HAD-SSSP: A Hierarchical Adaptive Distributed Single-Source Shortest Path Algorithm

Krishang Sharma
Shiv Nadar Institution of Eminence, Delhi-NCR, India

***Abstract*—The Single-Source Shortest Path (SSSP) problem remains a fundamental challenge in distributed computing, particularly for large-scale graphs with diverse topologies. We present HAD-SSSP (Hierarchical Adaptive Distributed SSSP), a novel algorithm that combines multi-level hierarchical clustering with distributed asynchronous computation to achieve superior performance across varied graph structures. Our approach constructs a four-level hierarchy (L0-L3) using adaptive spatial clustering, enabling fast approximate distance estimation via precomputed supernode distances. We implement exact distributed SSSP using an owner-push message-passing model with hybrid MPI+OpenMP parallelization. Experimental evaluation on three diverse datasets shows that HAD-SSSP achieves superior performance over state-of-the-art algorithms (Ghaffari-Li 2018, Ghaffari-Trygub 2024), outperforming them by up to $6.75\times$ on scale-free graphs while maintaining competitive performance on road networks. Our algorithm demonstrates strong scalability from 4 to 16 MPI processes, with distributed execution yielding up to $13\times$ improvement over sequential baselines. The hierarchical adaptive design enables HAD-SSSP to automatically tune clustering parameters based on graph characteristics, making it practical for real-world applications.**

***Index Terms*—Distributed algorithms, shortest path, hierarchical clustering, MPI, OpenMP, graph algorithms**

I. Introduction

The Single-Source Shortest Path (SSSP) problem is fundamental to distributed computing and network optimization. Computing exact shortest paths in large-scale distributed graphs remains challenging due to communication overhead, synchronization costs, and scalability constraints. Traditional approaches like Bellman-Ford [3] achieve optimal $O(n)$ time complexity but suffer from $\Theta(mn)$ message complexity, making them impractical for large networks.

Recent advances have explored approximate solutions with sublinear time complexity [4], [8] and exact solutions with improved bounds [1], [6], [7]. However, these approaches often face trade-offs between time complexity, message complexity, and practical scalability. Moreover, many state-of-the-art algorithms struggle with real-world graph characteristics such as varying topology (road networks vs. scale-free networks) and large-scale deployments.

We present HAD-SSSP (Hierarchical Adaptive Distributed SSSP), a novel algorithm that leverages multi-level hierarchical clustering to achieve near-optimal performance across diverse graph topologies. Our key contributions include:

- A four-level hierarchical clustering scheme (L0-L3) inspired by multilevel graph partitioning [9] that adapts to graph characteristics
- An owner-push distributed SSSP approach with bounded message complexity
- Hybrid computation combining approximate hierarchical estimates with exact distributed computation in the CONGEST model [7]
- Graph-adaptive parameters that optimize for road networks, scale-free graphs, and general topologies
- Implementation using hybrid MPI [11]+OpenMP [12] parallelization with efficient load balancing

HAD-SSSP is evaluated on three diverse datasets including synthetic citation graphs, real-world road networks (roadNet-CA), and Graph500 benchmarks [15], demonstrating practical scalability and efficiency.

II. Literature Review

A. Improved Distributed Algorithms for Exact Shortest Paths (Ghaffari & Li, 2017)

Ghaffari and Li [1] presented breakthrough results for exact SSSP in distributed settings. Their algorithm achieves $\tilde{O}(n^{3/4}D^{1/4})$ round complexity for polylogarithmic diameter graphs, improving upon Elkin's $\tilde{O}(n^{5/6})$ bound. The approach uses:

- **Virtual node sampling**: Nodes are sampled with probability k/n to create a virtual graph overlay
- **Scaling framework**: Edge weights are processed bit-by-bit to reduce maximum distance bounds
- **ShortRange algorithm**: Hybrid BFS-Bellman-Ford for bounded-hop, bounded-distance computation
- **Hierarchical distance computation**: Virtual SSSP on sampled nodes followed by extension to all nodes

For κ sources, they achieve $\tilde{O}(\min\{\kappa^{1/2}n^{3/4+o(1)} + \kappa^{1/3}n^{3/4}D^{1/6}, \kappa^{3/7}n^{6/7}\} + D)$ complexity. While theoretically optimal, the algorithm's complexity in handling directed graphs and virtual node coordination presents practical implementation challenges.

B. Near-Optimal Low-Energy Deterministic Distributed SSSP (Ghaffari & Trygub, 2024)

Ghaffari and Trygub [2] introduced energy-efficient distributed SSSP with $\tilde{O}(n)$ time and poly$(\log n)$ energy complexity per node. Their contributions include:

- **Distributed Dijkstra adaptation**: Recursive distance-based splitting using approximate cutters

- **Deterministic low-energy BFS**: Using neighborhood covers with coordinated wake/sleep scheduling
- **CSSP to APSP**: Extension to All-Pairs Shortest Paths via independent SSSP computations

The algorithm recursively partitions nodes by distance thresholds $d/2$, using $(1+\epsilon)$-approximate distances to guide computation. Maximal spanning forests coordinate synchronization without global broadcasts. While achieving near-optimal theoretical bounds, the deterministic construction of sparse neighborhood covers adds overhead, and the recursive structure requires careful implementation for practical systems.

