Efficient Communication-Aware Distributed Δ-Stepping for Single-Source Shortest Paths

Rakibul Hassan and Shaikh Arifuzzaman
Department of Computer Science
University of Nevada, Las Vegas (UNLV), Las Vegas, USA.
Email: *hassar2@unlv.nevada.edu, shaikh.arifuzzaman@unlv.edu.*

***Abstract*—Single-Source Shortest Path (SSSP) is a fundamental kernel in large-scale graph analytics and many scientific and engineering applications, including transportation modeling, network resilience analysis, and graph-based scientific workflows. Among parallel approaches, Δ-Stepping has emerged as a widely used alternative to Dijkstra's algorithm for handling large weighted graphs. However, existing distributed-memory implementations often face scalability limitations due to frequent frontier exchanges and costly global synchronization.**

In this paper, we present a communication-efficient distributed Δ-stepping algorithm that eliminates explicit frontier communication. Instead of exchanging frontier sets across processes, our proposed method performs a single distance-only `MPI_Allreduce` per iteration and reconstructs the active frontier locally by comparing successive distance vectors. By limiting communication to one collective per iteration, this design enables scalable SSSP on distributed-memory systems.

Our evaluation across synthetic and real-world networks shows that the proposed algorithm scales efficiently to billions of edges. With 16 MPI processes, we achieve up to a $10\times$ speedup relative to sequential Dijkstra's algorithm. Crucially, the implementation maintains robust performance across diverse and irregular graph topologies, bridging the gap between theoretical scalability and real-world application.

***Index Terms*—SSSP, Δ-Stepping, Distributed Graph Algorithms, MPI, Communication Optimization, Scientific Computing**

I. Introduction

The Single-Source Shortest Path (SSSP) problem [12], [18], [23], [32] seeks to compute the shortest-path distances from a designated source vertex to all other vertices in a graph. It is a foundational problem in graph theory and graph analytics, with applications spanning transportation systems [5], computer networks [16], social network analysis [21], and biological network modeling [25]. In many real-world settings, SSSP is not an isolated computation but rather a core primitive embedded within larger analytical pipelines. Representative examples include routing and traffic-flow modeling in transportation simulations [19], [22], reachability and resilience analysis in communication and power-grid infrastructures [27], shortest-path–based heuristics in computational biology [2], and data-intensive graph processing frameworks [34]. In these environments, SSSP computations are often executed repeatedly across multiple sources, time steps, or optimization iterations, causing the cumulative computational cost to grow rapidly.

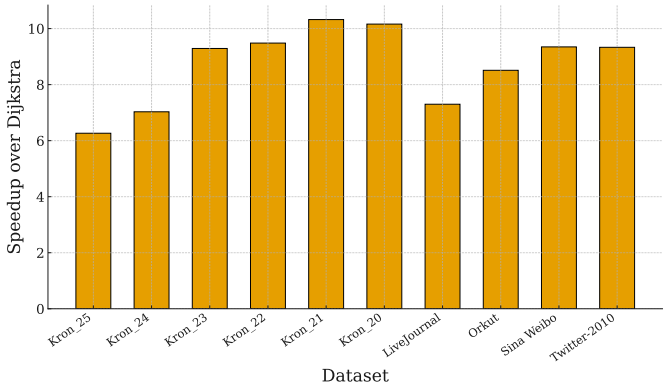

Fig. 1. Speedup of the distributed approach (16 processes) relative to sequential Dijkstra. Synthetic graphs achieve $6\times$–$10\times$ speedup, with peak performance on larger Kronecker inputs. Real-world graphs show $7\times$–$9.4\times$ speedup, demonstrating robust gains despite irregular topologies.

Scalability Challenges in Large Graphs. Traditional sequential algorithms such as Dijkstra's algorithm [12] become impractical for modern graph datasets that may contain billions of vertices and edges. Consequently, scalable parallel and distributed implementations are required to process such massive graphs efficiently. In contemporary scientific and engineering applications, graph instances frequently exceed the memory capacity of a single compute node [3], [17], making distributed-memory execution unavoidable. In this setting, graph data must be partitioned across multiple nodes, and algorithmic progress depends on communication among processors. As a result, global synchronization and communication overhead can dominate end-to-end runtime. At the scale of modern scientific simulations, even small inefficiencies in communication can propagate into substantial performance bottlenecks, limiting problem resolution and delaying downstream analysis. These challenges highlight the need for distributed SSSP algorithms that maximize parallelism while minimizing communication and synchronization overhead.

Δ-Stepping for Parallel SSSP. Among parallel approaches for graphs with non-negative edge weights, the Δ-stepping algorithm [24] is widely adopted due to its balance between work efficiency and parallelism. Instead of enforcing the strict global ordering required by Dijkstra's algorithm, Δ-stepping groups vertices into distance-based buckets and relaxes them in batches, enabling concurrent processing of vertices within the same bucket while limiting redundant work. This design

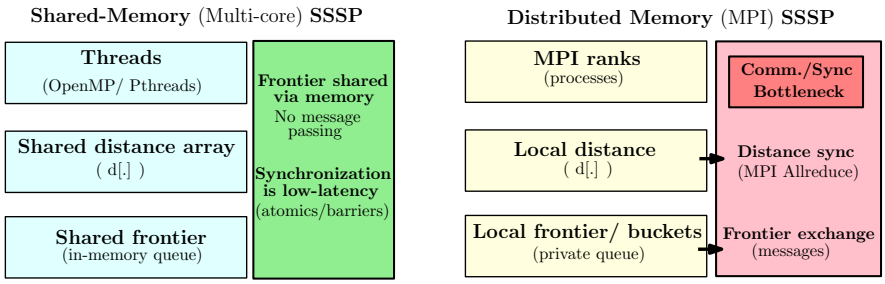

Fig. 2. Comparison between shared and distributed-memory SSSP models. Shared-memory execution relies on implicit coordination through a common address space, whereas distributed-memory execution requires explicit message passing and global synchronization to exchange frontiers and distance updates, leading to increased communication overhead at scale.

exposes substantial parallelism and has made Δ-stepping a practical foundation for scalable SSSP implementations across shared-memory systems [18], [31], GPUs [11], [33], and distributed platforms [26], [32].

