An Efficient Multi-core Parallel Implementation of SSSP Algorithm with Decreasing Delta-stepping

Rakibul Hassan
Department of Computer Science
University of Nevada, Las Vegas
Las Vegas, USA
hassar2@unlv.nevada.edu

Shaikh Arifuzzaman
Department of Computer Science
University of Nevada, Las Vegas
Las Vegas, USA
shaikh.arifuzzaman@unlv.edu

***Abstract*—Single-Source Shortest Path (SSSP) algorithms are essential in various applications, from optimizing transportation networks to analyzing social networks. This paper focuses on implementing and optimizing a parallel Delta-Stepping algorithm for SSSP on shared-memory systems. We evaluate our algorithm's performance using synthetic Kronecker graphs and large-scale real-world datasets, including LiveJournal, Orkut, and Twitter. Our parallel Delta-Stepping algorithm achieves significant speedup through several techniques: gradual reduction of the delta value, property-driven graph reordering, bucket fusion, and dynamic load balancing. Extensive experiments show that our Parallel Delta-Stepping with Decreasing Delta (PD3) achieves speedups between $10\times$ and $65\times$ compared to the best-known serial implementations and is faster than or comparable to other existing parallel methods. These results highlight the importance of customized parallelization strategies and algorithmic optimizations for efficient SSSP computation in shared-memory systems, with broad implications for various graph-based applications.**

***Index Terms*—SSSP, Parallel Algorithm, Large graphs, Delta-Stepping, Parallel Optimization, Multi-core Systems**

I. Introduction

Graph algorithms are essential in various domains such as network routing, transportation network optimization, and social network analysis [3], [6]–[8], [14], [16]. With the proliferation of massive-scale graphs containing millions of vertices and billions of edges, the need for efficient parallel implementation of these algorithms has become increasingly critical [4], [5], [9], [11], [13], [23], [24]. Among the essential graph algorithms, Single-Source Shortest Path (SSSP) algorithms are widely used as one of the fundamental tools for extracting valuable insights from complex graph structures [8], [19]. Motivated by the challenges posed by large-scale graph datasets, our research focuses on efficient implementation of parallel SSSP algorithms on shared-memory systems. In particular, we address the scalability concerns by implementing Parallel Delta-Stepping Algorithm with some important optimizations.

Among classical algorithms for finding the shortest path from a single source in a graph, Dijkstra's algorithm [12] and the Bellman-Ford algorithm [8] are the most widely used. Dijkstra's algorithm employs a priority queue and is highly work-efficient due to its linear edge visitation. In contrast, the Bellman-Ford algorithm has a higher time complexity but can handle graphs with negative cycles and offers greater potential for parallelization [25]. Despite Dijkstra's algorithm being optimal in terms of work efficiency, its inherently sequential nature limits its scalability. Conversely, the Bellman-Ford algorithm is highly parallelizable because it allows for independent processing of each vertex. However, its inefficiency due to redundant edge visits makes it suboptimal for massive graph datasets [25].

Parallel Delta-Stepping strikes a balance between work efficiency and parallelizability. It uses a bucket-based approach to process vertices concurrently, prioritizing vertices in a bucket that have smaller distances from the source [19]. A parameter called delta is used to define the buckets, facilitating concurrent processing while maintaining efficiency.

In this paper, we implemented a parallel version of Delta-Stepping algorithm called PD3 which dynamically modifies the delta parameter from a higher value at the beginning to speed up frontier expansion to a lower, more-optimal value towards the end steps of the algorithm. It also uses a bucket fusion technique to relax the edges in two phases and vertices with smaller distances from the source vertex, get the opportunity of repeated updates as far as they fall into the smallest bucket [30]. Moreover, as real-world graphs exhibit a power-law degree distribution, load balancing is an inherent problem [6], [22]. PD3 utilizes dynamic load balancing technique to distribute the workload properly to each of the threads. Additionally, we have implemented some pre-processing steps like graph reordering to improve the performance of PD3 algorithm. For example, to reach the depth of graph quickly, we have reordered the input graph based on degree distribution where the vertices with higher degree get the lower index. Also, we have sorted the edge list in ascending order so that the smaller edges which are potential parts of SSSP tree appears sooner and we have less amount of relaxations.