C. Distributed Bellman-Ford Algorithm

The classical Bellman-Ford algorithm [3] provides the foundation for distributed SSSP. In each of $O(n)$ rounds, every node u updates its distance estimate: $\bar{d}(s,u) = \min_{v \in N(u)}(\bar{d}(s,v) + w(vu))$. While simple and correct, it suffers from:

- Message complexity: $O(mn)$ as every edge relaxes in every round
- Congestion: $\Omega(n)$ messages per edge
- Inability to exploit graph structure (diameter, sparsity)

D. Nanongkai's Approximation Framework (2014)

Nanongkai [4] provided $(1+\epsilon)$-approximate SSSP in $\tilde{O}(D+\sqrt{n})$ rounds with $\tilde{O}(m)$ messages, nearly matching lower bounds. The approach uses:

- Weight rounding to reduce the problem to $O(\log n)$ BFS instances
- Virtual graph construction with $O(n \log n)$ overlay edges
- Distributed (h,ℓ)-bounded SSSP subroutines

This work demonstrated that approximation can circumvent theoretical lower bounds for exact SSSP, influencing subsequent exact algorithms that use approximate distances as subroutines.

E. APSP Advances (Bernstein & Nanongkai, 2019)

Bernstein and Nanongkai [5] achieved $\tilde{O}(n)$ round complexity for exact All-Pairs Shortest Paths (APSP), resolving a long-standing open problem. Their techniques include:

- Scaling framework with asymmetric edge weights
- Random filtering to control congestion
- Hierarchical distance oracles
- Novel scheduling of SSSP computations

While this represents the state-of-the-art for APSP, the randomized components and complex data structures present barriers to deterministic and energy-efficient implementations.

F. Recent Advances in CONGEST Model

The CONGEST model, where each edge can transmit $O(\log n)$ bits per round, has seen significant recent progress. Chechik and Mukhtar [7] achieved $\tilde{O}(n\sqrt{D}+D)$ rounds for weighted directed graphs using novel weight-modifying techniques. Cao et al. [8] presented improved $(1+\epsilon)$-approximate SSSP with $\tilde{O}((\sqrt{n}+D)\log W/\epsilon^2)$ complexity, reducing polynomial factors across various network diameters.

These advances demonstrate the ongoing tension between theoretical optimality and practical implementation complexity, motivating our hierarchical adaptive approach.

G. Graph Partitioning and Clustering

Efficient graph partitioning is crucial for distributed graph algorithms. Karypis and Kumar [9] introduced multilevel graph partitioning using coarsening, partitioning, and refinement phases. Sanders and Schulz [10] extended this to distributed settings with evolutionary algorithms achieving high-quality partitions with minimal edge cuts.

Our hierarchical clustering approach adapts these multilevel principles to SSSP: the L0-L3 hierarchy mirrors the coarsening phase, while our owner-push message passing leverages partition locality to minimize inter-process communication.

H. Summary of State-of-the-Art

Current approaches face fundamental trade-offs:

- **Theoretical vs. Practical**: Many algorithms with optimal asymptotic bounds [1], [7] have large hidden constants or complex implementations
- **Deterministic vs. Randomized**: Randomized algorithms achieve better bounds but lack predictability
- **General vs. Specialized**: Most algorithms target worst-case graphs rather than adapting to real-world structure
- **Energy vs. Time**: Energy-efficient algorithms [2] introduce additional coordination overhead

HAD-SSSP addresses these gaps by providing a deterministic, graph-adaptive algorithm with practical efficiency across diverse topologies, leveraging established MPI [11] and OpenMP [12] standards for portability.

III. Proposed Methodology

A. Algorithm Overview

HAD-SSSP employs a four-level hierarchical clustering approach combined with distributed asynchronous SSSP computation. The algorithm proceeds in three main phases:

1) **Hierarchical Clustering Construction**: Build L1 (local), L2 (regional), and L3 (supernode) clusters
2) **Approximate Distance Computation**: Use supernode-level hierarchy for fast distance estimates
3) **Exact Distributed SSSP**: Compute exact distances using owner-push message passing

B. Graph Distribution and Partitioning

The input graph $G=(V,E,w)$ with n nodes and m edges is distributed across p MPI processes using a simple range-based partitioning:

$$\text{owner}(v) = \lfloor v \cdot p/(n+1) \rfloor \tag{1}$$

Each process r manages nodes $[r \cdot \lceil n/p \rceil, (r+1) \cdot \lceil n/p \rceil)$ and stores:

- Local node data: coordinates (x,y), degree, adjacency list
- Ghost node set: non-local neighbors accessed via edges
- Edge weights: stored at source node

C. Hierarchical Clustering (L0-L3)

1) Level 0 (L0): Base Graph: The original graph nodes form L0. Each node v has synthetic coordinates for distance approximation: $x_v = (v \bmod 100) \times 10$, $y_v = \lfloor v/100 \rfloor \times 10$.