Achieving scalable performance for SSSP requires careful consideration of the underlying hardware architecture, as communication and synchronization costs often determine scalability [13], [29]. In shared-memory systems, threads operate within a single address space and benefit from low-latency memory access and efficient synchronization primitives. As illustrated in Fig. 2 (left), frontier management is largely implicit, with coordination achieved through mechanisms such as atomic operations and lightweight barriers, enabling efficient fine-grained parallelism and strong performance on multi-core architectures. In contrast, distributed-memory systems require explicit partitioning of graph data across nodes, and relaxations involving remote vertices necessitate message exchanges across the network, introducing communication latency and synchronization overhead. While distributed execution is often required due to memory limitations [4], it is also driven by the architectural evolution of modern high-performance computing (HPC) systems, where clusters of compute nodes collectively provide the resources needed for extreme-scale graph analytics. In such environments, communication management becomes a central challenge for scalable distributed SSSP implementations.

Communication Bottlenecks in Distributed Execution. In distributed-memory systems, processes operate in private memory spaces and must coordinate progress through explicit message passing [1]. As illustrated in Fig. 2 (right), each process maintains local distance and frontier information for its partition of the graph, while updates to vertices residing on remote processes require communication across the network. Consequently, distributed SSSP execution involves frequent frontier exchanges and distance updates among processes, typically accompanied by iteration-level synchronization to maintain algorithmic correctness. Compared with the low-latency primitives available in shared-memory environments, these collective communication and synchronization steps introduce substantially higher overhead. As system scale increases, the cost of frontier exchange and distance synchronization can quickly dominate the overall runtime, becoming a primary barrier to scalability.

Although Δ-stepping provides substantial algorithmic parallelism, many existing distributed implementations incur significant communication costs due to frequent global coordination and irregular message patterns. These challenges are further amplified in large-scale HPC environments, where network latency and synchronization overhead often limit achievable performance. Addressing these issues requires algorithmic designs that reduce communication frequency and shift a larger portion of the workload toward local computation.

Our Approach. This paper presents a communication-aware MPI-based Δ-stepping algorithm designed to mitigate synchronization bottlenecks in distributed SSSP. Our approach reduces global coordination by emphasizing locally driven relaxation and carefully managing frontier updates, thereby decreasing communication overhead while preserving the parallel efficiency of the underlying Δ-stepping framework. The key contributions of this paper are as follows.

- We present a communication-aware distributed implementation of the Δ-stepping algorithm that avoids explicit frontier exchange, addressing a major scalability bottleneck in distributed SSSP.
- We introduce a distance-only synchronization mechanism based on `MPI_Allreduce`, enabling local frontier reconstruction and reducing global synchronization to a single collective operation per iteration.
- We demonstrate the effectiveness of the proposed approach through an extensive experimental evaluation on large synthetic and real-world graphs, achieving up to $10\times$ speedup (see Fig. 1) over sequential Dijkstra's algorithm.

II. Background and Related Work

The problem of computing the SSSP has been extensively studied in both sequential and parallel computing paradigms. The most popular Dijkstra's algorithm [12] provides an optimal shortest path solution for graphs with non-negative weights, but is inherently sequential due to its priority queue operations. To address this limitation, parallel approaches such as Bellman-Ford [18] and Δ-stepping [24] have been developed to improve computational efficiency. The original Bellman-Ford algorithm [7] is well suited for graphs with negative edge weights. In its parallel implementation, the algorithm relaxes all edges simultaneously in each iteration, making it well-suited for architectures that allow bulk synchronous processing. Busato and Bombieri [8] implemented a GPU-based parallel Bellman-Ford algorithm, demonstrating substantial speedups over the CPU version. However, recent studies [11], [18] suggest Bellman-Ford's inefficiencies in

parallel systems, noting that its performance is significantly impacted by excessive and unnecessary relaxations.

The Δ-stepping algorithm, introduced by Meyer and Sanders [24], offers a highly parallelizable approach by relaxing edges in buckets defined by a parameter Δ. This approach allows better relaxation scheme and reduces synchronization overhead. Many shared-memory evaluations [10], [14], [18], [23], [31] have confirmed that bucket-based approaches significantly reduce redundant relaxations while exposing substantial parallelism on multi-core CPUs. Recent advancements have also successfully adapted these techniques to massively parallel GPU architectures [11], [30], [33]. In terms of benchmarking, GAP [6] provides standardized methodologies to compare the performance of the graph kernel, focusing on robustness between graph structures.

However, distributed-memory environment introduces a different set of challenges for processing graph algorithms [28]. Hoefler et al. [20] demonstrated that communication noise and synchronization costs strongly influence scalability. They demonstrate that even small variations in communication latency can significantly impact scalability in large parallel systems. Similarly, [32] shows that distributed graph algorithms are often dominated by communication rather than computation, making communication aware strategies essential for better performance. These challenges are amplified in SSSP workloads because distance updates must be propagated across partition boundaries. Hence, comparatively fewer studies have been conducted on SSSP for distributed systems. Distributed graph frameworks such as PowerGraph [15], further highlight the need for efficient graph partitioning strategies that preserve locality. In general, previous studies identify three key factors that influence parallel SSSP performance: minimizing redundant relaxations, reducing communication and synchronization overhead, and exploiting hardware parallelism. These insights motivate our communication-efficient distributed SSSP design.

III. Distributed Δ-stepping Overview and Bottlenecks

Given a weighted directed graph $G = (V, E, w)$ with non-negative edge weights $w(u, v) \geq 0$ and a source vertex $s \in V$, the SSSP problem computes the minimum distance $d(v)$ from s to every vertex $v \in V$.