In this paper, we present our implementation and experimentation of various SSSP algorithms, including Dijkstra's algorithm, the parallel Bellman-Ford algorithm, and our efficient parallel delta-stepping algorithm, *PD3*. Our study aims to provide a comprehensive comparative analysis of these algorithms' performances across a range of graph sizes and structural properties, using both synthetic and real-world large graphs. Additionally, we evaluate the applicability of our PD3 algorithm by comparing it against state-of-the-art parallel SSSP implementations and the traditional serial Dijkstra's algorithm. PD3 demonstrates superior performance and scalability when handling massive graph networks. Experimental results reveal substantial speedups achieved by PD3, ranging from $10\times$ to $65\times$ compared to Dijkstra's algorithm, and up to a 1.4-fold improvement over the state-of-the-art parallel implementation developed by the GAP Benchmark Suite (GAPBS) [7]. These findings underscore PD3's effectiveness in overcoming the scalability challenges inherent in traditional SSSP algorithms. The contributions of this study are summarized as follows:

- We introduce an efficient parallel version of the Delta-Stepping algorithm (PD3) that utilizes an optimized delta parameter and bucket fusion technique to enhance the efficiency and scalability of the SSSP algorithm.
- We implement a dynamic load balancing technique to evenly distribute workloads across threads, effectively addressing challenges posed by power-law degree distributions in large graphs.
- We incorporate preprocessing steps, including graph reordering based on degree distribution and sorting edge lists, to improve parallel performance by accelerating graph traversal and reducing the number of distance updates.
- We conduct comprehensive experiments comparing parallel Bellman-Ford, the state-of-the-art parallel SSSP implementation by GAPBS, and PD3, demonstrating PD3's superior performance and scalability (e.g., achieving $10\times$ to $65\times$ speedups over Dijkstra's algorithm and up to a 1.4-fold improvement over the GAPBS parallel implementation).
- We highlight the applicability of the PD3 algorithm in handling massive graph networks, providing valuable insights and demonstrating its effectiveness on both synthetic and real-world networks.

II. Background and Related Work

This section covers the foundational concepts and previous work relevant to SSSP problem. We discuss existing methods, highlight their limitations, and establish the context for the proposed approach.

A. Single-Source Shortest Path (SSSP) Problem

The Single-Source Shortest Path (SSSP) problem aims to determine the shortest paths from a given root vertex r to all other vertices in a weighted graph $G = (V, E, w)$. Here, V is the set of vertices, E is the set of edges, $n = |V|$ represents the number of vertices, and $m = |E|$ represents the number of edges. The weight function $w(e)$ assigns a non-negative weight to each edge $e \in E$. To solve the SSSP problem, the algorithm starts by setting the distance $d(r)$ to 0 for the root vertex r which is also known as source vertex and $d(v)$ to infinity for all other vertices $v \in V$. The root vertex r is initially active. The algorithm then proceeds through a series of relaxation operations, updating the distances $d(v)$ to reflect the shortest paths.

Relaxation is the key operation in the SSSP problem and is employed by various algorithms, such as the Dijkstra's algorithm, the Bellman-Ford algorithm, and the Delta-Stepping algorithm. For an edge $e = (u, v, w)$, the relaxation operation is defined as:

$$d(v) = min[d(v), d(u) + w(e)] \tag{1}$$

This operation ensures that the shortest known distance to each vertex is improved iteratively, leading to the final shortest distances from the root vertex r to all other vertices in the graph.

B. CSR Representation

There are several choices of data structures to represent the graph for SSSP problem. Linear algebra based SSSP implementations such as GraphBLAS [15], [18] uses matrix representation for input graph. Adjacency list is also popularly used for solving graph problems. In this study, we have used Compressed Sparse Row (CSR) format which is a widely used method for efficient graph representation in shared-memory systems [7], [10], [28]. Unlike adjacency matrices, which often lead to wasted storage space due to their sparsity, or adjacency lists, which can be slow for data access, the CSR format offers a balanced solution by reducing memory usage while maintaining relatively fast access times.

C. Related Work

The prevalence of power law degree distribution in real-world graphs poses a significant challenge for developing efficient parallel algorithms [26]. This distribution leads to load imbalances, complicating the task of designing algorithms capable of effectively handling real-world datasets. Additionally, the evaluation of algorithmic efficiency often requires scale-free datasets that mimic the characteristics of real-world graphs. Prior to 2010, the scarcity of weighted scale-free datasets hindered the development of parallel and scalable Single-Source Shortest Path (SSSP) algorithms. However, the introduction of the Graph Benchmark Suite in 2010, addressed this challenge by providing a method to generate scale-free Recursive Matrix-based Kronecker graphs [1]. These graphs exhibit power law properties similar to those of real-world graphs, enabling researchers to accurately measure the efficiency and scalability of SSSP algorithms on datasets that closely resemble real-world scenarios. This development has significantly advanced the field of parallel and scalable SSSP algorithms, facilitating more accurate evaluations and enabling the creation of algorithms tailored to handle the complexities of real-world graphs.