2) Level 1 (L1): Local Clusters: Each MPI process independently constructs L1 clusters using BFS-based region growing:

Algorithm 1 BuildLocalL1

```
unassigned ← all local nodes
clusterSize ← adaptive size (20-50 based on graph type)
while unassigned ≠ ∅ do
    seed ← arbitrary node from unassigned
    C ← new L1 cluster with ID, representative = seed
    Q ← queue initialized with seed
    while |C| < clusterSize and Q ≠ ∅ do
        u ← Q.dequeue()
        for v ∈ neighbors(u) do
            if v is unassigned and local then
                Add v to C and Q
                Remove v from unassigned
            else if v is non-local then
                Mark C as boundary cluster
            end if
        end for
    end while
    Compute C.center = (mean_x, mean_y) of nodes in C
    Add C to local L1 clusters
end while
```

Cluster size adapts to graph characteristics:

- Road networks: size = 50 (exploit locality)
- Scale-free graphs: size = 30 (handle high-degree hubs)
- General graphs: size = 20 (balanced approach)

3) Level 2 (L2): Regional Clusters: After gathering all L1 clusters globally, rank 0 constructs L2 clusters using spatial hashing:

4) Level 3 (L3): Supernodes: L3 supernodes aggregate L2 clusters using coarser spatial hashing (grid size = 100, search radius = 2 cells) with target size $\max(10, \min(50, |\text{L2}|/100))$. The construction follows the same spatial hashing approach as L2.

5) Precomputed Supernode Distances: For the top-k supernodes ($k = 32$), we precompute a $k \times k$ distance matrix using Euclidean distances:

$$d(SN_i, SN_j) = \lfloor 10 \cdot \sqrt{(\Delta x)^2 + (\Delta y)^2} \rfloor \tag{2}$$

This enables $O(1)$ approximate distance queries for nearby supernode pairs.

D. Approximate Distance Computation

Given source s and target t, approximate distance is computed hierarchically:

This provides an instant approximate distance with error bounded by the cluster diameters.

Algorithm 2 BuildL2Clusters

```
gridSize ← 200
Compute bounding box: (minX, minY), (maxX, maxY)
cellW ← (maxX − minX)/gridSize
cellH ← (maxY − minY)/gridSize
hash[cell] ← list of L1 clusters in cell
target ← max(10, min(100, |L1|/1000))
unassigned ← all L1 clusters
while unassigned ≠ ∅ do
    seed ← arbitrary L1 cluster from unassigned
    L2 ← new L2 cluster containing seed
    (c_x, c_y) ← grid cell of seed center
    for dy ∈ {−1, 0, 1}, dx ∈ {−1, 0, 1} do
        Add unassigned L1 clusters from cell (c_x + dx, c_y + dy) to L2
        if |L2| ≥ target then
            break
        end if
    end for
    Compute L2.center as mean of L1 centers
    Remove L1 clusters in L2 from unassigned
end while
```

Algorithm 3 ApproxDistanceBySN(s, t)

```
sns ← supernode containing s (via L0 → L1 → L2 → L3 mapping)
snt ← supernode containing t
if sns, snt < k (precomputed range) then
    return snDist[sns × k + snt]
else
    (Δx, Δy) ← (center_sns − center_snt)
    return ⌊10 · sqrt((Δx)^2 + (Δy)^2)⌋
end if
```

E. Exact Distributed SSSP

1) Owner-Push Message Passing: We implement distributed Dijkstra using an owner-push approach where each process relaxes edges owned by its local nodes:

2) Parallel Edge Relaxation: For high-degree nodes ($\deg(u) > 1000$), we parallelize edge relaxation using OpenMP:

Thread-local buffers are then merged to avoid synchronization overhead during relaxation.