A. Δ-stepping Algorithm

The Δ-Stepping algorithm organizes vertices into buckets $B[i]$ according to their tentative distances, where bucket i contains vertices with distances in the range $[i\Delta, (i+1)\Delta)$. The algorithm repeatedly processes the smallest non-empty bucket by relaxing outgoing edges and updating tentative distances. To improve efficiency, edges are commonly categorized as *light* edges ($w < \Delta$) and *heavy* edges ($w \geq \Delta$), allowing repeated relaxation of light edges within the same bucket while heavy edges are relaxed only once. This bucket-based relaxation strategy exposes parallelism while avoiding the strict ordering constraints of Dijkstra's algorithm.

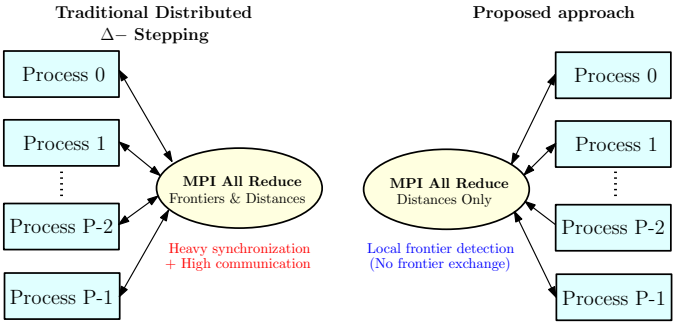

Fig. 3. Comparison of communication patterns between traditional distributed Δ-stepping and the proposed optimization. The traditional approach performs collective communication for both frontier and distance updates, while our scheme uses a single distance-only `MPI_Allreduce` per iteration and reconstructs frontiers locally.

B. Distributed Execution Model

In a distributed-memory MPI setting, the set of vertices is partitioned across P processes such that $V = \bigcup_{i=0}^{P-1} V_i$, where each process owns a subset of vertices and their outgoing adjacency lists. Each process performs relaxations on its local subgraph and participates in global synchronization to ensure consistency of tentative distance values. Depending on the implementation, the distance information may be partially or fully replicated across processes to facilitate relaxation and synchronization.

Key communication points. Traditional distributed Δ-stepping implementations involve two recurring communication steps: (i) exchanging newly activated vertices (frontiers) when relaxations affect vertices owned by remote processes, and (ii) merging tentative distance updates produced independently by different processes. Both steps typically require collective communication or fine-grained message passing, and introduce global synchronization. As the number of processes increases, these communication phases become increasingly expensive and can dominate execution time, particularly for irregular graphs with skewed degree distributions and rapidly changing frontier sizes. These observations highlight that the scalability of distributed Δ-stepping is often limited not by local computation, but by the frequency and cost of communication and synchronization. In particular, explicit frontier exchange introduces variable-size messaging and repeated global coordination, motivating designs that reduce or eliminate such communication when possible.

IV. Optimizing Distributed Δ-stepping

Conventional distributed Δ-stepping implementations propagate frontier updates between processes using collective operations or point-to-point messaging at each iteration, resulting in substantial communication and synchronization costs. To mitigate these costs, we introduce a communication-aware optimization that eliminates explicit frontier exchange (see Fig. 3). Instead of communicating active vertex sets, each process maintains two versions of the distance vector: one representing the distances from the previous iteration and the other representing updated distances through local relaxations

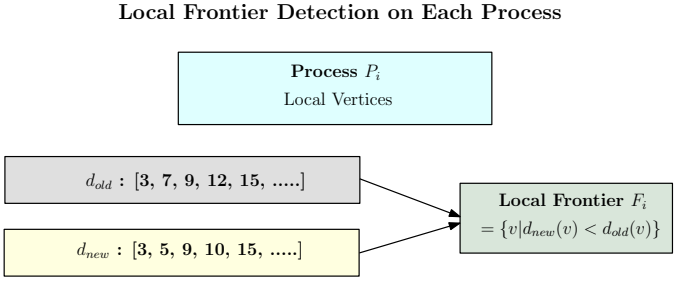

Fig. 4. Local frontier reconstruction after the global distance synchronization. Each process computes $F_i = \{v \mid d_{\text{new}}(v) < d_{\text{old}}(v)\}$ using only local distance arrays, eliminating explicit frontier exchange.

(see Fig. 4). At the end of each iteration, all processes participate in a single `MPI_Allreduce` operation on the distance vector using the minimum operator. This operation produces a globally consistent set of tentative distances. Following distance reduction, each process reconstructs its local frontier by identifying vertices whose distances have decreased relative to the previous iteration. Formally, a vertex $v \in V_i$ becomes active on process i whenever its tentative distance is improved during an iteration, i.e.,

$$d_{\text{new}}(v) < d_{\text{old}}(v)$$

This locally reconstructed frontier is then used for subsequent bucket processing, without requiring any explicit exchange of vertex identifiers between processes. By replacing explicit frontier communication with distance-only synchronization and local frontier reconstruction, the proposed approach reduces global communication to a fixed number of collective operations per iteration. This transformation converts an irregular communication-intensive messaging pattern into a predictable synchronization scheme that is easier to optimize and scale.

Communication Cost Perspective. We use the standard α–β communication model to reason about the cost of distributed synchronization, where α denotes the latency or startup cost of a communication operation and β denotes the per-element transmission cost. In traditional distributed Δ-stepping, each iteration may involve multiple communication events, including explicit frontier exchanges and repeated synchronization of distance updates. These operations incur a high latency cost due to frequent collective invocations and introduce additional bandwidth cost that scales with the size and variability of the frontier. In contrast, the proposed approach reduces communication to a fixed pattern consisting of a single distance-only `MPI_Allreduce` per iteration and a lightweight boolean reduction for convergence detection. Although reducing the full distance vector incurs a predictable bandwidth cost proportional to the number of vertices, it substantially lowers the latency-dominated cost by minimizing the number of global synchronization points. This trade-off is particularly beneficial for large-scale and irregular graphs, where frontier sizes fluctuate significantly.

