament tree and plugging in the numbers of relaxations and steps, we get the following results.

Corollary 5.4. *Radius-Stepping [24] uses $O\left(k_\rho m \log \frac{n^2 \log \rho L}{m\rho}\right)$ work and $O\left(\frac{k_\rho n}{\rho} \cdot \log \rho L \log n\right)$ span, with $O(m \log n + n\rho^2)$ work and $O(\rho \log \rho + \log n)$ span for preprocessing.*

We can also improve another parallel SSSP algorithm Shi-Spencer [77] by replacing their original search-tree-based priority queue with our tournament tree (more details in the full paper).

Corollary 5.5. *Shi-Spencer algorithm [77] can be computed using $O\left((m + n\rho) \log \frac{n^2}{m+n\rho}\right)$ work and $O\left(\frac{n \log n}{\rho}\right)$ span, with $O(m + n\rho^2 \log n \log \rho)$ work and $O(\log n \log \rho)$ span for preprocessing.*

We also derive the bounds for ρ-Stepping on directed graphs in Thm. 3.1, and give the formal analysis for Δ^*-Stepping:

Theorem 5.6. *Δ^*-Stepping uses $O\left(\frac{k_n(\Delta+L)}{\Delta}\right)$ steps, and thus has $O\left(k_n m \log \frac{nL}{m\Delta}\right)$ work and $O\left(\frac{k_n(\Delta+L)}{\Delta} \log n\right)$ span based on LaB-PQ.*

Due to the space limit, the proof of Δ^*-Stepping is given in the full paper. We note that for the original Δ-Stepping, such bounds do not hold. An additional factor of k_n will be introduced in span if each step needs to settle down all vertices in the distance threshold.

5.3 Number of Steps for Undirected Graphs

We can show tighter span bounds for ρ-Stepping on undirected graphs, which is inspired by existing results including Radius-Stepping [24] and Shi-Spencer's algorithm [77].

Theorem 5.7 (Number of steps, Undirected). *On an undirected (k_ρ, ρ)-graph, the ρ-Stepping algorithm finishes in $O(k_\rho n/\rho)$ steps.*

In real-world graphs, we usually have $k_\rho \ll \rho$ for large ρ. Hence, by picking a large ρ, say $n/\log n$, ρ-Stepping only requires a small number of rounds and provides ample parallelism. As a comparison, Radius-Stepping requires $O\left(\frac{k_\rho n}{\rho} \log \rho L\right)$ steps, a factor of $O(\log \rho L)$ more for the worst-case guarantee.

Due to the space limit, we show the proof in the full version of this paper, and only provide our proof sketch here. We will show that after step $(2k_\rho + 3)t$ for $t \geq 1$, ρ-Stepping will successfully settle at least the closest $t\rho$ vertices from s and relax their neighbors. This means we need $n/\rho \cdot O(k_\rho) = O(k_\rho n/\rho)$ steps. We will show this by induction. We note that the base case trivially holds when $t = 1$, since s can reach ρ closest vertices in k_ρ hops. Assume this is true for t, we will show that this is also true for $t + 1$.

More specifically, let $\mathcal{N}_\rho(u)$ be the set of ρ-nearest vertices from vertex u. For simplicity, let $\mathcal{T}_\rho = \mathcal{N}_\rho(s)$ where s is the source vertex. The inductive hypothesis assumes that vertices in $\mathcal{T}_{t\rho}$ are settled. We then show that within the next $2k_\rho + 2$ steps, all vertices in $\mathcal{T}_{(t+1)\rho} \setminus \mathcal{T}_{t\rho}$ are settled (updated to the exact distance).

Let $v \in \mathcal{T}_{(t+1)\rho} \setminus \mathcal{T}_{t\rho}$ and $P = \{s = v_0, v_1, v_2, \ldots, v_l = v\}$ be a the shortest path from s to v with the fewest hops. Assume all vertices from v_0 to v_i are in $\mathcal{T}_{t\rho}$, and beyond v_i all vertices are not in $\mathcal{T}_{t\rho}$. We show that v is within $2k + 2$ hops from v_i. More details will be given in the full version of this paper.

5.4 Comparisons and Discussions

For ρ-Stepping, the number of total steps is $O(k_\rho n/\rho)$ for undirected graphs and $O(k_n n/\rho)$ for directed graphs. The undirected case is a factor of $O(\log \rho L)$ better than Radius-Stepping (Radius-Stepping does not have non-trivial span bound on direct graphs). The work bound is off by a factor of k_n/k_ρ on undirected graphs, but it applies to directed graphs. Also, our experiments show that, on social and web graphs, k_n/k_ρ is usually small (Fig. 6) for reasonably large values of ρ (e.g., $\rho > \sqrt{n}$).