Different algorithms are used for finding SSSP. Dijkstra's [12] algorithm gives the best solution with optimal edge visit but it is very much serial in nature. On the other hand, Bellman-Ford [8] can work with different vertices at the same time, but it might do a lot of unnecessary repeated work. The Delta-Stepping algorithm [19] balances these two by using different groups to organize vertices based on their distance from the source vertex. This technique helps it to run the algorithm efficiently while also being fast. Many studies have shown that Delta-Stepping is one of the quickest and most effective methods when it comes to parallel SSSP computing [17], [20], [21], [27].

Several studies have explored SSSP algorithms on both shared and distributed memory systems. Chakaravarthy et al. [9] introduced an optimized version of the Delta-Stepping algorithm for shared-memory systems. Their optimizations included hybridization with Bellman-Ford, optimizing directions in SSSP, and implementing an effective load-balancing strategy. Their implementation achieved a promising result using 32,768 IBM Blue Gene nodes on an R-MAT generated graph with edge weights ranging from 1 to 255. However, for extremely large power-law graphs, the scalability required significant enhancement. Yu, Huashan, Wang, and Luo introduced an Edge-Fencing strategy to optimize SSSP computation on large-scale graphs [29]. Their technique customizes the schedule for each SSSP computation, focusing on path-centric approach and utilizing a small set of fence values to select relaxed edges efficiently. The Edge-Fencing approach demonstrated significant performance improvements, achieving $3.83\times$-$55\times$ higher Giga Traversed Edges Per Second (GTEPS) compared to regular Delta-Stepping. But their experiments were limited into small size graphs. They used Kronecker graphs to scale 21 for their experimentation.

Wang et al. [26] also implemented a version of Delta-Stepping algorithm name hyper stepping by combining Parallel Degree Heap with Optimized Delta-Stepping [9]. Using a Supercomputer, they ran Kronecker graphs with weight [0,1) and got significant speedup. Their implementation reached 7638 GTEPS with 103158 processors (over 40 million cores). Wang et al. achieved a performance improvement of $3.7\times$ and handled graph sizes 512 times larger with their implementation. However, their work provided more focus on work efficiency than resource utilization. Zhang et al. [31] introduced a Bucket-aware Asynchronous Single-Source Shortest Path Algorithm for GPU. Their method, based on "Property-driven Reordering", recognizes the impact of vertex degree and edge weight properties on SSSP execution. By reordering vertices in descending order of degree and reassigning indices accordingly, the algorithm prioritizes frequently accessed vertices with higher degrees. Additionally, edges are reordered based on weight, with small-weighted edges receiving priority for updates. The authors also implemented adaptive load balancing and kernel fusion techniques to enhance performance. While their approach showed promising results on small-scale Kronecker graphs for GPU based systems, its performance suffered on road networks, indicating areas for further improvement in real-world applications.

In our proposed SSSP approach, we adopt the property driven reordering technique along with dynamic load balancing to optimize the performance of Delta-Stepping algorithm for CPU environment.

III. Methodology

This section covers the detailed process and methodology employed in developing the proposed PD3 algorithm.

A. Frontier Selection Process

In our exploration of shortest path algorithms, we examine three prominent methodologies: Dijkstra's algorithm, Bellman-Ford algorithm, and Delta-Stepping algorithm. Each algorithm adopts a distinct strategy for selecting vertices to expand, influencing its efficiency and parallelization potential. Dijkstra's algorithm, renowned for its work efficiency, prioritizes the currently found shortest non-expanded vertex as the next frontier. This approach ensures optimal path discovery but operates in a serial fashion, hindering parallelization possibilities. Conversely, the Bellman-Ford algorithm casts a wider net, considering all vertices for potential relaxation in each iteration. This highly parallelizable nature allows efficient parallel computation but introduces the overhead of revisiting vertices multiple times, particularly in graphs with extensive edges or cycles. In striking a balance between efficiency and parallelization, the Delta-Stepping algorithm emerges as a compelling compromise. By partitioning the graph into intervals of predetermined size (referred to as delta), the algorithm selects vertices within these intervals for expansion, constituting the next frontier. This methodology mitigates unnecessary vertex revisits while maintaining a degree of parallelism, offering a promising trade-off between the serial nature of Dijkstra's algorithm and the exhaustive exploration of Bellman-Ford. Figure 1 visually depicts the frontier selection process for each of the algorithms.