V. Methodology and Implementation Details

A. Graph Representation and Data Distribution

We represent the input graph using a compressed sparse row (CSR) layout on each MPI rank, storing only the vertices owned by that rank and their outgoing adjacency lists. The global vertex set V is partitioned by vertex identifiers into contiguous ranges, such that rank i owns the vertex subset

$$V_i = \{\, v \mid \ell_i \le v < r_i \,\},$$

where ℓ_i and r_i denote the start and end indices of the partition. This vertex-range mapping enables constant-time ownership checks during relaxation and avoids the need for additional indirection or lookup structures.

Graph partitioning is performed by the master rank, which reads the full edge list and determines contiguous vertex-ID boundaries using a prefix-sum over vertex degrees to approximately balance the number of edges per rank. We adopt this centralized loader for simplicity and reproducibility; graph loading and distribution are excluded from the reported runtimes since our evaluation focuses on the scalability of the SSSP computation kernel. Each process then receives and stores its local subgraph in CSR format.

For distance management, we maintain two arrays, d_{old} and d_{new}, to support local frontier reconstruction across iterations. In our implementation, each rank stores distance values for all vertices, enabling a straightforward application of `MPI_Allreduce` over a full distance vector. This design choice prioritizes simplicity, portability, and clarity of communication patterns. While this incurs additional memory overhead, it avoids complex ownership-aware reductions and allows us to isolate the impact of synchronization overhead.

B. Bucket Structure and Relaxation Strategy

The algorithm follows the Δ-stepping paradigm, organizing tentative distances into buckets of fixed width Δ. A vertex v with tentative distance $d(v)$ is assigned to a bucket indexed $\lfloor d(v)/\Delta \rfloor$. Each rank maintains its own local bucket structure, which stores vertices owned by that rank. During execution, the algorithm processes the buckets in increasing order of distance. For a given bucket, relaxations are performed iteratively to capture cascading improvements. Light edges are repeatedly relaxed until no further distance reductions occur within the current bucket, ensuring local closure. Meanwhile, the heavy edges are processed once. When a relaxation targets a vertex owned by the same rank, the update is immediately incorporated into local buckets. When a relaxation targets a vertex owned by another rank, the update is recorded in the distance array and incorporated after global synchronization.

C. MPI Synchronization and Local Frontier Reconstruction

After local relaxations for an iteration, all ranks participate in a distance-only synchronization step using `MPI_Allreduce` with the minimum operator. This collective operation merges tentative distances across ranks and produces a globally consistent distance vector. Unlike traditional distributed SSSP implementations, we do not exchange

Algorithm 1: Optimized distributed Δ-Stepping with distance-only synchronization.

Input : Local CSR graph (V_i, E_i, w), source vertex s, bucket width Δ, MPI communicator
Output: Shortest-path distances $d(v)$ for all $v \in V$
Initialize $d_{\text{old}}(v) \leftarrow \infty$, $d_{\text{new}}(v) \leftarrow \infty$ for all $v \in V$ and $F_i \leftarrow \emptyset$;
if $s \in V_i$ **then**
  $d_{\text{new}}(s) \leftarrow 0$;
while *true* **do**
  `// Identify current bucket`
  Determine the smallest bucket index k such that $\exists v \in V_i$ with $d_{\text{new}}(v) \in [k\Delta, (k+1)\Delta)$;
  `// Construct local frontier for current bucket` k
  $F_{curr} \leftarrow \{\, v \in V_i \mid d_{\text{new}}(v) \in [k\Delta, (k+1)\Delta) \,\}$;
  `// Local` Δ`-stepping relaxations`
  while $F_{curr} \neq \emptyset$ **do**
    $F_{curr}, F_i \leftarrow \textsc{Relax}(F_{curr}, F_i, k, d_{\text{new}}, \Delta)$;
  `// Distance-only global sync.`
  `MPI_Allreduce(`d_{new}, MIN);
  `// Frontier reconstruction`
  $F_i \leftarrow F_i \cup \{\, v \in V_i \mid d_{\text{new}}(v) < d_{\text{old}}(v) \}$;
  `// Global termination check`
  $frontierNonEmpty_i \leftarrow (|F_i| > 0)$;
  `MPI_Allreduce(`$frontierNonEmpty_i$, OR) $\rightarrow anyActive$;
  if $anyActive =$ ***false*** **then**
    break
  $d_{\text{old}} \leftarrow d_{\text{new}}$;

Algorithm 2: Local edge relaxation and same-bucket closure on rank i.

Input : Current frontier F_{curr}, Frontier set F_i, bucket index k, distances d_{new}, bucket width Δ
Output: Updated Current Frontier F'_{curr}, Updated frontier set F'_i
$F'_i \leftarrow \{F_i - F_{curr}\}$ and $F'_{curr} \leftarrow \emptyset$
foreach $u \in F_{curr}$ **do**
  foreach $(u, v) \in E_i$ **do**
    $cand \leftarrow d_{\text{new}}(u) + w(u, v)$;
    if $cand < d_{\text{new}}(v)$ **then**
      $d_{\text{new}}(v) \leftarrow cand$;
      if $d_{\text{new}}(v) < k\Delta$ & $v \in V_i$ **then**
        $F'_{curr} \leftarrow F'_{curr} \cup \{v\}$;
      else
        $F'_i \leftarrow F'_i \cup \{v\}$;
return F'_{curr}, F'_i;

explicit frontier sets or active vertex lists. Instead, each rank reconstructs its local frontier by comparing updated distances with distances from the previous iteration. Formally, the local frontier of rank i is defined as:

$$F_i = \{\, v \in V_i \mid d_{\text{new}}(v) < d_{\text{old}}(v) \,\}.$$

Only vertices whose distances have strictly decreased are reinserted into the frontier set and are considered active in subsequent relaxations. Global convergence is detected using a boolean flag that indicates whether any rank has a non-empty frontier. Each rank computes a local activity flag and participates in a second `MPI_Allreduce` using a logical OR operator. If no rank reports activity, the algorithm terminates.