Both ρ-Stepping and Δ^*-Stepping focus on practical considerations. Since in practice we usually pick a large ρ, the number of

steps is small. This leads to a small overhead for step-based synchronization. Thm. 5.6 show that Δ^*-Stepping only incurs a factor of $1 + L/\Delta$ more steps (recall $L = \max w(e)$) than Bellman-Ford, upper bounding the synchronization cost in practice (Fig. 5). Regarding work, Thm. 3.1 and 5.6 show that both tournament tree-based and array-based versions are efficient when using proper parameters of ρ and Δ. Exactly in our experiments, the best values ρ and Δ match the analysis here (e.g., a large ρ on social networks). We note that Bellman-Ford has better work and span than both ρ-Stepping and Δ^*-Stepping. In fact, it seems hard to beat the work and span of Bellman-Ford (parameterized on k_n) if no shortcut edges are allowed. Our analysis provides worst-case guarantees for ρ-Stepping and Δ^*-Stepping, and they seem good for the (k_ρ, ρ) parameters of many real-world graphs. In practice, both ρ-Stepping and Δ^*-Stepping exhibit better performance than Bellman-Ford because of visiting fewer vertices and edges (more efficient "work"). Since analyzing SSSP algorithms based on (k, ρ)-graph is new, many interesting questions remain open.

The work for Radius-Stepping and Shi-Spencer can be improved by at most a logarithmic term.

6 IMPLEMENTATION DETAILS

We implemented three algorithms in the stepping algorithm framework: ρ-Stepping, Δ^*-Stepping, and Bellman-Ford, all using array-based LaB-PQ. Our implementations are simple, and are unified for the three algorithms (we only need to change ExtDist and FinishCheck accordingly, as shown in Tab. 2). We present some useful optimizations we used in our implementation. Most of them apply to all the three algorithms. Our code is available at: https://github.com/ucrparlay/Parallel-SSSP.

Sparse-dense optimization. We use sparse-dense optimization similar to Ligra [78]. When the current frontier is small (sparse mode), we explicitly maintain an array of vertices as the frontier. Otherwise (dense mode), we use an array of n bit flags to indicate whether each vertex is in the current frontier, and skip those not in the frontier when processing them. The dense mode has a more cache-friendly access pattern, and avoids explicitly maintaining the frontier array, but always needs $O(n)$ time to check all vertices. Hence, the sparse mode is used when the frontier size is small than a certain threshold.

Queue size estimation and scattering. One challenge in the sparse mode is maintaining the frontier array since the size can change dramatically during the execution. Some existing implementations (e.g., Ligra) use a parallel pack to generate the next frontier sequence, which scans all edges incident the current frontier for two rounds (one for computing offsets and another round to pack). This can incur a large overhead. To avoid this, we use a resizable hash table to maintain the next frontier, and scatter the vertices to the next frontier by putting them into random slots in the hash table. In the process of our algorithm, we use sampling to estimate the next frontier size in order to resize the hash table.

Bidirectional relaxation for undirected graphs. We use a novel optimization for undirected graphs. Before the algorithm relaxes all v's neighbors (Line 5 in Algorithm 1), it first attempts to relax v using all its neighbors. This aims to update v's distance first, so it will be more "effective" when v relaxes other vertices later. Another reason is that parallel SSSP implementation is usually I/O bounded. Since in relaxations, we need to check v's neighbors' distances anyway, we can load them to the cache and use them to relax v's tentative distance first with small cost. This optimization only applies to undirected graphs.

Threshold estimation for ρ-Stepping. In both Δ-Stepping and Radius-Stepping (although we did not implement Radius-Stepping), the distance threshold can be directly computed. In ρ-Stepping, we need to compute the threshold (the ρ-th smallest element in the frontier) in each step. We use the sampling-based idea as mentioned in Sec. 3.2. In particular, at the beginning of Extract, we first sample $s = O(n/\rho + \log n)$ uniformly random samples from the current frontier. Then we sort the samples and pick the threshold from the samples. Since s is small, this step is sequential and fast. In ρ-Stepping, if the frontier size is smaller than ρ, we pick θ as the maximum distances in the frontier.

In our experiments, we observe that in ρ-Stepping, the threshold estimation in the first dense rounds is usually inaccurate. This is because in the early stage, the ρ-th closest distance in the frontier is usually far from the source, and during the relaxation, much more vertices go below this threshold. Hence, we add a heuristic to adjust the threshold: using 10% of ρ at the first two dense rounds.