Here, the green colored vertices indicate the vertices which will be expanded in the next iteration. For Dijkstra's algorithm, only the non-expanded vertex with shortest distance is selected, hence selecting vertex 2 as it has the distance 2 from the source vertex that is the current minimum. For Bellman-Ford algorithm, all of the vertices are explored in the next iteration. However, for Delta-Stepping algorithm with delta value = 3, vertex 2 and 3 will be selected as frontier and expanded in the next iteration as their distances which are 2 and 3 respectively, belongs within the range of delta value 3.

B. Proposed SSSP Approach

Figure 2 Summarizes the workflow of our proposed approaches for the SSSP problem. Here, we aim to comprehensively address the challenges posed by collecting various types of network data, including social, transportation, and biological networks. Also, we have used scale free Kronecker graphs upto scale 23 for our experiments. To effectively handle the collected data, we employ multiple data representation techniques such as adjacency matrices, adjacency lists, and Compressed Sparse Row (CSR) format. Analyzing the performance of our algorithms with all of these representations, we have chosen CSR format as it provides better performance for large scale graph data.

A crucial aspect of our approach is data pre-processing, where we implement "Property-driven Reordering" strategy

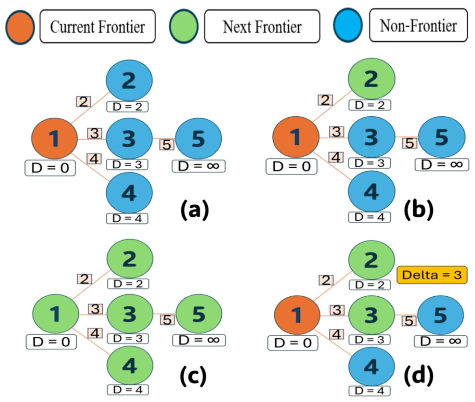


Fig. 1. (a) Sample graph; Frontier list expansion process of: (b) Dijkstra, (c) Bellman Ford, and (d) Delta-Stepping methods.

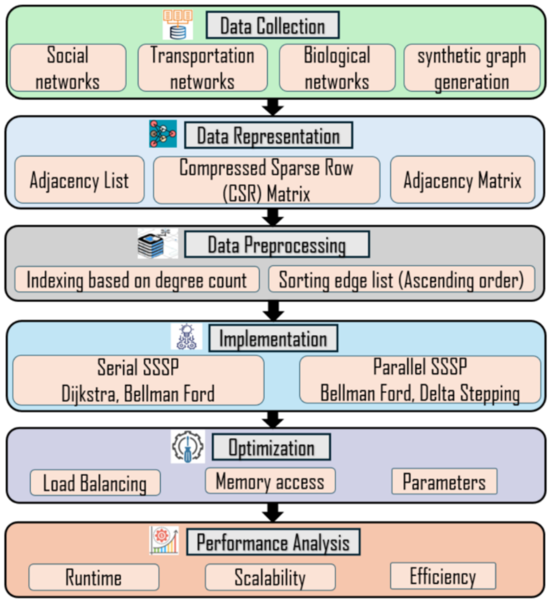


Fig. 2. Workflow of our proposed SSSP methods.

that involves degree-based indexing and sorting of the edge list to optimize data access patterns and improve the efficiency of our PD3 algorithm. For implementation, we consider both serial and parallel SSSP algorithms including serial Dijkstra, serial Bellman-Ford, parallel Bellman-Ford, state of the art Parallel Delta-Stepping algorithm from GAPBS and PD3 with varying thread configurations. Our PD3 implementation incorporates optimization techniques such as dynamic load balancing, gradual decrease of delta and bucket fusion which are essential for maximizing the efficiency and scalability. Finally, we conduct a thorough performance analysis of our proposed approach, focusing on scalability, runtime, and speed-up metrics.

C. Parallel Delta-Stepping with Decreasing Delta (PD3)

This paper introduces parallel version of the Delta-Stepping algorithm, known as PD3, which incorporates several optimizations to enhance its performance on large-scale graphs. This section details the techniques and strategies employed in PD3. The algorithm 1 shows the pseudo code of proposed PD3 implementation. Algorithm 2 shows how the delta is decreased. We have experimented on different decrease rate and the optimal one is chosen. Details of each optimization strategies are provided in the following subsections.

