Extending the Shortest Path Algorithm for Large Graphs with Cycles and Parallel Computing Capabilities

Muon Ha[1(✉)], Yulia Shichkina[2], and Xuan-Hien Nguyen[3]

[1] Telecommunications University, Nha Trang, Vietnam
muon.ha@mail.ru
[2] St. Petersburg State Electrotechnical University, St. Petersburg, Russia
shichkina@co-evolution.ai
[3] Hanoi University of Industry, Ha Noi, Vietnam
hien.nguyen15@haui.edu.vn

Abstract. The challenge of finding the shortest path in graphs containing cycles and negative weights presents substantial difficulties across multiple fields, including transportation, social network analysis, and the study of complex systems. While sequential algorithms provide effective solutions for small to medium-sized graphs, they become impractical for large graphs due to constraints in processing time and computational resources. This paper presents an extension of the shortest path algorithm into a parallel computing environment, leveraging the power of modern multicore systems. Experimental results demonstrate that the parallel version achieves superior performance compared to traditional algorithms, making it suitable for handling large-scale graphs efficiently.

Keywords: Parallel Algorithm · Shortest Path · Negative Weights · Graphs with Cycles · Large Graphs

1 Introduction

Determining the shortest path in graphs represents a core challenge in graph theory and computer science, with a wide range of applications in areas such as transportation, network design, and social data evaluation. Classical algorithms such as Dijkstra, Bellman-Ford, and A* have proven effective in many specific scenarios. However, when applied to large graphs, especially those with cycles and negative weights, these algorithms face significant limitations. For instance, Dijkstra cannot handle negative weights, while Bellman-Ford encounters high computational costs when applied to dense graphs. These limitations necessitate the development of new algorithms that can not only solve the problem accurately but also scale effectively for large-scale graphs.

In practice, graph data is becoming increasingly large and complex, spanning global transportation networks, online social systems, and molecular graphs in medical research. These systems demand not only fast processing but also high accuracy,

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2026
O. Gervasi et al. (Eds.): ICCSA 2025 Workshops, LNCS 15894, pp. 191–204, 2026.
https://doi.org/10.1007/978-3-031-97648-3_13

particularly when graphs include cycles and negative weights, often encountered in economic analysis or biological system modeling. These requirements make traditional algorithms inadequate. Thus, the development of optimized algorithms, especially in parallel computing environments, has become a critical task for the computer science research community.

Modern computing systems, including supercomputers, cloud computing platforms, and GPUs, offer significant potential to enhance the performance of graph algorithms. Several studies have demonstrated that leveraging parallel processing can significantly accelerate the execution time of shortest path algorithms. However, in practice, parallel algorithms face considerable challenges. First, partitioning the graph for concurrent processing without compromising the problem's integrity is a complex issue. Second, load balancing among processes is essential to ensure efficient utilization of computing resources. Finally, the costs of inter-process communication and memory optimization remain open problems.

In our previous work [1], we introduced a sequential algorithm based on adjacency matrix representation to address the shortest path problem on graphs with cycles and negative weights. This algorithm overcame the limitations of Dijkstra and Bellman-Ford as well as opened new avenues for handling more complex graphs. However, when applied to large graphs, the algorithm revealed two major drawbacks: prolonged processing times and limited scalability. These issues motivated us to extend our research into the parallel computing domain, leveraging modern computational capabilities.

This research centers on the design of an innovative parallel algorithm, leveraging graph partitioning, adaptive load distribution, and communication enhancement. The approach boosts efficiency while maintaining scalability for extensive graph structures, addressing the computational demands of real-world scenarios including transportation systems, social data evaluation, and spatial information frameworks. Empirical evidence confirms that the developed algorithm surpasses conventional techniques in terms of execution speed and resource efficiency.

The paper is structured as follows: Sect. 2 examines prior research and assesses existing techniques. Section 3 outlines the proposed parallel algorithm and its technical enhancements. Section 4 provides experimental outcomes on substantial datasets, evaluating performance metrics. Finally, Sect. 5 summarizes the findings and explores potential research directions.