Large neighbor sets. On road networks and the begin and end for all graphs, the frontier and its neighborhood are very small. Relaxing the neighbors in a round-based manner leads to insufficient workload and thus overhead in the synchronization cost. To optimize this case, we use a similar "bucket fusion" optimization proposed by Zhang et al. [90], which is later integrated to GAPBS [10]. In our Δ^*-Stepping and ρ-Stepping, when processing a vertex v, instead of using v's direct neighbors, we run a local BFS until we reach $t = 4096$ vertices (or when the tentative distances reach more than θ). We use these vertices as v's neighborhood $\mathcal{N}(v)$, and update them all. Note that this information is maintained and processed locally. As such, we can extend multiple hops in one round. We apply this optimization in sparse rounds with average edge degree fewer than 20, and thus call them *super sparse* rounds. This optimization can greatly optimize the performance for road networks, since as shown in Fig. 6, the values of k_ρ on road graphs is large. The impact on the performance for social and web graphs is smaller since the dense rounds spend the most time.

7 EXPERIMENTS

Experimental setup. We run all experiments on a quad-socket machine with Intel Xeon Gold 6252 CPUs with a total of 96 cores (192 hyperthreads). The system has 1.5TB of main memory and 36MB L3 cache on each socket. Our codes were compiled with `g++` 7.5.0 using `CilkPlus` with `-O3` flag. For all parallel implementations, we use all cores and `numactl -i all`, which evenly spreads the memory pages across the processors in a round-robin fashion.

We implemented three algorithms based on the framework in Sec. 3: Bellman-Ford (*PQ-BF*), Δ^*-Stepping (*PQ-*Δ^*), and ρ-Stepping (*PQ-*ρ). We use array-based LaB-PQ because we observe that when the output size of Extract is large, the array-based implementation has better performance than tournament tree (see more details in the full paper). For all graphs we use, the best running time is

	Graph	Social												Web			Road					
		OK			LJ (D)			TW (D)			FT			WB (D)			GE			USA		
	#vertices	3M			4M			42M			65M			89M			12M			24M		
	#edges	234M			68M			1.47B			3.61B			2.04B			32M			58M		
	#threads	(1)	(96h)	(SU)	(1)	(96h)	(SU)	(1)	(96h)	(SU)	(1)	(96h)	(SU)	(1)	(96h)	(SU)	(1)	(96h)	(SU)	(1)	(96h)	(SU)
Δ-step.	**GAPBS**	3.42	.240	14.2	1.14	.103	11.0	58.6	2.42	24.2	84.7	2.95	28.7	50.8	1.92	26.5	2.01	0.22	9.1	1.83	0.33	5.5
Δ-step.	**Julienne**[1]	4.82	.268	18.0	2.86	.140	20.4	43.1	1.82	23.7	95.4	2.75	34.7	86.1	2.04	42.2	1.54	6.62	0.2	2.04	10.16	0.2
Δ-step.	**Galois**	3.08	.194	15.9	1.72	.113	15.1	29.7	1.23	24.2	92.2	2.76	33.4	45.0	1.45	31.1	2.80	0.22	12.8	2.72	0.29	9.3
Δ-step.	***PQ-Δ***	3.45	**.123**	28.1	2.04	.082	25.0	39.3	1.07	36.9	115.4	2.55	45.3	62.8	1.27	49.6	5.54	**0.18**	30.7	4.81	**0.26**	18.8
BF	**Ligra**	5.07	.248	20.5	2.55	.115	22.1	42.6	1.55	27.5	218.2	5.12	42.6	81.4	2.13	38.2	-	-	-	-	-	-
BF	***PQ-BF***	3.71	.134	27.7	2.58	.095	27.2	45.7	1.18	38.6	147.7	2.72	54.4	97.6	1.71	57.2	12.97	0.30	42.6	16.28	0.41	39.8
ρ-step.	***PQ-ρ-fix***	3.56	.132	27.0	2.46	.087	28.2	37.6	**0.93**	40.6	112.7	**2.02**	55.8	60.6	1.07	56.7	6.43	0.21	31.1	3.84	0.30	12.7
ρ-step.	***PQ-ρ-best***	3.42	.125	27.5	2.07	**.080**	28.6	37.6	**0.93**	40.6	112.7	**2.02**	55.8	57.5	**1.06**	54.1	6.43	0.21	31.1	3.86	0.30	12.8
		($\rho = 2^{19}$)			($\rho = 2^{19}$)			($\rho = 2^{21}$)			($\rho = 2^{21}$)			($\rho = 2^{22}$)			($\rho = 2^{21}$)			($\rho = 2^{23}$)		

Table 4: Parallel and sequential running times for all implementations on all graphs. Our implementations are noted with ∗. (D): directed graph. (1): running time on one core. (96h): running time using 96 cores with hyperthreading (192 threads). (SU): speedup. On each graph, bold numbers are the fastest running time, and underline numbers denote the fastest Δ-Stepping implementation and the fastest Bellman-Ford implementation on each graph instance. For all Δ-Stepping algorithms, we report the best running time across all values of parameter Δ. For ρ-Stepping, we report the best running time across all values of parameter ρ as *PQ-ρ*-best, and report the running time with a fixed value of $\rho = 2^{21}$ as *PQ-ρ*-fix.
[1]: Julienne does not achieve satisfactory performance on road graphs. We have checked this with the authors, and the reason is that Julienne was not optimized on road graphs. The reported numbers are the best among all possible values of Δ.

achieved using a reasonably large ρ. We compare our implementations with state-of-the-art SSSP implementations: Bellman-Ford algorithm in Ligra [78], Δ-Stepping in Julienne [43], GAPBS [10, 90], and Galois [70]. Throughout the section, when we refer to "Δ-Stepping", it includes our Δ*-Stepping, and the existing Δ-Stepping in Julienne, GAPBS and Galois.