Algorithm 1: Parallel Delta-Stepping with Decreasing Delta Algorithm

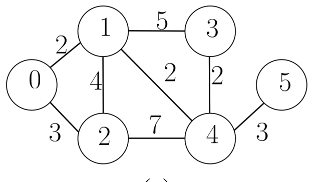
```
Input: Graph $G$, source vertex $r$, threshold value $\delta$
Output: Shortest paths from $s$ to all other vertices
Initialize array $dist$ with $\infty$ for all vertices except $r$
  with $dist[r] = 0$;
Initialize array of vectors $local\_bins$ for holding
  vertices;
Initialize frontier $F$ with source vertex $r$;
Set, MinimumDelta $\leftarrow \delta$ $Set, \delta \leftarrow$
  $\delta \times 3$ while $current\_bin \neq kMaxBin$ do
    for each thread do
        for each vertex $u$ in $F$ do
            if $dist[u] \geq \delta \times current\_bin$ then
                RelaxEdges($G, u, \delta, dist, local\_bins$);
            end
        end
        while small bin not empty do
            Process vertices in the small bin;
            for each vertex $u$ in the small bin do
                RelaxEdges($G, u, \delta, dist, local\_bins$);
            end
        end
        $\delta \leftarrow$
          $CalculateNewDelta(MinimumDelta, \delta)$
        Find next non-empty bin;
    end
    Update frontier $F$ with vertices from the next bin;
    Prepare for next iteration;
end
return final distances $dist$;
```

Algorithm 2: CalculateNewDelta

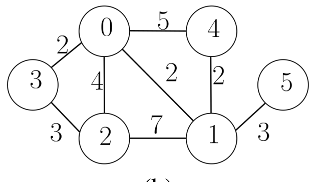
```
Input: Minimum delta value $min\_delta$, current delta
       value $current\_delta$
Output: New delta value
Calculate $new\_delta \leftarrow$
  $\max(min\_delta, current\_delta/1.20)$;
return $new\_delta$;
```

1) Optimized Use of Delta Parameters: PD3 begins with a higher delta value to quickly expand the frontier list, covering a good portion of the graph initially. Otherwise, a good number of initial iterations will not contain sufficient amount of frontier vertices to work with. As the algorithm progresses, the delta value is gradually reduced to an optimal level, balancing the speed of exploration with the precision of edge relaxation. We have empirically set the initial delta 3 times of the provided delta. After few iterations it decreases to the given delta.

2) Bucket Fusion Technique: PD3 uses a bucket fusion technique to facilitate edge relaxation in two phases. This technique allows vertices, which are closer to the source



(a)



(b)

Fig. 3. (a) Input graph, (b) Reordered graph based on degree based indexing.

vertex to receive repeated updates as long as they fall into the smallest bucket. By prioritizing updates for these closer vertices, PD3 improves the accuracy and speed of shortest path calculations. Also, these two phases can run asynchronously, which provides computational benefit in parallel environment.

3) Dynamic Load Balancing: Real-world graphs often exhibit a power-law degree distribution, where a few vertices have many connections, and most vertices have few. This distribution makes it challenging to balance the computational load across threads. PD3 addresses this with a dynamic load balancing technique, which dynamically distributes the workload among threads to ensure efficient use of resources and prevent bottlenecks.

4) Pre-processing Steps: To further enhance PD3's performance, several pre-processing steps are implemented. The input graph is reordered based on the degree distribution of vertices. Vertices with higher degrees are given lower indices, facilitating quicker traversal to the core parts of the graph. Figure 3 shows how input graph is converted to reordered graph based on degree count. This reordering helps rapidly reach the graph's core structure, improving the algorithm's overall efficiency [31]. Moreover, the edge list is sorted in ascending order so that edges forming part of the SSSP tree appear earlier in the processing sequence, resulting in fewer updates. Sorting the edge list minimizes redundant computations and speeds up the algorithm's convergence.

IV. Experimental Evaluation

In this section, we cover the evaluation of the proposed SSSP approaches through detailed performance analysis.

A. Dataset Description

For evaluating the performances of implemented SSSP algorithms, we used both synthetic and real world graph dataset. The synthetic datasets utilized in this study were generated using the Kronecker method with an edge factor of 16 as provided in Graph 500 benchmark [1]. For real-world datasets, we collected some large graphs covering different domains from renowned Graph data repository [2]. The domain covers social networks, transportation networks and web graphs. Table I shows the details of the datasets.