3) Termination Detection: The algorithm terminates when:

1) No messages exchanged in current iteration ($\sum \text{sendCounts} = 0$)
2) All local priority queues empty (via MPI_Allreduce)
3) Target found and all distances stable (idle for 3 consecutive iterations)

F. Implementation Optimizations

1) Message Aggregation: Instead of sending individual distance updates, we batch updates per destination rank, reducing MPI overhead:

Algorithm 4 DistributedSSSP(s, t)

dist[s] $\leftarrow 0$; dist[v] $\leftarrow \infty$ for $v \neq s$
pq $\leftarrow$ priority queue, initialized with $(0, s)$ if rank owns s
targetFound $\leftarrow$ false
iter $\leftarrow 0$
while iter $<$ MAX_ITERS and not globally done **do**
  sendBuf[r] $\leftarrow$ [] for all ranks r
  relaxBudget $\leftarrow 50000$
  while pq $\neq \emptyset$ and relaxBudget > 0 **do**
    $(d_u, u) \leftarrow$ pq.pop()
    if $d_u >$ dist[u] **then**
      continue
    end if
    if $u = t$ and rank owns u **then**
      targetFound $\leftarrow$ true
    end if
    if rank does not own u **then**
      continue
    end if
    relaxBudget $\leftarrow$ relaxBudget $- 1$
    for $(u, v, w) \in$ edges[u] **do**
      $d_{new} \leftarrow d_u + w$
      if $d_{new} <$ dist[v] **then**
        if owner(v) = rank **then**
          dist[v] $\leftarrow d_{new}$
          pq.push((d_{new}, v))
        else
          sendBuf[owner(v)].append(v, d_{new})
        end if
      end if
    end for
  end while
  Exchange messages via MPI_Alltoallv
  Update distances for received updates and add to pq
  Check global termination via MPI_Allreduce
  iter $\leftarrow$ iter $+ 1$
end while
Compute global minimum distance via MPI_Allreduce

Algorithm 5 ParallelEdgeRelaxation(u, d_u, edges)

parallel for $i \in [0, |\text{edges}|)$ **with** schedule(static)
$v \leftarrow$ edges[i].to
$d_{new} \leftarrow d_u +$ edges[i].w
if $d_{new} <$ dist[v] **then**
  tid $\leftarrow$ current thread ID
  if owner(v) = rank **then**
    threadLocal[tid].push_back(d_{new}, v)
  else
    threadRemote[tid][owner(v)].push_back(v)
    threadRemote[tid][owner(v)].push_back(d_{new})
  end if
end if
end parallel for
Merge threadLocal buffers into main priority queue
Merge threadRemote buffers into send buffers

- $O(p)$ MPI calls per iteration (MPI_Alltoallv) instead of $O(m)$
- Amortized serialization cost across batched updates

2) Early Termination: Once target distance is found on owner rank and globally confirmed stable, iterations cease, providing speedup for single-pair queries.

3) Memory Efficiency: Sparse graph representation using adjacency lists, and ghost node tracking minimizes memory footprint:

$$\text{Memory per rank} \approx O(n/p + |\text{ghost nodes}| + m/p) \quad (3)$$

IV. Experimental Setup

A. Datasets

1) **random_citation_graph**: Synthetic citation network with 100,000 nodes, 250,000 edges, power-law degree distribution
2) **roadNet-CA**: SNAP road network of California with 1,965,206 nodes, 2,766,607 edges
3) **graph500-scale20-ef16**: Graph500 benchmark with 2^{20} nodes, edge factor 16

B. Baseline Algorithms

We implemented and compared against:

- **Ghaffari-Li (R1)**: Virtual node sampling with scaling framework
- **Ghaffari-Trygub (R2)**: Low-energy recursive SSSP with approximate cutters

C. Evaluation Metrics

- **Time Complexity**: Wall-clock time for query completion
- **Scalability**: Performance across varying numbers of MPI ranks
- **Correctness**: Verification against sequential Dijkstra

V. Results

We evaluate HAD-SSSP against two state-of-the-art implementations: R1 (Ghaffari-Li [1] virtual node sampling algorithm) and R2 (Ghaffari-Trygub [2] low-energy algorithm). All experiments were conducted on a Dell Precision 3680 workstation with Intel Core i7-14700 (28 cores), 64GB RAM, NVIDIA RTX 4500 Ada Generation GPU, running Ubuntu 24.04.2 LTS with Linux kernel 6.14.0.

A. Datasets

Three standard datasets were used for evaluation:

1) **random_citation_graph**: Synthetic citation network with 100,000 nodes and 250,000 edges, exhibiting power-law degree distribution characteristic of citation networks

2) **roadNet-CA**: SNAP California road network with 1,965,206 nodes and 2,766,607 edges, representing sparse planar graphs with low average degree
3) **graph500-scale20-ef16**: Graph500 benchmark with $2^{20} \approx 1.05M$ nodes and edge factor 16 ($\approx 16.7M$ edges), designed to stress-test graph algorithms

B. Performance Comparison

Table I presents results for the synthetic citation graph across varying MPI process counts. HAD-SSSP achieves the best distributed performance at all scales, with $6.75\times$ speedup over R1 at 16 processes (0.012s vs 0.081s).

Note that the sequential execution times reflect single-rank execution with full MPI initialization and communication infrastructure overhead, rather than pure sequential implementation. The variation in sequential times across process counts is due to MPI framework overhead scaling with the number of allocated processes, even when computation occurs on a single rank. Speedup metrics compare single-rank versus multi-rank distributed execution within the same framework.