D. Algorithms

Algorithm 1 summarizes the high-level structure of our optimized distributed Δ-stepping algorithm, abstracting away implementation-specific details while making the main control flow explicit. The algorithm operates on the local subgraph (V_i, E_i) owned by each MPI process and maintains two distance vectors, d_{old} and d_{new}, which store tentative shortest-path estimates from the source. The procedure iterates over buckets parameterized by the bucket width Δ, performing local relaxations on vertices whose tentative distances fall within the current active bucket. The global consistency of distance values is maintained via a distance-only MPI_Allreduce with the minimum operator. Following this synchronization, each process checks for the size of its frontier and sets a boolean activity flag based on non-empty frontier accordingly. A second collective reduction using logical OR determines whether any process remains active.

Algorithm 2 details the local relaxation kernel applied to the active vertex set on each process. For each active vertex $u \in F_{curr}$, the kernel iterates over the outgoing edges $(u, v) \in E_i$ and computes a candidate distance $cand$. If the candidate improves the current tentative distance of v, the distance vector and the frontier set are updated. The vertex is scheduled for further update if it falls into the current bucket again.

E. Complexity Discussion

Let $|E_i|$ denote the number of edges owned by rank i. The local computational work per iteration is proportional to the number of edges incident to active vertices, i.e., $O(|E_i(F_i)|)$. The proposed optimization introduces an additional local scan over V_i to reconstruct the frontier, which is linear in the number of owned vertices and typically inexpensive compared to communication costs. The global communication cost per iteration consists of a single `MPI_Allreduce` over the distance vector and one boolean reduction for convergence detection. By eliminating explicit frontier exchange and variable-sized messaging, the algorithm avoids communication patterns that are sensitive to frontier size and graph irregularity. The primary trade-off is the cost of reducing a full distance vector, which motivates future work on sparse or ownership-aware distance synchronization. Nevertheless, the current design provides a clear and effective demonstration of how communication-

aware synchronization can substantially improve scalability in distributed SSSP.

VI. Experimental Setup

A. Platform and Software

All experiments were conducted on a distributed-memory high-performance computing cluster using MPI. Each compute node consists of multi-core CPUs connected via a high-speed interconnect and is equipped with 192 GB of main memory. This memory capacity is sufficient to support large-scale graph processing, including billion-edge inputs and replicated distance storage across MPI ranks. The implementation was compiled using an optimized MPI library to ensure the performance measurements reflect algorithmic behavior rather than software overhead. Unless otherwise stated, all reported runtimes exclude graph loading, partitioning, and pre-processing costs.

B. Parameters and Baselines

We evaluate the proposed distributed Δ-stepping implementation using $P \in \{1, 2, 4, 8, 16\}$ MPI processes to study strong-scaling behavior. Speedup is reported relative to the single-process execution, which serves as the distributed baseline. In addition, we compare the best achieved parallel runtime against a sequential Dijkstra implementation to quantify the performance gains obtained from parallelism. The bucket width Δ is fixed across all experiments to ensure a fair comparison between process counts and datasets. While the choice of Δ can influence performance, tuning it adaptively is orthogonal to the communication optimization studied in this work. We therefore treat Δ sensitivity as a limitation and leave systematic exploration of adaptive or graph-aware Δ selection as future work.

C. Datasets

We evaluate performance on a diverse collection of synthetic and real-world graphs, including inputs with up to **billions of edges**. Real-world graphs include large-scale social and information networks that typically exhibit power-law degree distributions, high variance in vertex degrees, and irregular connectivity patterns. Synthetic Kronecker graphs are also used to study scaling behavior as we can generate scale-free large graphs with properties like real world graphs. Edge weights are ranged [1,1000]. The characteristics of the data set, including the size of the graph and the average degree, are summarized in Table I.

VII. Results

In this section, we evaluate the performance of our parallel SSSP implementation on large scale synthetic and real-world graphs. For each graph, we measure the runtime of our parallel implementation on $P \in \{1, 2, 4, 8, 16\}$ processes and compute speedups relative to the 1-process execution as well as the sequential Dijkstra baseline.

TABLE I
Dataset Description

Type	Dataset	#Nodes	#Edges	Avg Deg
Synthetic	Kron_25	33,554,432	1,073,741,824	32
	Kron_24	16,777,216	536,870,912	32
	Kron_23	8,388,608	268,435,456	32
	Kron_22	4,194,304	134,217,728	32
	Kron_21	2,097,152	67,108,864	32
	Kron_20	1,048,576	33,554,432	32
Social	LiveJournal	4,036,538	69,362,378	17
	Orkut	3,072,626	234,370,166	76
	Sina Weibo	58,655,849	522,642,142	8
	Twitter-2010	21,297,772	795,077,428	37

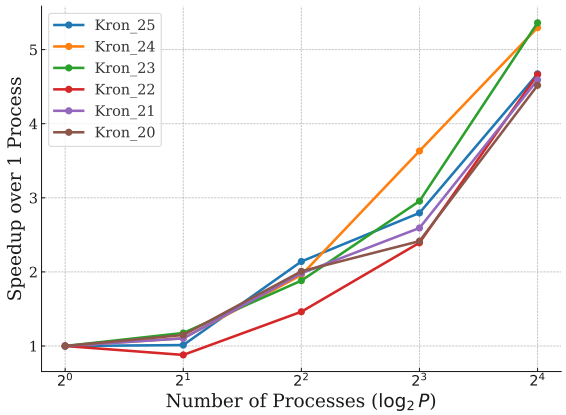

Fig. 5. Strong scaling results for Kronecker graphs. Results show near-linear scaling across all scales, with $4.5\times$ to $5.3\times$ speedup at $P = 16$ compared to the parallel baseline.