2 Related Works

The task of computing shortest paths in graphs serves as a critical focus within graph theory and computer science, enabling practical solutions in domains like traffic optimization, social connectivity modeling, communication systems, and resource distribution. Traditional methods, such as Dijkstra's and Bellman-Ford's algorithms, have been foundational solutions for this challenge. Dijkstra's algorithm [2] is predominantly utilized for graphs with non-negative weights due to its high computational effectiveness. Conversely, the Bellman-Ford algorithm is suitable for graphs with negative weights, though it encounters performance constraints with densely connected graphs [3, 4].

In recent years, many researchers have focused on improving these algorithms for large-scale graphs. Author in [5] use of Fibonacci Heaps significantly reduced the complexity of priority queue operations, making Dijkstra's algorithm particularly effective on dense graphs. Another approach introduced the Degree-Based Search algorithm, leveraging node degree metrics to optimize graph traversal and accelerate shortest path computation [6]. These methods both improved computational efficiency and reduced traversal complexity.

Parallel computing has significantly contributed to tackling the scalability issues encountered in shortest path computations across extensive graph structures. The Delta-stepping technique, a widely recognized approach, divides graphs into smaller subsets which facilitates simultaneous execution and minimizes data exchange between processes [7]. Enhanced implementations of Dijkstra's and Bellman-Ford's algorithms have demonstrated notable efficiency gains on multi-core and distributed platforms, though difficulties related to workload distribution and communication overheads remain unresolved [8, 9].

GPU-accelerated methods have also been developed to handle large-scale graphs efficiently. Algorithms like BFS and SSSP, accelerated by GPUs, leverage parallel processing capabilities to achieve substantial reductions in computation time. However, these methods face challenges related to memory optimization and thread balancing [10]. Other studies, such as Hyperlane, have combined dynamic graph partitioning with GPU processing to improve performance in distributed systems [11].

Shortest path algorithms have been applied across various domains. In transportation, parallelized A* algorithms have been utilized for real-time urban traffic routing, achieving high efficiency in large networks [12]. In wireless sensor networks, these algorithms are critical for enhancing energy efficiency and communication reliability [13]. In the field of telecommunications, the parallelized Johnson algorithm has been optimized to reduce communication costs across large-scale networks [14]. Several recent studies have also expanded the application of these algorithms to new domains such as social network analysis and robot navigation. Specifically, in social networks, the optimized algorithms have supported the identification of key influential groups, thereby improving the efficiency of information dissemination [15]. In addition, advanced models, such as image-blurring environments, have been proposed to address uncertainty in graph data, enabling applications that require precise data analysis [16].

Despite these advancements, the processing of large-scale graphs still encounters major challenges. Key issues such as achieving efficient graph partitioning for parallel execution without compromising accuracy, maintaining load balancing among processes, and minimizing communication costs have not been fully addressed yet. Moreover, integrating modern computing platforms like cloud systems and GPU-CPU hybrid architectures remains a promising research direction that has not been thoroughly explored.

This study proposes a new parallel algorithm that incorporates dynamic graph partitioning, asynchronous communication, and adaptive load balancing. These improvements aim to enhance performance and scalability, thus opening up new application opportunities in intelligent transportation, distributed systems, and large-scale data analysis.

3 Methodology

In our earlier work [1], we proposed a shortest path algorithm that utilizes the adjacency matrix representation of graphs to effectively tackle challenges posed by cycles and negative weights. The sequential algorithm was tested and shown to operate efficiently on medium-sized graphs, with the ability to identify and manage cycles while improving performance by removing redundant paths. However, when applied to large-scale graphs, this algorithm encountered limitations in processing time and computational resources.

To overcome these limitations, we extended our research to a parallel computing environment, leveraging the power of modern multicore processing systems. Implementing the algorithm in a parallel environment not only significantly reduced processing time but also expanded its applicability to large and complex graphs.