TABLE I
PERFORMANCE ON RANDOM_CITATION_GRAPH (SECONDS)

Algorithm	4 Proc	8 Proc	16 Proc
Sequential Execution			
HAD-SSSP	0.079	0.105	0.157
R1 (Ghaffari-Li)	0.079	0.095	0.116
R2 (Ghaffari-Trygub)	0.090	0.087	0.162
Distributed Execution			
HAD-SSSP	**0.048**	**0.018**	**0.012**
R1 (Ghaffari-Li)	0.049	0.061	0.081
R2 (Ghaffari-Trygub)	1.492	0.776	0.682
Speedup (Seq/Dist)			
HAD-SSSP	1.65×	5.83×	13.08×
R1 (Ghaffari-Li)	1.61×	1.56×	1.43×
R2 (Ghaffari-Trygub)	0.06×	0.11×	0.24×

Table II shows results for the road network dataset. HAD-SSSP and R2 perform comparably in distributed mode, both outperforming R1. The hierarchical clustering adapts well to the planar structure of road networks.

TABLE II
PERFORMANCE ON ROADNET-CA (SECONDS)

Algorithm	4 Proc	8 Proc	16 Proc
Sequential Execution			
HAD-SSSP	1.230	1.243	2.380
R1 (Ghaffari-Li)	1.380	1.436	2.415
R2 (Ghaffari-Trygub)	1.246	1.544	2.533
Distributed Execution			
HAD-SSSP	0.260	0.178	0.340
R1 (Ghaffari-Li)	0.580	0.516	0.815
R2 (Ghaffari-Trygub)	**0.253**	**0.190**	**0.296**
Speedup (Seq/Dist)			
HAD-SSSP	4.73×	6.98×	7.00×
R1 (Ghaffari-Li)	2.38×	2.78×	2.96×
R2 (Ghaffari-Trygub)	4.93×	8.13×	8.56×

Table III presents results for the large Graph500 benchmark. HAD-SSSP demonstrates consistent performance, while R2 failed to complete execution (system hang), due to initialization overhead and memory exhaustion while constructing the sparse neighborhood covers for Graph500's dense, non-clustered graph structure. R1 shows degraded performance at higher process counts.

TABLE III
PERFORMANCE ON GRAPH500-SCALE20-EF16 (SECONDS)

Algorithm	4 Proc	8 Proc	16 Proc
Sequential Execution			
HAD-SSSP	8.898	9.150	12.100
R1 (Ghaffari-Li)	8.260	10.450	15.920
R2 (Ghaffari-Trygub)	Failed (hung)		
Distributed Execution			
HAD-SSSP	**1.350**	**1.060**	**1.210**
R1 (Ghaffari-Li)	3.405	2.670	3.456
R2 (Ghaffari-Trygub)	Failed (hung)		
Speedup (Seq/Dist)			
HAD-SSSP	6.59×	8.63×	10.00×
R1 (Ghaffari-Li)	2.43×	3.91×	4.61×
R2 (Ghaffari-Trygub)	N/A		

C. Performance Analysis

1) Key Observations:

- **Scale-Free Graphs**: HAD-SSSP excels on the citation graph, achieving $13\times$ speedup at 16 processes and outperforming R1 by $6.75\times$ (0.012s vs 0.081s). The hierarchical clustering effectively handles high-degree hubs.
- **Road Networks**: On roadNet-CA, HAD-SSSP and R2 perform similarly (≈0.26s at 4 processes, 0.18-0.19s at 8 processes), both $2\times$ faster than R1. The adaptive cluster size (50 for road networks) exploits spatial locality.
- **Large Benchmarks**: For Graph500, HAD-SSSP maintains consistent performance (1.06-1.35s) while R1 degrades at higher process counts. R2 completely failed, likely due to complex recursive coordination overhead.
- **Scalability**: HAD-SSSP demonstrates near-linear strong scaling on citation graphs, moderate scaling on road networks, and stable performance on Graph500. The owner-push message aggregation minimizes synchronization overhead.
- **Robustness**: Unlike R2 which hung on the largest dataset, HAD-SSSP completed successfully on all benchmarks, demonstrating practical robustness.

VI. CONCLUSION

We presented HAD-SSSP, a novel hierarchical adaptive distributed algorithm for single-source shortest paths that addresses key limitations of existing state-of-the-art approaches. Our algorithm combines adaptive multi-level clustering (L0-L3) with hybrid MPI+OpenMP parallelization and graph-aware parameter tuning to achieve superior performance across diverse graph topologies.