We test seven graphs, including four social networks com-orkut (OK) [89], Live-Journal (LJ) [8], Twitter (TW) [59] and Friendster (FT) [89], one web graph WebGraph (WB) [64], and two road graphs [1] RoadUSA (USA) and Germany (GE). The graph information is provided in Table 4. In almost all experiments, the social and web graphs show a similar trend. This is because they follow similar power-law-like degree distribution. Throughout the section, we use "scale-free networks" to refer to social and web graphs. On scale-free networks, we set edge weight uniformly at random in range $[1, 2^{18})$. On road graphs, the edge weights are from the original dataset, which is up to 2^{25}.

For all Δ-Stepping algorithms (except for Fig. 1 where we vary Δ), we report the *best running time* across all Δ values. When we report average of multiple sources, we first find the best Δ value on one source, and use it for other sources. We do this for every graph-implementation combination. For all ρ-Stepping algorithms (except for in Tab. 4 where we explicitly report the best running time across ρ values), we use a fixed value of $\rho = 2^{21}$. For most experiments, we report the average of 10 sources. When taking the average is meaningless, we use one representative source.

In this section, we will first discuss the overall performance of all implementations. We then compare some statistics to better understand the performance of *PQ-ρ*, *PQ-Δ** and *PQ-BF*. We evaluate the number of vertices visited by the algorithm as an indicator of the overall work. Since road graphs exhibit different properties from the scale-free networks, we then discuss road graphs separately. We also analyze the two algorithms ρ-Stepping and Δ-Stepping with their corresponding parameters. Due to page limit, we will discuss the k_ρ properties for each graph in the full version, and only show the k_ρ-ρ curves in this paper (Fig. 6). Due to page limit, we postpone some figures, discussions and more experiments to the full paper [48] (e.g., using different machines and different types of sources). Generally, we show that our ρ-Stepping has **especially good performance on scale-free networks**, and the performance gain of ρ-Stepping is from three aspects: **good parallelism**, **less overall work**, and **more evenly distributed work to all steps**. We summarize conclusions and interesting findings at the end of this section.

Overall Performance. We present the running time of all implementations in Tab. 4. In all cases, one of our implementations achieves the best performance, and is 1.14× to orders of magnitude faster than the previous implementations. We show a heat map of relative parallel running time in Fig. 3.

On scale-free networks, *PQ-ρ* and *PQ-Δ** outperform all existing implementations. *PQ-ρ* has better performance. On average over five graphs, *PQ-ρ* is 1.41× faster than Galois, 1.83× faster than Julienne and GAPBS, and 1.93× faster than Ligra.

On road graphs, *PQ-Δ** is the fastest, and *PQ-ρ* is also competitive. Ligra did not finish in 30 seconds on road graphs, since Ligra uses plain Bellman-Ford that is inefficient for graph with deep shortest-path tree (more than 10^4, see Fig. 6). Our *PQ-BF* with the neighbor-set optimization (see Sec. 6) finishes on both graphs in about 0.4s.

We report the sequential running time of the corresponding parallel version and show self-speedup in Table 4. We note that comparing the sequential running time of different implementations does not seem useful because both Δ-Stepping and ρ-Stepping are parameterized. To get the best sequential performance, one should just use a small Δ or ρ. The reported time is the sequential performance using the corresponding parameter that performs best in parallel, and it makes more sense just to compare the speedup numbers. The self-speedup of *PQ-ρ* is almost always the best among all implementations (*PQ-Δ** is close but slightly worse). Hence, the good performance of *PQ-ρ*, especially on scale-free networks, is partially due to good scalability. In other words, *PQ-ρ* achieves the best "work-span tradeoff" in practice.

Among the implementations of the same algorithm, *PQ-BF* outperforms Ligra on all graphs. For all Δ-Stepping algorithms, *PQ*-Δ* is also the fastest on all graphs. Overall, our three algorithms outperform existing implementations, indicating the efficiency of stepping algorithm framework for parallel SSSP implementations.