The main idea of deploying the algorithm in a parallel environment is to partition the graph's adjacency list into smaller subsets, each assigned to independent processes. Each process performs local computations and exchanges information with other processes to merge results, ensuring the integrity of the problem-solving process. This approach enhances scalability while maintaining accuracy in solving the shortest path problem.

General Process Overview:

Step 1: Data Partitioning. In this step, the adjacency list of the graph is divided into subsets, with each part containing a subset of edges.

Step 2: Local Computation. The processes compute the sets of vertices (V_1, V_2) and determine the local difference set (V').

Step 3: Information Exchange**.** Processes exchange information. Each process resolves its local calculations and merges the results into the union set (U', U'').

Step 4: Merging and Iteration. The results from all processes are merged to construct the generalized set $V = \Delta' \cup \Delta''$ with $\Delta' = V'' - U'$ and $\Delta'' = V' - U''$ which are the difference sets, and the computation continues until the optimal path is achieved.

To specifically illustrate the implementation of the shortest path algorithm in a parallel environment with two threads, the detailed procedure is described as follows:

1. Initialization

$M_{new} = \{\}$ and $M_{old} = \{\}$ Initialize the set representing new and old representative sets.

The graph is represented as an adjacency list, in which:

V_1: The set of starting vertices.

V_2: The set of target vertices.

P: : The set of edges (length) in the graph.

2. Partitioning the Adjacency List

Divide the adjacency list into two equal parts:

- **Thread 1:** Processes the first half of the adjacency list.
- **Thread 2:** Processes the second half of the adjacency list.

3. Find the Difference Set: $V = V_1 - V_2$

The computation is performed in each thread as follows:

3.1. Determine Local Difference Sets:

Thread 1: $V' = V_1' - V_2'$.

Thread 2: $V'' = V_1'' - V_2''$.

3.2. Exchange Information Between Threads:

Thread 1 sends V' to Thread 2 and receives V''.

Thread 2 sends V'' to Thread 1 and receives V'.

3.3. Merge the Results:

Thread 1: Compute the intersection $U' = V_2' \cap V''$, then $\Delta' = V'' - U'$.

Thread 2: Compute the intersection $U'' = V_2'' \cap V'$, then $\Delta'' = V' - U''$.

Final merge: $V = \Delta' \cup \Delta''$.

4. Merge Paths

Edges in the adjacency list are merged if the following conditions are met:

- The starting vertex of the edge belongs to V.
- The ending vertex matches the starting vertex of another edge.

5. Remove Invalid Edges

Edges in the adjacency list are removed if:

- The ending vertex belongs to the set of starting vertices.
- The edge does not lead to the destination vertex of the target path.

6. Update Optimal Paths

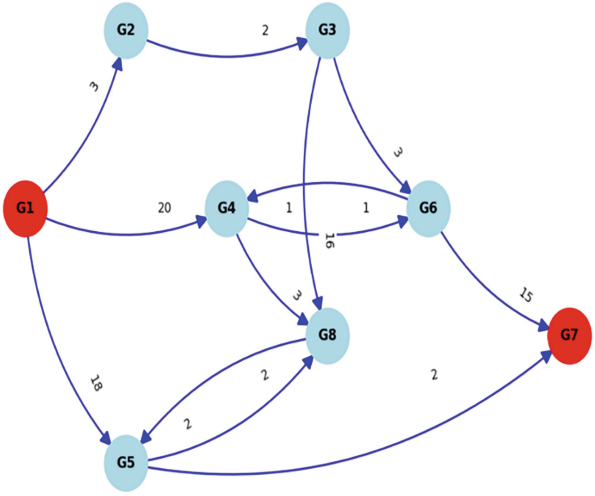
If $P(M_{old}) > P(M_{new})$: update M_{new}.

If $P(M_{old}) = P(M_{new})$, choose the shorter path.

7. Repeat

Repeat the steps with the updated adjacency list until no vertex in V_1 belongs to V_2.