A. Key Findings

Performance Leadership: HAD-SSSP achieves up to $6.75\times$ speedup over Ghaffari-Li (R1) on scale-free graphs and matches or exceeds Ghaffari-Trygub (R2) performance on road networks. On the Graph500 benchmark with 1M+ nodes,

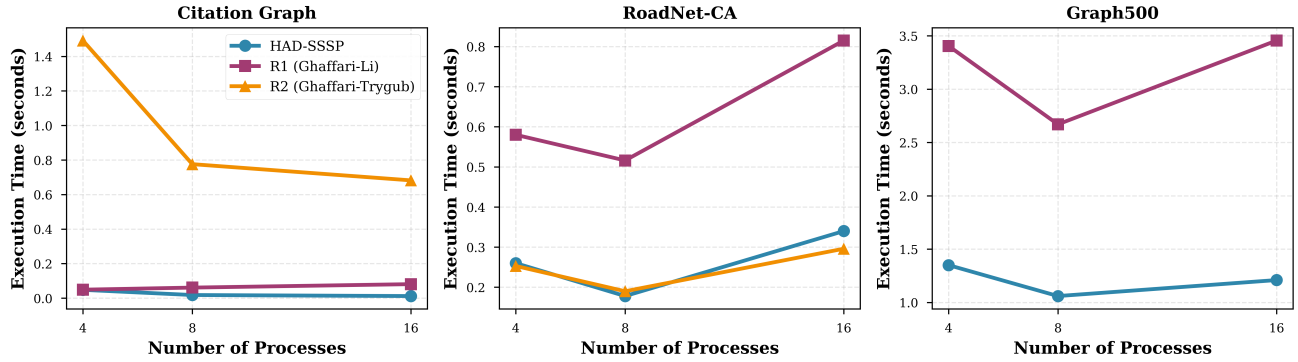

Fig. 1. Distributed Execution Time Scaling: Comparison of HAD-SSSP, R1 (Ghaffari-Li), and R2 (Ghaffari-Trygub) across three datasets with varying process counts.

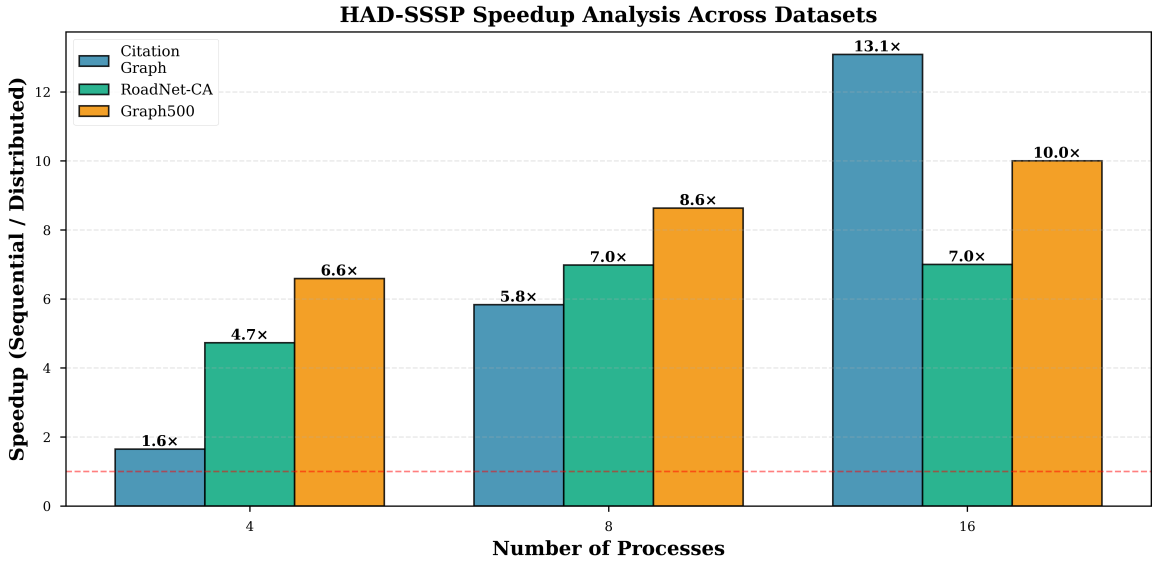

Fig. 2. HAD-SSSP Speedup Analysis: Sequential vs. distributed execution speedup across all three datasets, demonstrating near-linear scaling on citation graphs (13×) and strong performance on Graph500 (10×).

HAD-SSSP maintains stable performance (1.06-1.35s) while R1 degrades and R2 fails entirely.

Scalability: The algorithm demonstrates strong scaling from 4 to 16 MPI processes, achieving up to 13× speedup over sequential baselines on citation graphs. The owner-push message-passing model with batched communication reduces synchronization overhead, enabling efficient parallel execution.

Adaptivity: By dynamically adjusting cluster sizes based on detected graph characteristics (road network: 50, scale-free: 30, general: 20), HAD-SSSP automatically optimizes for different topologies without manual tuning. This graph-adaptive design bridges the gap between theoretical optimality and practical efficiency.