A. Scaling on Synthetic Kronecker Graphs

The speedup results for the synthetic Kronecker graphs (Kron_20–Kron_25) are shown in Fig. 5. To highlight the effect of doubling the number of processes, the x-axis is expressed as $\log_2 P$. All synthetic graphs demonstrate strong and consistent speedup up to 16 processes. The largest graph, Kron_25, achieves a speedup of approximately $4.7\times$ over the 1-process runtime, while the smaller graphs achieve speedups between $4.5\times$ and $5.4\times$. This behavior is expected due to the similar structure of Kronecker graphs. The corresponding runtime plot, shown in Fig. 6, uses a log-scale y-axis to provide a clear view of scaling trends. Runtime decreases steadily as the number of processes increases. When comparing against the sequential Dijkstra baseline (Fig. 1) using the best parallel runtime per graph, synthetic datasets achieve speedups ranging from $6.3\times$ to $10.3\times$.

B. Scaling on Real-World Social Networks

Fig. 7 and Fig. 8 show the scaling behavior for real-world networks. These graphs are structurally different from the Kronecker graphs, with heavy-tailed degree distributions, community clusters, and highly skewed frontier sizes that affect load-balance and synchronization.

The LiveJournal graph exhibits modest scaling, reaching only approximately $1.3\times$ speedup at 16 processes. This is consistent with its relatively small and irregular frontier sets that

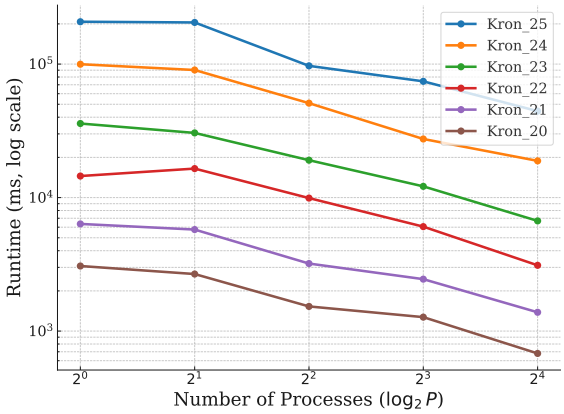

Fig. 6. Runtime vs. number of processes for synthetic Kronecker graphs (log-scale). The consistent downward trend demonstrates strong parallel efficiency.

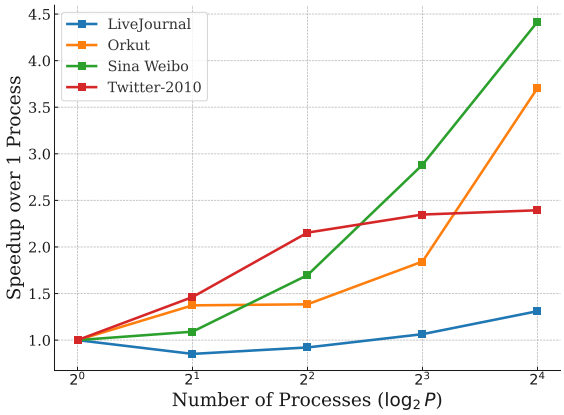

Fig. 7. Strong scaling speedup on real-world social grpahs. While smaller datasets (e.g., LiveJournal) exhibit earlier speedup saturation due to limited local work, larger networks show a strong upward trajectory.

limit parallelism. In contrast, denser and larger networks such as Orkut and Sina Weibo achieve significantly better scaling: $3.7\times$ and $4.4\times$ speedup at 16 processes, respectively. Twitter-2010 shows moderate scaling, reaching about $2.4\times$ speedup. The log-scale runtime curves in Fig. 8 reveal a consistent downward trend for all datasets. However, diminishing returns appear earlier than in the synthetic graphs, typically after 8 processes. Nevertheless, parallel implementation still produces meaningful speedups.

C. Comparison with Dijkstra's Algorithm

To understand the benefit of our proposed distributed SSSP approach over traditional sequential methods, we compare its performance directly against Dijkstra's algorithm across all datasets. Fig. 9 presents a log-scale runtime comparison between Dijkstra's algorithm and the proposed distributed approach. Across all synthetic Kronecker graphs, the proposed method consistently reduces runtime by more than an order of magnitude for the larger inputs (e.g., Kron_23–Kron_25). Real-world social networks also benefit substantially: Sina Weibo and Twitter-2010 show large reductions in runtime,

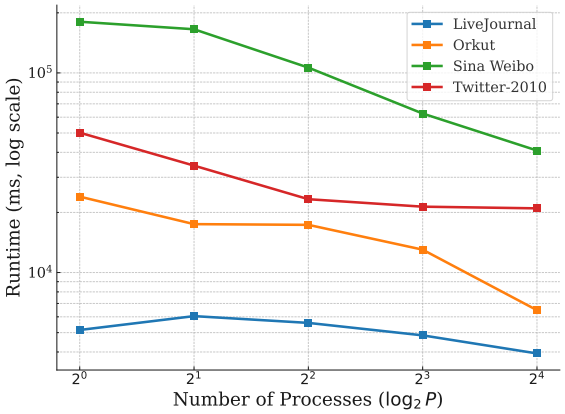

Fig. 8. Runtime vs. number of processes for real-world social networks. Despite irregular structure, all datasets show measurable runtime reduction with increasing processes.

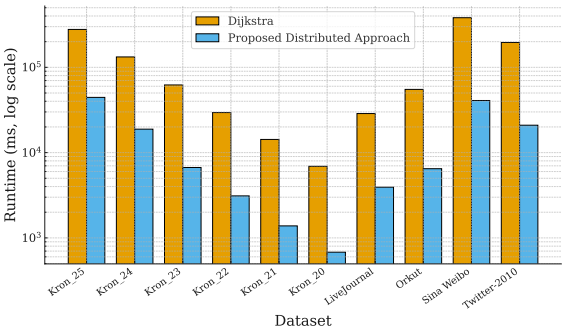

Fig. 9. Performance comparison between sequential Dijkstra and the proposed distributed Δ-stepping implementation (log-scale). Our approach consistently achieves an order-of-magnitude reduction in runtime across both synthetic and real-world datasets.

while LiveJournal exhibits more modest improvement due to its smaller frontiers and limited available parallelism.