Example: Given the following graph with the adjacency list below:



V_1	V_2	P
G_1	G_2	3
G_1	G_4	20
G_1	G_5	18
G_2	G_3	2
G_3	G_6	3
G_3	G_8	16
G_4	G_6	1
G_4	G_8	3
G_5	G_8	2
G_5	G_7	2
G_6	G_4	1
G_6	G_7	15
G_8	G_5	2

The algorithm processes in parallel on 2 threads to find the shortest path from vertex G_1 to vertex G_7, as detailed below:

<table>
<tr><th>Thread 1</th><th>Thread 2</th></tr>
<tr><td>
<table>
<tr><th>V_1</th><th>V_2</th><th>P</th></tr>
<tr><td>G_1</td><td>G_2</td><td>3</td></tr>
<tr><td>G_1</td><td>G_4</td><td>20</td></tr>
<tr><td>G_1</td><td>G_5</td><td>18</td></tr>
<tr><td>G_2</td><td>G_3</td><td>2</td></tr>
<tr><td>G_3</td><td>G_6</td><td>3</td></tr>
<tr><td>G_3</td><td>G_8</td><td>16</td></tr>
<tr><td>G_4</td><td>G_6</td><td>1</td></tr>
</table>
V_1'= {G_1, G_2, G_3, G_4}
V_2'= {G_2,G_3,G_4,G_5,G_6,G_8}
</td><td>
<table>
<tr><th>V_1</th><th>V_2</th><th>P</th></tr>
<tr><td>G_4</td><td>G_8</td><td>3</td></tr>
<tr><td>G_5</td><td>G_8</td><td>2</td></tr>
<tr><td>G_5</td><td>G_7</td><td>2</td></tr>
<tr><td>G_6</td><td>G_4</td><td>1</td></tr>
<tr><td>G_6</td><td>G_7</td><td>15</td></tr>
<tr><td>G_8</td><td>G_5</td><td>2</td></tr>
<tr><td>G_8</td><td>G_7</td><td>16</td></tr>
</table>
V_1''= {G_4, G_5, G_6, G_8}
V_2''={G_4,G_5,G_8, G_7}
</td></tr>
<tr><td colspan="2">Step 3.</td></tr>
<tr><td>3.1. V'=V_1'-V_2'={G_1}</td><td>3.1. V''=V_1''-V_2''={G_6}</td></tr>
<tr><td>3.2 Exchange V' and V''
V''={G_6}</td><td>3.2 Exchange V' and V''
V'={G_1}</td></tr>
<tr><td>3.3. Intersection
U'=V2'∩V''={G_6}</td><td>3.3. Intersection
U''=V_2''∩V'=Ø</td></tr>
<tr><td>3.4. Difference
Δ'=V''-U'=Ø</td><td>3.4. Difference
Δ''=V'-U''={G_1}</td></tr>
<tr><td colspan="2">3.5. Union
V=Δ'∪Δ'' ={G_1}</td></tr>
<tr><td colspan="2">Step 4:</td></tr>
</table>

Send to the Thread 1:

V_1	V_2	P
G_1	G_2	3
G_1	G_4	20
G_1	G_5	18

Receive from Thread 2

V_1	V_2	P
G_1	G_2	3
G_1	G_4	20
G_1	G_5	18

Merge data between two threads

V_1	V_2	P
G_2	G_3	2
G_3	G_6	3
G_3	G_8	16
G_4	G_6	1
G_4	G_8	3
G_5	G_8	2
G_5	G_7	2
G_6	G_4	1
G_6	G_7	15
G_8	G_5	2
G_8	G_7	16

Split into two parts to send each part to each thread

V_1	V_2	P
G_5	G_7	2
G_6	G_4	1
G_6	G_7	15
G_8	G_5	2
G_8	G_7	16

V_1	V_2	P
G_5	G_7	2
G_6	G_4	1
G_6	G_7	15
G_8	G_5	2
G_8	G_7	16

V_1	V_2	P
G_1G_2	G_3	5
G_1G_4	G_6	21
G_1G_4	G_8	23
G_1G_5	G_8	20
G_2	G_3	2
G_3	G_6	3
G_3	G_8	16
G_4	G_6	1
G_4	G_8	3
G_5	G_8	2