Number of visits to vertices. Unlike Dijkstra, other parallel SSSP algorithms can visit each vertex or edge more than once. While this allows for parallelism, the total work is also increased. To show how much "redundant" work is done for the stepping algorithms, we measure the average number of visits per vertex (Fig. 7), and the number of visited vertices in each step on four representative graphs (Fig. 5)[2]. We note that the other systems vary a lot in implementation details, and it is hard to directly measure these quantities from their code. Hence, we compare among our implementations. For the same reason, *PQ*-Δ* may not precisely reflect the numbers of other Δ-Stepping implementations. In this paragraph, we first focus on the scale-free networks, and discuss road graphs later.

Figure 7 shows the average number of visits per vertex. On the two small graphs (OK and LJ), since the work cannot saturate all 192 threads, *PQ*-ρ act similar to Bellman-Ford to maximize parallelism and uses visits more vertices than Δ-Stepping. For the larger graphs (TW, FT, and WB), *PQ*-ρ always triggers the smallest average visit to vertices. The trend showed in Fig. 7 exactly matches the sequential time of each implementation. Hence, one advantage of *PQ*-ρ over *PQ-BF* and *PQ*-Δ* on scale-free networks is less total work.

Figure 5 shows the number of visited vertices per step. In *PQ-BF*, the numbers always grow quickly to a large value, stay for a few steps, and finish quickly. Although usually using the fewest steps, *PQ-BF* is the slowest, since the dense steps cause many redundant relaxations. *PQ*-Δ* usually uses more steps than both *PQ-BF* and *PQ*-ρ. In most of the steps (at the beginning and the end), *PQ*-Δ* visits only a small number of vertices, but the peak values are much higher than *PQ*-ρ. *PQ*-ρ shows a more even pattern across the steps: in most of the steps, it processes a moderate number of vertices, and the peak value is much smaller than *PQ*-Δ* or *PQ-BF*.

These patterns in Fig. 5 reflect the nature of the three algorithms. Bellman-Ford always visits all vertices in the frontier in each step. This created significant redundant work. Δ*-Stepping controls work-span tradeoff based on distances. On scale-free networks, it reaches the peak work in some middle steps, which is significantly higher than other steps. ρ-Stepping controls the work-span tradeoff using the number of vertices processed per step. We believe on scale-free networks, this quantity is a closer indicator to the actual "work" in each step than the distance gap is. In other words, ρ-Stepping controls the work in each step that is minimal to saturate all processors, so it explores sufficient parallelism with minimized redundant work.

Discussions for Road Graphs. Road graphs are (almost) planar and have different *k*-ρ patterns than other graphs, and the shortest-path trees are deep and slim. Hence, without the special optimizations (e.g., in Ligra and Julienne), the performance is slow. As mentioned in Sec. 6, our optimization expands multiple levels in the shortest-path tree in one step. This makes the performance of our implementations competitive or better than GAPBS and Galois.

On road graphs, *PQ*-Δ* is the fastest. This somehow indicates that expanding with distance may be a good strategy for road graphs. One possible reason is that they are planar graphs with Euclidean distance. Hence, setting fixed-width "annuli" seems a reasonable work-parallelism tradeoff, when using a proper Δ. Since the frontier on road graphs is small, *PQ*-ρ has insufficient frontier size in each step for enough parallelism. Hence, it is hard for *PQ*-ρ to control the number of vertices visited precisely, and the performance is slightly slower than *PQ*-Δ*. However, *PQ*-ρ has more stable performance than *PQ*-Δ* in the parameter space (Figs. 1 and 2).

Δ-Stepping and Δ. We test all Δ-Stepping algorithms with varying Δ on all graphs. For each test case, we normalize the running time to the best time across all Δ values. For page limit, we present four graphs in Fig. 1, and the full results in the full paper [48].

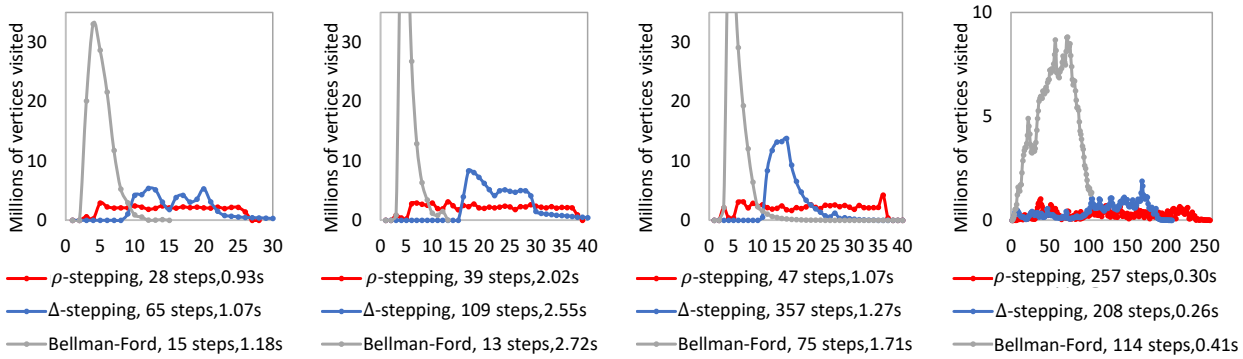
On the same graph, the best choice of Δ varies a lot for different systems. On TW, Julienne's best Δ is $2^{12}\times$ larger than Galois's. The best Δ for one system can make another system up to 4× slower. The selection of Δ in one system does not generalize to other systems. Secondly, even though all scale-free networks have the same edge weight distribution, for the same implementation, the best choice of Δ varies a lot on different graphs. Therefore, the selection of Δ on one graph does not generalize to other graphs. On the same graph, the performance is sensitive to the value of Δ. Usually, 2–4× off may lead to a 20% slowdown, and 4–8× off may lead to a 50% slowdown. A badly-chosen Δ can largely affect the performance. As a result, for every graph-implementation combination, we have to search the best parameter Δ. Fortunately, we find out that different sources show relatively stable performance for the same implementation-graph pair. This is also the conventional way of tuning Δ (we also did so). We present the results in the full paper [48].