Overall, these results highlight the effectiveness of the proposed distributed SSSP approach in outperforming classical sequential algorithms across both synthetic graphs and real-world networks. When comparing the best achieved parallel runtime against the Dijkstra baseline, social networks obtain between $7.3\times$ and $9.3\times$ speedup. The substantial reductions in runtime and consistently large speedups confirm the practical advantage of our method for large-scale graph analytics workloads. Table II shows the runtimes and speedups of our implemented algorithm.

D. Communication and Scalability Discussion

The scalability behavior observed in our experiments is strongly influenced by both the structural properties of the input graphs and the communication patterns induced by the distributed Δ-stepping algorithm. Although synthetic Kronecker graphs and real-world networks share a power-law degree distribution, they differ substantially in higher-order characteristics such as community structure, clustering, and edge locality. Kronecker graphs primarily model global degree

TABLE II
RUNTIME (MILLISECONDS) AND SPEEDUP OF PROPOSED DISTRIBUTED Δ-STEPPING IMPLEMENTATION

Type	Dataset	Dijkstra	Runtime of Distributed Δ-Stepping (ms)					Min	Speedup	Speedup
		Runtime (ms)	P=1	2	4	8	16	Runtime (ms)	(vs P=1)	(vs Dijkstra)
Synthetic	Kron_25	278318	207595	205014	96985	74258	44425	44425	4.67	6.26
	Kron_24	132434	99776	90287	50968	27474	18838	18838	5.30	7.03
	Kron_23	62153	35868	30506	19059	12136	6690	6690	5.36	9.29
	Kron_22	29482	14499	16487	9916	6058	3109	3109	4.66	9.48
	Kron_21	14275	6349	5760	3205	2449	1383	1383	4.59	10.32
	Kron_20	6909	3072	2675	1530	1272	680	680	4.52	10.16
Social	LiveJournal	28712	5155	6049	5594	4845	3933	3933	1.31	7.30
	Orkut	55127	23999	17476	17333	13018	6477	6477	3.71	8.51
	Sina Weibo	381829	180448	165442	106291	62594	40853	40853	4.42	9.35
	Twitter-2010	195650	50190	34324	23300	21376	20966	20966	2.39	9.33

heterogeneity and hub formation but lack tightly connected communities, resulting in smoother and more predictable frontier expansion. This behavior promotes relatively balanced relaxation workloads across MPI ranks and enables more stable scaling trends as the number of processes increases. In contrast, real-world graphs exhibit pronounced community structure and highly irregular connectivity patterns. Frontier growth in such graphs is often uneven, with bursts of activity concentrated within dense subgraphs or around high-degree hub vertices. This causes load imbalance, where some ranks process significantly more relaxations than others. As process counts increase, this imbalance becomes more severe, limiting parallel efficiency despite available computational resources. Hence, though proposed distance-only collective strategy mitigates communication cost significantly by eliminating explicit frontier transmission, diminishing returns still appear at higher process counts. This effect is particularly pronounced for smaller graphs, such as LiveJournal, where the computation-to-communication ratio becomes unfavorable beyond moderate concurrency levels.

E. Comparison with Existing SSSP Approaches

Table III provides a qualitative comparison of the proposed algorithm against representative SSSP implementations across different architectures. We intentionally omit direct runtime comparisons with shared-memory (e.g., Wasp [10]) and GPU systems (e.g., Wang et al. [30], Gunrock [11]) because fundamental differences in memory hierarchies and synchronization costs render cross-platform GTEPS metrics misleading for weighted SSSP. While these single-node frameworks achieve high throughput, they are fundamentally constrained by local memory capacity. Among distributed implementations, our work occupies a distinct design point compared to extreme-scale hybrid approaches like Chakaravarthy et al. [9]. While the latter optimizes communication through hybrid Δ-stepping/Bellman-Ford (BF) strategies and direction optimization, our approach eliminates explicit frontier exchange entirely.

F. Limitations

The proposed method still relies on collective synchronization, which may introduce latency at very large MPI process counts. Performance can also depend on graph structure and the choice of the Δ parameter, which is fixed in the current implementation. Finally, the evaluation focuses on medium-scale clusters, and scalability characteristics may evolve at extreme scales.

VIII. CONCLUSION AND FUTURE WORK

This paper presents a communication-aware MPI-based distributed Δ-stepping algorithm for SSSP computation. The key contribution of our approach is the elimination of explicit frontier exchange, replacing it with a distance-only collective communication scheme. By reducing each iteration to a single `MPI_Allreduce` operation combined with local frontier reconstruction, the proposed design significantly reduces communication volume and simplifies synchronization patterns. Comprehensive experimental evaluation on both synthetic and real-world networks demonstrates the effectiveness of this approach. Synthetic graphs exhibit stable scaling behavior, while real-world graphs, despite their irregular structure and community-driven locality, still achieve meaningful performance gains. Across all evaluated datasets, the proposed implementation consistently outperforms the classical sequential Dijkstra baseline and achieves speedups of up to $10\times$. These results confirm that communication-efficient distributed SSSP can provide substantial benefits even in the presence of load imbalance and irregular frontier dynamics.

Future work will focus on improving scalability by overlapping collective communication with computation and implementing adaptive parameter selection based on graph topology. Additionally, investigating sparse, ownership-aware reduction strategies and hybrid MPI+threading models could further minimize communication overhead on large-scale clusters.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation (NSF) under Award Number 2323533.

REFERENCES

[1] M. Alam, S. Arifuzzaman, H. Bhuiyan, M. Khan, V. A. Kumar, and M. V. Marathe. Distributed memory parallel algorithms for massive graphs. In *Massive Graph Analytics*, pages 85–107. Chapman and Hall/CRC, 2022.