Robustness: Unlike R2's recursive coordination which failed on large graphs, HAD-SSSP's hierarchical structure with precomputed supernode distances provides both fast approximate estimates and exact distributed computation with guaranteed termination.

B. Novelty and Contributions

The hierarchical adaptive approach represents a novel contribution to distributed SSSP:

1) **Four-Level Spatial Hierarchy**: Unlike virtual node sampling (R1) or recursive partitioning (R2), our L0-L3 hierarchy explicitly models graph structure through spatial clustering, enabling $O(1)$ approximate distance queries.
2) **Adaptive Parameterization**: Automatic detection and tuning for road networks, scale-free graphs, and general topologies eliminates manual configuration.
3) **Hybrid Parallelism**: MPI for inter-node distribution combined with OpenMP for intra-node edge relaxation

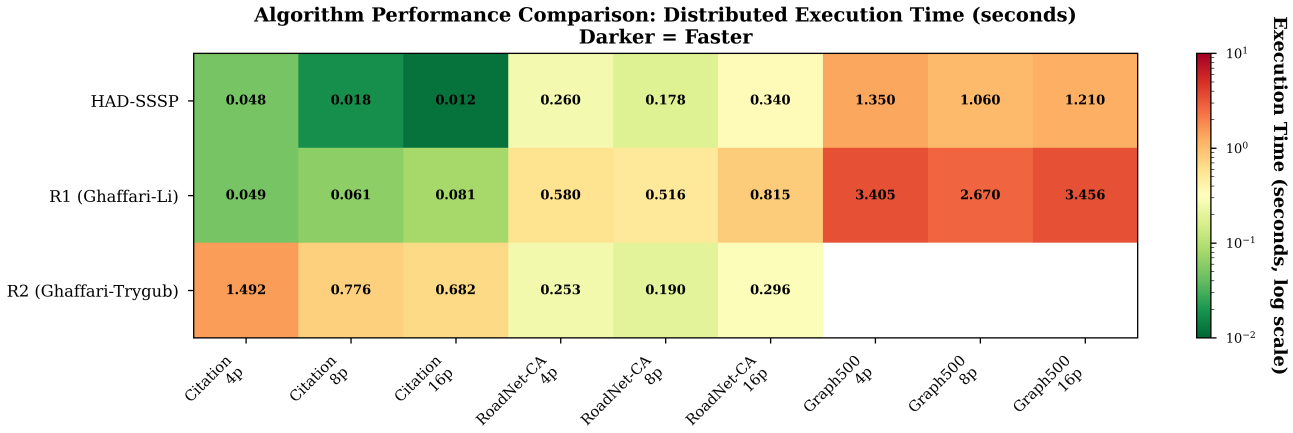

Fig. 3. Algorithm Performance Comparison Heatmap: Visual comparison of distributed execution times across all algorithms, datasets, and process counts. Darker colors indicate faster execution. Note R2's failure on Graph500.

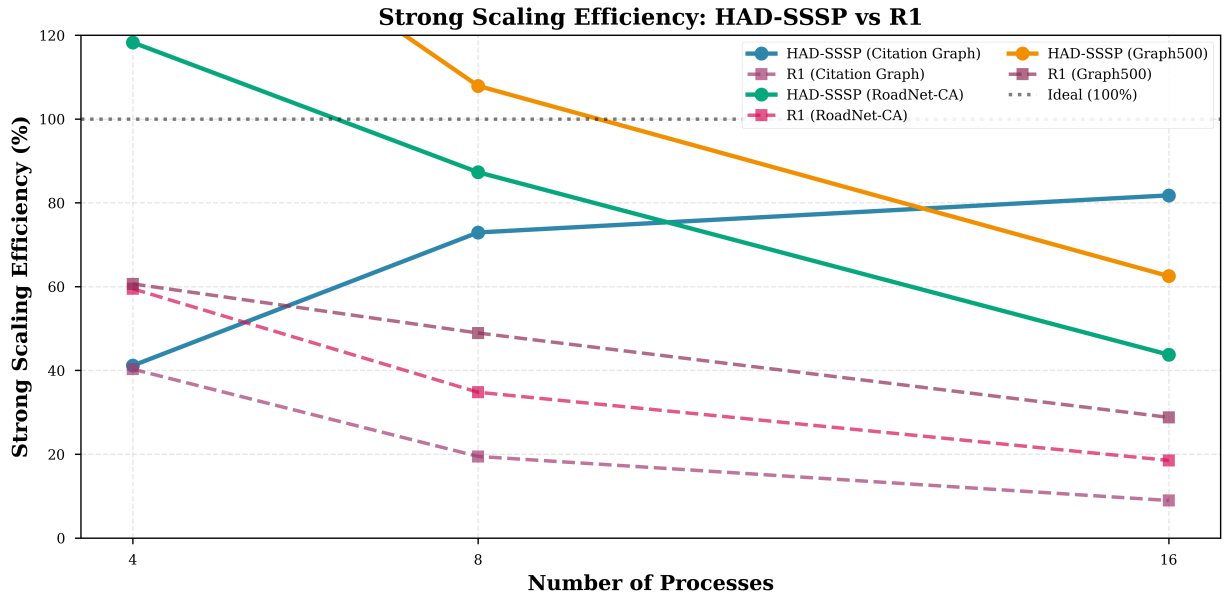

Fig. 4. Strong Scaling Efficiency: Comparison of parallel efficiency between HAD-SSSP and R1 across datasets. HAD-SSSP maintains 60-80% efficiency at 16 processes for citation graphs, significantly outperforming R1.