ρ-Stepping and ρ. We test ρ-Stepping with varying ρ. When ρ is small, the running time increases significantly. This is also due to the lack of parallelism (similar to when Δ-Stepping uses small Δ). When ρ gets large, the performance drops by no more than 20%. The best choices of ρ are very consistent on different graphs. This is because the choice of ρ in practice depends on the right level of parallelism we want to achieve, instead of the graph structure or edge weight distribution. As discussed, ρ-Stepping distributes work more evenly to each step. The goal of setting ρ is to enable enough work to exploit full parallelism in each step, but without introducing more redundant work. The performance on road graphs are less sensitive, probably because the frontier size seldom reaches ρ in road graphs. We also tested ρ-Stepping on various machines. We observe that the best choice of ρ is still relatively consistent among different settings. We will present more results in the full paper.

Generally speaking, using large Δ or ρ gives better (and more stable) performance than small Δ or ρ values. This is not surprising because when these parameters are large, Δ-Stepping and ρ-Stepping degenerate to Bellman-Ford that still has reasonable performance on social networks. When the parameters are small, Δ-Stepping and ρ-Stepping both degenerate to Dijkstra and loses parallelism.

Summary. In summary, our *PQ*-ρ generally achieves the best performance on the five scale-free networks. On average of the five graphs, *PQ*-ρ is 1.41-1.93× faster better than existing systems. On

[2]We also measured the number of visited edges, which show very similar trend to the vertices. Due to page limit, we report the numbers in the full version of the paper, which also shows the results for all seven graphs.



(a). TW **(b). FT** **(c). WB** **(d). USA**

Figure 5: Number of visited vertices in each step in *PQ*-ρ, *PQ*-Δ^* and *PQ-BF*. Here we only run on one source vertex, since it has unclear meaning to compute the average of multiple runs on each step. Hence, the runtimes can be different from Table 4 (average on 100 runs from 10 source vertices), and some curves are bumpy. We use 96 cores (192 hyperthreads).

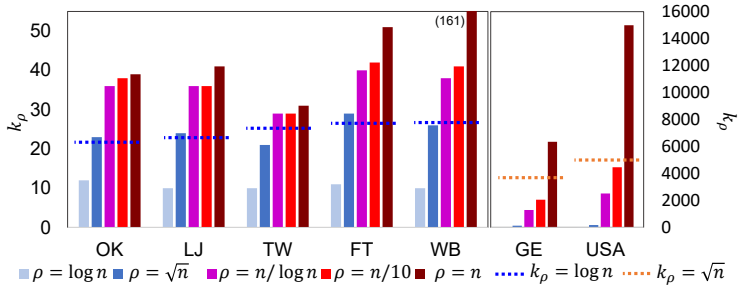


Figure 6: The values of k_ρ with different values of ρ for different graphs.

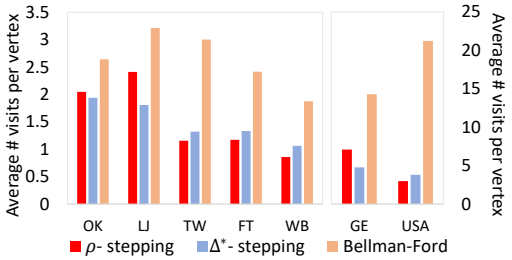


Figure 7: Number of visits per vertex and per edge, respectively, for *PQ*-ρ, *PQ*-Δ^* and *PQ-BF* on all graphs.

the two road graphs, *PQ*-Δ^* always has the best performance, which is at least 14% better than existing systems. The good performance of *PQ*-ρ on scale-free networks comes from three aspects. The first is scalability, indicated by the good self-speedup. Secondly, it visits fewer vertices and edges on large scale-free networks, which indicates less overall work. Lastly, the work is more evenly distributed to each step, such that each step can exploit sufficient parallelism, and also avoid performing "ineffective" work to relax the neighbors of unsettled vertices. This also indicates that on scale-free networks with uniformly distributed edge weights, controlling the number of vertices visited per step is a good strategy. On road graphs with Euclidean distance, Δ-Stepping shows better performance.