TABLE III
QUALITATIVE COMPARISON OF REPRESENTATIVE SSSP IMPLEMENTATIONS

System	Architecture	Approach	Comm. Model	Scalability	Primary Goal
Proposed work	Distributed (MPI)	Δ-Stepping	**Distance-only collectives**	Large-scale Cluster	**Comm. efficiency**
Scalable SSSP [9]	Distributed (MPI)	Hybrid Δ stepping + BF	Mixed collective/P2P	Extreme-scale	Raw throughput
Wasp [10]	Multicore CPU	Asynchronous Stealing	Shared-memory/Atomics	Single node (128+ threads)	Work efficiency
Wang et al. [30]	GPU	Work-Efficient SSSP	Parallel Priority Queue	Single/Multi-GPU	Scheduling quality
Gunrock [11]	GPU	Frontier-based Δ stepping	Kernel-driven	Single/Multi-GPU	Throughput

[2] N. M. Amato, K. A. Dill, and G. Song. Using motion planning to map protein folding landscapes and analyze folding kinetics of known native structures. In *Proceedings of the sixth annual international conference on Computational biology*, pages 2–11, 2002.

[3] S. Arifuzzaman, H. S. Arikan, M. Faysal, M. Bremer, J. Shalf, and D. Popovici. Unlocking the potential: Performance portability of graph algorithms on kokkos framework. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 526–529. IEEE, 2024.

[4] S. Arifuzzaman, M. Khan, and M. Marathe. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(1):1–34, 2019.

[5] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm engineering: Selected results and surveys*, pages 19–80. Springer, 2016.

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

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

[8] F. Busato and N. Bombieri. An efficient implementation of the bellman-ford algorithm for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233, 2015.

[9] V. T. Chakaravarthy, F. Checconi, P. Murali, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[10] M. D'Antonio, T. S. Mai, P. Tsigas, and H. Vandierendonck. Wasp: Efficient asynchronous single-source shortest path on multicore systems via work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 2109–2125, 2025.

[11] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, 2014.

[12] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[13] M. A. M. Faysal, M. Bremer, C. Chan, J. Shalf, and S. Arifuzzaman. Fast parallel index construction for efficient k-truss-based local community detection in large graphs. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 132–141, New York, NY, USA, 2023. Association for Computing Machinery.

[14] X. Gan, Q. Tang, F. Xiong, S. Li, B. Yang, and T. Li. Tianhestar: Orchestrating sssp applications on tianhe supercomputer. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 534–542. IEEE, 2024.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.

[16] R. Guérin and A. Orda. Computing shortest paths for any number of hops. *IEEE/ACM transactions on networking*, 10(5):613–620, 2002.

[17] T. Harris. Hardware trends: Challenges and opportunities in distributed computing. *ACM SIGACT News*, 46(2):89–95, 2015.

[18] R. Hassan and S. Arifuzzaman. An efficient multi-core parallel implementation of sssp algorithm with decreasing delta-stepping. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2024.

[19] T. Hoefler, T. Schneider, and A. Lumsdaine. Optimized routing for large-scale infiniband networks. In *2009 17th IEEE Symposium on High Performance Interconnects*, pages 103–111. IEEE, 2009.

[20] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[21] M. Lambertini, M. Magnani, M. Marzolla, D. Montesi, and C. Paolino. Large-scale social network analysis. In *Large-scale data analytics*, pages 155–187. Springer, 2013.

[22] J. Macfarlane and B. Xu. Temporal sampling constraints for geospatial path reconstruction in a transportation network. In *Proceedings of the 10th ACM SIGSPATIAL Workshop on Computational Transportation Science*, pages 1–6, 2017.

[23] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel shortest path algorithms for solving large-scale instances. In *The Shortest Path Problem*, pages 249–290, 2006.

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

[25] J. Oyelade, I. Isewon, O. Aromolaran, E. Uwoghiren, T. Dokunmu, S. Rotimi, O. Aworunse, O. Obembe, and E. Adebiyi. Computational identification of metabolic pathways of plasmodium falciparum using the k-shortest path algorithm. *International Journal of Genomics*, 2019(1):1750291, 2019.

[26] T. Panitanarak. Scalable single-source shortest path algorithms on distributed memory systems. In *International Conference on Soft Computing in Data Science*, pages 19–33. Springer, 2018.

[27] F. K. A. Salem, M. Jaber, C. Abdallah, O. Mehio, and S. Najem. A distributed spatiotemporal contingency analysis for the lebanese power grid. *IEEE Transactions on Computational Social Systems*, 6(1):162–175, 2019.

[28] N. S. Sattar and S. Arifuzzaman. Parallelizing louvain algorithm: Distributed memory challenges. In *2018 IEEE Intl Conf on Dependable, Autonomic and Secure Computing (DASC 2018)*, pages 695–701. IEEE, 2018.

[29] N. S. Sattar, K. Z. Ibrahim, A. Buluc, and S. Arifuzzaman. Dygdpcd: A distributed parallel community detection algorithm for large-scale dynamic graphs. *International Journal of Parallel Programming*, 53(1):4, 2025.

[30] K. Wang, D. Fussell, and C. Lin. A fast work-efficient sssp algorithm for gpus. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 223–237, 2021.

[31] Y. Wang, H. Cao, Z. Ma, W. Yin, and W. Chen. Scaling graph 500 sssp to 140 trillion edges with over 40 million cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[32] H. Zhang, J. Xie, and X. Zhang. Communication-efficient-stepping for distributed computing systems. In *2023 19th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 369–374. IEEE, 2023.

[33] Y. Zhang, H. Cao, J. Zhang, Y. Sun, M. Dun, J. Huang, X. An, and X. Ye. A bucket-aware asynchronous single-source shortest path algorithm on gpu. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, pages 50–60, 2023.

[34] B. Zhao, J. Zhong, B. He, Q. Luo, W. Fang, and N. K. Govindaraju. Gpu-accelerated cloud computing for data-intensive applications. In *Cloud Computing for Data-Intensive Applications*, pages 105–129. Springer, 2014.