B. Experimental Environment

We conducted experiments on a Linux machine with **13th Gen Intel Core i7-13700 processor(16 cores)** and **32GB RAM**, also including cache L1 640 KB, L2: 24 MB, and L3: 30 MB. Implementations are conducted in C++ with OpenMP for parallelization.

TABLE I
Dataset details

Dataset	#Vertices	#Edges	Max Degree	Avg Degree
Kron_20	1,048,576	33,554,432	138344	32
Kron_21	2,097,152	67,108,864	210643	32
Kron_22	4,194,304	134,217,728	320556	32
Kron_23	8,388,608	268,435,456	487144	32
road-road-usa	23,947,347	57,708,624	9	2
Live Journal	4,036,538	69,362,378	14815	17
Orkut	3,072,626	234,370,166	33313	76
Youtube	1,157,827	5,975,248	28754	5
socfb-uci-uni	58,790,782	184,416,390	4960	3
soc-sinaweibo	58,655,849	522,642,142	278491	8
soc-twitter-2010	21,297,772	397,538,714	132512905	18

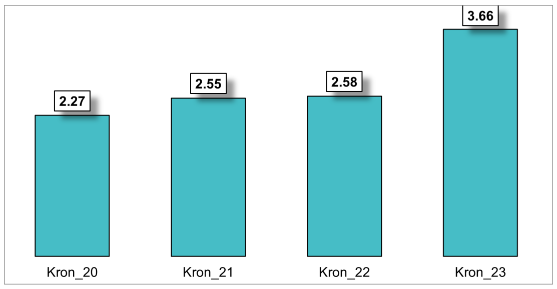


Fig. 4. Speed-up for parallel Bellman Ford implementation with respect to serial Bellman Ford on Kronecker graphs.

C. Evaluation of Implemented SSSP Approaches

For all SSSP algorithms, we initially generated weights for the graphs randomly within the [1,1000] range. We have used integer weights as the compared state of the art implementation also does so. Our experiments with Delta-Stepping involved testing different delta values, and the optimal value of delta varies for different graphs. We have used the best found delta for each dataset and for comparison, same delta is used for corresponding algorithms. We have implemented parallel Bellman-Ford algorithm and analyzed its performance and scalability. The Figure 4 and 5 shows the speed-up for Parallel Bellman-Ford algorithm both on synthetic and real world graph. As we have experimented using a machine with 16 core processor, the optimal runtime is found using around 16 threads for all of the parallel algorithms. We can see up to 3.66× speed-up for parallel implementation to it's serial version. For larger graphs it shows better speed up which indicates increase in the graph size does not affects the scalability. However, due to high repeated edge visits, the results are not comparatively better than Dijkstra's algorithm.

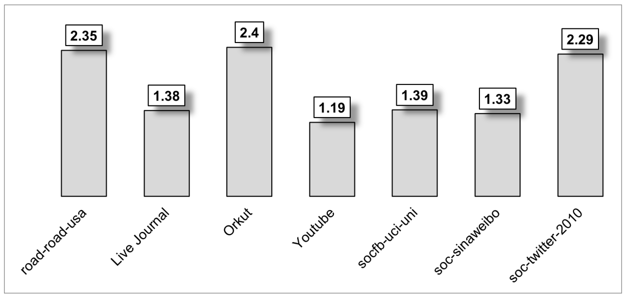


Fig. 5. Speed-up for parallel Bellman-Ford implementation with respect to serial Bellman-Ford on real-world graphs.

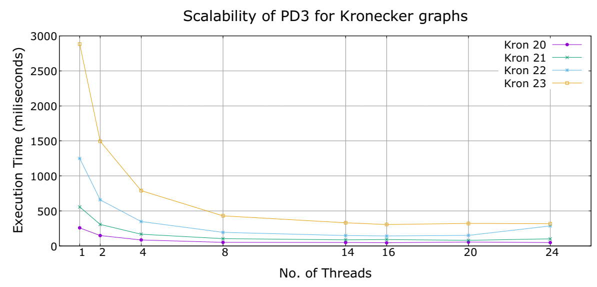


Fig. 6. Threads vs runtime in milliseconds for proposed PD3 implementation experimented on Kronecker graphs.