Our *PQ*-ρ implementation generally shows stable performance across ρ values on all tested graphs. A fixed ρ almost always gives performance within 5% off the performance with the best ρ.

Finally, on all tested graphs, *PQ*-ρ and *PQ*-Δ^* are faster than all existing SSSP implementations (except for RoadUSA, *PQ*-ρ is 0.01s slower than Galois). *PQ-BF* is faster than Ligra on all graphs. This indicates the efficiency of the stepping algorithm framework on implementing and optimizing parallel SSSP algorithms.

8 RELATED WORK ON PARALLEL SSSP

Practical parallel SSSP implementations. There have been dozens of practical implementations of parallel SSSP. In this paper, we compared to a few of them. Galois [70] uses an approximate priority queue ordered by integer metric with NUMA-optimization to improve the performance of SSSP. GraphIt [90, 91] proposed a priority queue abstraction and a new optimization, bucket fusion, to reduce the synchronization overhead of Δ^*-Stepping. The optimizations are later adopted by GAPBS [10], which is the one we compared to. Julienne [43] proposed and used the bucketing data structure to order the vertices for Δ-Stepping based on semisorting [54]. Ligra [78] includes one of the most efficient Bellman-Ford implementations.

There has also been a significant amount of work on other implementations, include those on the distributed setting [14, 63, 93], GPUs [41, 87], among many others. Our reported running time in this paper is much faster than in these papers on the same graphs, and comparing the superiorities of different settings on parallel SSSP is out of the scope of this paper. Parallel SSSP based on parallel priority queues are reviewed in Sec. 4.

Theoretical work on parallel SSSP. There has been a rich literature of theoretical parallel SSSP algorithms. Among them, many algorithms [34, 35, 57, 77, 80, 86] achieve very similar bounds to Radius-Stepping [24] we discussed in this paper, but require adding shortcut edges. Basically the product of work and span is $\tilde{\Theta}(nm)$ (referred to as the transitive closure bottleneck [56]). Some algorithms are analyzed based on edge weights [65, 67], and many others are on approximate shortest-paths [6, 30, 49, 60, 69] and other models [7, 52, 58]. While these algorithms are insightful, to the best of our knowledge, none of them have implementations.

ACKNOWLEDGEMENT

This work is in partial supported by NSF grant CCF-2103483.

REFERENCES

[1] Openstreetmap © openstreetmap contributors. https://www.openstreetmap.org/, 2010.

[2] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala. Parallel batch-dynamic graph connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 381–392, 2019.

[3] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[4] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 11–20, 2015.

[5] D. Anderson, G. E. Blelloch, and K. Tangwongsan. Work-efficient batch-incremental minimum spanning trees with applications to the sliding-window model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[6] A. Andoni, C. Stein, and P. Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *ACM Symposium on Theory of Computing (STOC)*, pages 322–335, 2020.

[7] J. Augustine, K. Hinnenthal, F. Kuhn, C. Scheideler, and P. Schneider. Shortest paths in a hybrid network model. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1280–1299. SIAM, 2020.

[8] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 44–54, 2006.

[9] A. Bäumker, W. Dittrich, F. Meyer, and I. Rieping. Realistic parallel algorithms: Priority queue operations and selection for the bsp* model. In *European Conference on Parallel Processing*, pages 369–376. Springer, 1996.

[10] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[11] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[12] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[13] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[14] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 93–104, 2017.

[15] T. Bingmann, T. Keh, and P. Sanders. A bulk-parallel priority queue in external memory with stxxl. In *International Symposium on Experimental Algorithms (SEA)*, pages 28–40. Springer, 2015.

[16] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008.

[17] G. E. Blelloch, D. Ferizovic, and Y. Sun. Just join for parallel ordered sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[18] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[19] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[20] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2004.

[21] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[22] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[23] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[24] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan. Parallel shortest paths using radius stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[25] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1998.

[26] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory of Computing Systems (TOCS)*, 32(3), 1999.

[27] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[28] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.

[29] I. Calciu, H. Mendes, and M. Herlihy. The adaptive priority queue with elimination and combining. In *International Symposium on Distributed Computing (DISC)*, pages 406–420. Springer, 2014.

[30] N. Cao, J. T. Fineman, and K. Russell. Efficient construction of directed hopsets and parallel approximate shortest paths. In *ACM Symposium on Theory of Computing (STOC)*, pages 336–349, 2020.