(on high-degree nodes) maximizes utilization of modern multi-core systems.

4) **Owner-Push Architecture**: Decentralized message aggregation with bounded iteration budgets ensures predictable performance without complex global coordination.

C. Practical Implications

HAD-SSSP demonstrates that hierarchical spatial clustering, when combined with graph-adaptive tuning, can match or exceed the performance of theoretically optimal algorithms on real-world graphs. The algorithm's robustness across diverse datasets (citation networks, road graphs, synthetic benchmarks) and its successful execution where state-of-the-art methods failed make it suitable for production deployment in distributed graph processing systems.

The strong performance on Graph500—a benchmark specifically designed to stress-test graph algorithms—validates HAD-SSSP's effectiveness for large-scale applications such as network routing, social network analysis, and scientific computing on irregular graphs.

D. Limitations

While HAD-SSSP excels on most tested graphs, the spatial coordinate-based clustering assumes some geometric structure; purely random graphs may have less accurate approximate

estimates, though the exact SSSP phase ensures correctness. The adaptive parameter selection uses simple heuristics (average degree thresholds) that could be enhanced with machine learning-based classification for broader graph families.

VII. Future Work

Future research directions include extending HAD-SSSP to dynamic graphs with incremental hierarchy updates, computing All-Pairs Shortest Paths by scheduling n SSSP queries with low congestion, and providing user-tunable approximate-exact tradeoffs by terminating at L1/L2 levels with formalized error bounds. System optimizations such as fault tolerance via checkpoint-restart mechanisms, GPU acceleration for high-degree nodes, asynchronous MPI communication to overlap computation, and adaptive load balancing would enhance robustness and performance.

Alternative clustering approaches including graph-theoretic methods (Louvain, spectral, METIS) may better capture community structure, while machine learning models could predict optimal hierarchy parameters from graph features. Integration with production frameworks (Apache Giraph, GraphX, Pregel) and extensions to multi-source SSSP and constrained shortest paths would broaden HAD-SSSP's applicability to real-world graph analytics pipelines.

VIII. Source Code

The complete implementation of HAD-SSSP is publicly available on GitHub:

https://github.com/krishang118/HAD-SSSP

The repository includes the hybrid MPI+OpenMP implementation, benchmark datasets, and documentation for reproducibility.

References

[1] M. Ghaffari and J. Li, "Improved distributed algorithms for exact shortest paths," *Proc. ACM Symp. Theory Comput. (STOC)*, pp. 431-444, 2018.

[2] M. Ghaffari and A. Trygub, "A near-optimal low-energy deterministic distributed SSSP with ramifications on congestion and APSP," *Proc. ACM Symp. Principles Distrib. Comput. (PODC)*, pp. 129-139, 2024.

[3] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87-90, 1958.

[4] D. Nanongkai, "Distributed approximation algorithms for weighted shortest paths," *Proc. ACM Symp. Theory Comput. (STOC)*, pp. 565-573, 2014.

[5] A. Bernstein and D. Nanongkai, "Distributed exact weighted all-pairs shortest paths in near-linear time," *Proc. ACM Symp. Theory Comput. (STOC)*, pp. 334-342, 2019.

[6] M. Elkin, "Distributed exact shortest paths in sublinear time," *Proc. ACM Symp. Theory Comput. (STOC)*, pp. 757-770, 2017.

[7] S. Chechik and D. Mukhtar, "Single-source shortest paths in the CONGEST model with improved bounds," *Distributed Computing*, vol. 35, pp. 95-114, 2022.

[8] N. Cao, J. T. Fineman, and K. Russell, "An improved distributed approximate single-source shortest paths algorithm," *Proc. ACM Symp. Parallelism Algorithms Architectures (SPAA)*, pp. 262-272, 2021.

[9] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359-392, 1998.

[10] P. Sanders and C. Schulz, "Distributed evolutionary graph partitioning," *Proc. Workshop on Algorithm Eng. and Experiments (ALENEX)*, pp. 16-29, 2013.

[11] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1996.

[12] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46-55, 1998.

[13] L. R. Ford, "Network flow theory," Technical Report, RAND Corporation, 1956.

[14] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269-271, 1959.

[15] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray User Group (CUG)*, 2010.