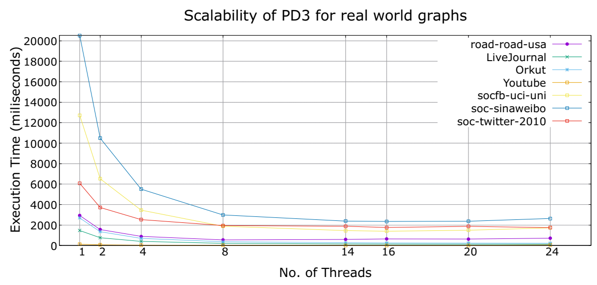


Fig. 7. Threads vs runtime in milliseconds for proposed PD3 implementation experimented on real-world graphs.

We conducted a thorough analysis on our PD3 implementation. PD3 implementation shows promising results and scalability for large graphs. Fig. 6 and Fig. 7 show number of threads vs runtime in milliseconds for PD3 on synthetic and real world graphs, respectively. We observe that the runtime reduces significantly till thread 16 for each of the graphs. The increase in graph size is not affecting the scalability. We observe good scalability for both large and small graphs. Fig. 8 and 9 show speed up for PD3 algorithm to its serial version for synthetic and real world graphs, respectively. We can see significant speed-up ranging from 5.4× to 9.38× for Kronecker graphs. With the increase in graph size, the performance is getting better. Similarly PD3 also exhibits excellent speed-up for real-world graphs as well ranging from 3.47× to 12.08×. The speed-up comparison for each parallel SSSP implementation with respect to serial Dijkstra's

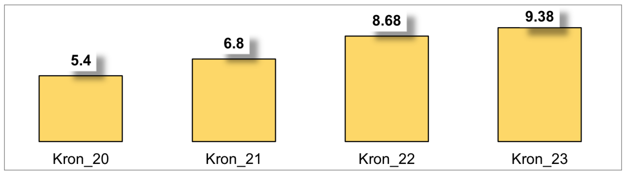


Fig. 8. Speed-up of proposed Parallel Delta-Stepping algorithm (PD3) with respect to single thread implementation on Kronecker graphs.

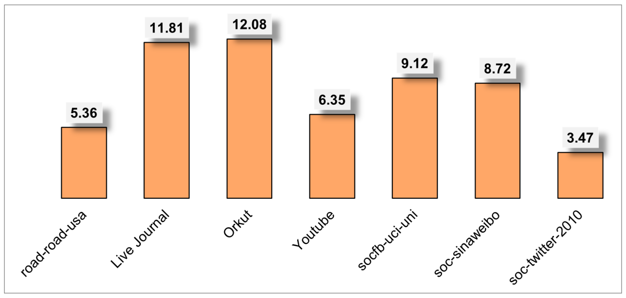


Fig. 9. Speed-up of proposed Parallel Delta-Stepping algorithm (PD3) with respect to single thread implementation on real-world graphs.

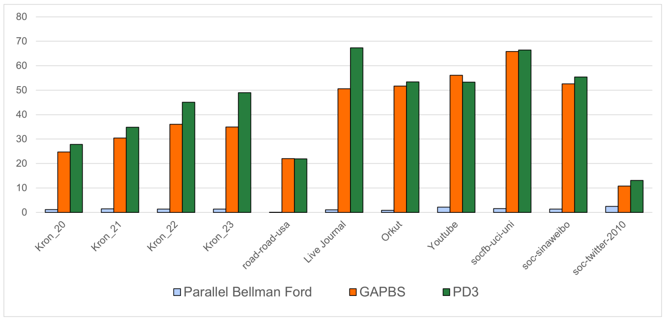


Fig. 10. Comparative speed-up (with respect to Dijkstra's algorithm) for implemented parallel SSSP algorithms.

TABLE II
COMPARATIVE RUNTIME (MS) OF DIFFERENT SSSP IMPLEMENTATIONS

	Serial	Parallel		
Graphs	**Dijkstra**	**Bellman-Ford**	**GAPBS**	**PD3**
Kron_20	1337	1134	54	48
Kron_21	2857	1979	94	82
Kron_22	6495	4765	180	144
Kron_23	15032	11305	430	307
road-road-usa	11991	751968	545	547
Live Journal	8347	7769	165	124
Orkut	11897	13322	230	223
Youtube	1066	496	19	20
socfb-uci-uni	92732	59112	1409	1396
soc-sinaweibo	130333	95591	2478	2353
soc-twitter-2010	22822	9472	2117	1749

algorithm is given in the Fig. 10. It is clearly visible that PD3 completely outperforms the parallel Bellman-Ford algorithm. PD3 also shows comparatively better speed-up and scalability than state of the art GAPBS [7] implementation of Parallel Delta-Stepping. Table II presents a comparative analysis of the runtime performance of PD3 against other state-of-the-art serial and parallel implementations. PD3 consistently demonstrates a significantly lower runtime compared to Dijkstra's algorithm and the parallel Bellman-Ford algorithm. Notably, the performance of PD3 remains stable even as graph sizes increase. Additionally, our PD3 implementation achieves a significantly lower or comparable runtime to the state-of-the-art GAPBS parallel Delta-Stepping implementation across all input graphs.