[31] D. Z. Chen and X. S. Hu. Fast and efficient operations on parallel priority queues. In *International Symposium on Algorithms and Computation*, pages 279–287. Springer, 1994.

[32] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 339–350, 2017.

[33] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

[34] E. Cohen. Using selective path-doubling for parallel shortest-path computations. *Journal of Algorithms*, 22(1):30–56, 1997.

[35] E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 47(1):132–166, 2000.

[36] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4), 2017.

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[38] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer, 1998.

[39] V. A. Crupi, S. K. Das, and M. C. Pinotti. Parallel and distributed meldable priority queues based on binomial heaps. In *ICPP Workshop on Challenges for Parallel Processing*, volume 1, pages 255–262. IEEE, 1996.

[40] S. K. Das, M. C. Pinotti, and F. Sarkar. Optimal and load balanced mapping of parallel priority queues in hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):555–564, 1996.

[41] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 349–359. IEEE, 2014.

[42] N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, 1992.

[43] L. Dhulipala, G. E. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[44] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[45] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun. Semi-asymmetric parallel graph algorithms for nvrams. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9), 2020.

[46] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), 1959.

[47] D. Dinh, H. V. Simhadri, and Y. Tang. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[48] X. Dong, Y. Gu, Y. Sun, and Y. Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. *arXiv preprint 2105.06145*, 2021.

[49] M. Elkin and O. Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM Journal on Computing*, 48(4):1436–1480, 2019.

[50] L. R. Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

[51] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3), 1987.

[52] M. Ghaffari and J. Li. Improved distributed algorithms for exact shortest paths. In *ACM Symposium on Theory of Computing (STOC)*, pages 431–444, 2018.

[53] Y. Gu, O. Obeya, and J. Shun. Parallel in-place algorithms: Theory and practice. pages 114–128, 2021.

[54] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[55] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 317–328, 2013.

[56] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. MIT Press, 1990.

[57] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

[58] F. Kuhn and P. Schneider. Computing shortest paths and diameter in the hybrid network model. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 109–118, 2020.

[59] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[60] J. Li. Faster parallel algorithm for approximate shortest path. In *ACM Symposium on Theory of Computing (STOC)*, pages 308–321, 2020.

[61] J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*, pages 206–220. Springer, 2013.

[62] Y. Liu and M. Spear. A lock-free, array-based priority queue. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 323–324, 2012.

[63] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[64] R. Meusel, O. Lehmberg, and S. Bizer, Christian andVigna. Web data commons - hyperlink graphs. http://webdatacommons.org/hyperlinkgraph.

[65] U. Meyer. Heaps are better than buckets: parallel shortest paths on unbalanced graphs. In *European Conference on Parallel Processing*, pages 343–351. Springer, 2001.

[66] U. Meyer. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 797–806, 2001.

[67] U. Meyer. Buckets strike back: Improved parallel shortest-paths. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 8–pp. IEEE, 2002.

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

[69] G. L. Miller, R. Peng, A. Vladu, and S. C. Xu. Improved parallel algorithms for spanners and hopsets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 192–201, 2015.

[70] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471, 2013.

[71] M. C. Pinotti and G. Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33–40, 1991.

[72] A. Ranade, A. Cheng, E. Deprit, J. Jones, and S. Shih. Parallelism and locality in priority queues. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 490–496. IEEE, 1994.

[73] P. Sanders. Fast priority queues for cached memory. *J. Experimental Algorithmics*.

[74] P. Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998.

[75] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures*. Springer.

[76] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268. IEEE, 2000.

[77] H. Shi and T. H. Spencer. Time-work tradeoffs of the single-source shortest paths problem. *Journal of Algorithms*, 30(1):19–32, 1999.

[78] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013.

[79] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[80] T. H. Spencer. Time-work tradeoffs for parallel algorithms. *jacm*, 44(5):742–778, 1997.

[81] Y. Sun and G. Blelloch. Implementing parallel and concurrent tree structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, page 447–450, 2019.

[82] Y. Sun and G. E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.

[83] Y. Sun, D. Ferizovic, and G. E. Blelloch. Pam: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.

[84] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[85] T. Tseng, L. Dhulipala, and G. Blelloch. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106. SIAM, 2019.

[86] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[87] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–12, 2016.

[88] Y. Wang, S. Yu, Y. Gu, and J. Shun. A parallel batch-dynamic data structure for the closest pair problem. In *ACM Symposium on Computational Geometry (SoCG)*, 2021.

[89] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[90] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun. Optimizing ordered graph algorithms with graphit. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, pages 158–170, 2020.

[91] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[92] T. Zhou, M. Michael, and M. Spear. A practical, scalable, relaxed priority queue. In *International Conference on Parallel Processing (ICPP)*, pages 1–10, 2019.

[93] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 301–316, 2016.