V. Conclusion

In this paper, we conducted extensive experiments to assess the performance of parallel SSSP algorithms. Our proposed PD3 algorithm shows substantial improvements in scalability and runtime compared to existing state-of-the-art serial and parallel SSSP implementations. The PD3 algorithm exhibits robust scalability across both synthetic and real-world graph datasets. While our study offers valuable insights into the performance of parallel SSSP algorithms on shared-memory systems, future work will focus on evaluating these algorithms in distributed memory environments. Additionally, since road networks do not exhibit the desired speed-up due to their inherent structure, alternative techniques could be explored for these specific cases. Further research can also investigate the use of additional tools, such as memory access hardware and GPU-based acceleration, to enhance the scalability and efficiency of SSSP algorithms.

Acknowledgment

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

References

[1] Graph500 benchmark. http://www.graph500.org/.

[2] Network data repository. https://networkrepository.com/.

[3] S. Abdelhamid, M. M. Alam, R. Alo, S. Arifuzzaman, et al. {CINET} 2.0: {A} cyberinfrastructure for network science. In *10th {IEEE} International Conference on e-Science, eScience 2014, Sao Paulo, Brazil, October 20-24, 2014*, pages 324–331, 2014.

[4] 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, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.

[5] S. Arifuzzaman, M. Khan, and M. Marathe. A space-efficient parallel algorithm for counting exact triangles in massive networks. In *Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications*, August 2015.

[6] S. Arifuzzaman, M. Khan, and M. Marathe. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Trans. Knowl. Discov. Data (TKDD)*, 14(1):5:1–5:34, 2019.

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

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

[9] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2031–2045, 2016.

[10] Y. Chi, L. Guo, and J. Cong. Accelerating sssp for power-law graphs. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 190–200, 2022.

[11] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–70, 2021.

[12] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 287–290. 2022.

[13] M. A. M. Faysal, S. Arifuzzaman, C. Chan, M. Bremer, D. Popovici, and J. Shalf. Hypc-map: A hybrid parallel community detection algorithm using information-theoretic approach. In *2021 IEEE High Performance Extreme Computing Conference (HPEC 2021)*, 2021.

[14] 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 2023, page 132–141, New York, NY, USA, 2023. Association for Computing Machinery.

[15] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, et al. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[16] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[17] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 23–35. SIAM, 2007.

[18] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang. Lagraph: A community effort to collect graph algorithms built on top of the graphblas. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 276–284. IEEE, 2019.

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

[20] 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.

[21] T. Panitanarak and K. Madduri. Performance analysis of single-source shortest path algorithms on distributed-memory systems. In *SIAM Workshop on Combinatorial Scientific Computing (CSC)*, page 60. Citeseer, 2014.

[22] A. Raval, R. Nasre, V. Kumar, S. Vadhiyar, K. Pingali, et al. Dynamic load balancing strategies for graph applications on gpus. *arXiv preprint arXiv:1711.00231*, 2017.

[23] N. S. Sattar and S. Arifuzzaman. Community detection using semi-supervised learning with graph convolutional network on gpus. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 5237–5246, 2020.

[24] N. S. Sattar and S. Arifuzzaman. Scalable distributed louvain algorithm for community detection in large graphs. *Journal of Supercomputing*, 78, 2022.

[25] 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 133–146, 2021.

[26] 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.

[27] 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 *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 1–12, 2016.

[28] H. Yang, H. Su, Q. Lan, M. Wen, and C. Zhang. High performance graph analytics with productivity on hybrid cpu-gpu platforms. In *Proceedings of the 2nd International Conference on High Performance Compilation, Computing and Communications*, pages 17–21, 2018.

[29] H. Yu, X. Wang, and Y. Luo. An edge-fencing strategy for optimizing sssp computations on large-scale graphs. In *Proceedings of the 50th International Conference on Parallel Processing*, pages 1–11, 2021.

[30] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun. Optimizing ordered graph algorithms with graphit. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 158–170, 2020.

[31] 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.