V_1	V_2	P
G_1G_5	G_7	20
G_5	G_7	2
G_6	G_4	1
G_6	G_7	15
G_8	G_5	2
G_8	G_7	16

Receive from Thread 2
M_{new}= {G_1, G_5, G_7}, P=20

Calculate: M_{new}= {G_1, G_5, G_7}, P=20
Send to the Thread 1

Receive from Thread 2			Delete the green row and Send to the Thread 1:		
V_1	**V_2**	**P**	**V_1**	**V_2**	**P**
G_5	G_7	2	G_5	G_7	2
G_6	G_4	1	G_6	G_4	1
G_6	G_7	15	G_6	G_7	15
G_8	G_5	2	G_8	G_5	2
G_8	G_7	16	G_8	G_7	16
Reformat the input for the next iteration			Reformat the input for the next iteration		
V_1	**V_2**	**P**	**V_1**	**V_2**	**P**
G_1G_2	G_3	5	G_1G_2	G_3	5
G_1G_4	G_6	21	G_1G_4	G_6	21
G_1G_4	G_8	23	G_1G_4	G_8	23
G_1G_5	G_8	20	G_1G_5	G_8	20
G_2	G_3	2	G_5	G_7	2
G_3	G_6	3	G_6	G_4	1
G_3	G_8	16	G_6	G_7	15
G_4	G_6	1	G_8	G_5	2
G_4	G_8	3	G_8	G_7	16
G_5	G_8	2			
$V1' = \{G_1G_2, G_1G_4, G_1G_5, G_2, G_3, G_4\}$ $V2' = \{G_3, G_6, G_8\}$			$V1'' = \{G_1G_2, G_1G_4, G_1G_5, G_5, G_6, G_8\}$ $V2'' = \{G_3, G_6, G_8, G_7, G_4, G_5\}$		
Repeat step 3					
3.1. $V'=V1'-V2' = \{G_1G_2, G_1G_4, G_1G_5, G_2, G_4\}$			3.1. $V''=V1''-V2'' = \{G_1G_2, G_1G_4, G_1G_5, G_6, G_8\}$		
3.2 Exchange V' and V'' $V'' = \{G_1G_2, G_1G_4, G_1G_5, G_6, G_8\}$			3.2 Exchange V' and V'' $V' = \{G_1G_2, G_1G_4, G_1G_5, G_2, G_4\}$		
3.3. Intersection $U'=V2' \cap V'' = \{G_6, G_8\}$			3.3. Intersection $U''=V2'' \cap V' = \{G_4\}$		
3.4. Difference $\Delta'=V''-U' = \{G_1G_2, G_1G_4, G_1G_5\}$			3.4. Difference $\Delta''=V'-U'' = \{G_1G_2, G_1G_4, G_1G_5, G_2\}$		
3.5. Union $V=\Delta' \cup \Delta'' = \{G_1G_2, G_1G_4, G_1G_5, G_2\}$					
…					
Final result: $M_{new} = \{G_1, G_2, G_3, G_6, G_4, G_8, G_5, G_7\}$, P = 16					

4 Experiments and Evaluation

4.1 Experimental Setup

The development of the parallel shortest path algorithm was executed in the C ++ programming language, utilizing *Boost.MPI* as the core infrastructure for coordinating communication between processes. The application is tailored for operation across distributed computing platforms, capitalizing on *Boost.MPI*'s features for message transmission and Boost.Serialization for optimized data transfer among processes. Evaluations were conducted on a system featuring an Intel Core i7-12700 processor, 32GB of memory, and operating on Ubuntu 22.04.

The program begins by reading the graph data from an input file, which is expected to be in adjacency list format. The graph data is divided among multiple processes, with the number of processes determined dynamically during execution. Each process handles a portion of the graph's edges, assigned through a modulo-based partitioning approach to ensure an even distribution of workload. This partitioning strategy enables each process to perform local computations independently, thereby minimizing the frequency of inter-process communication.

The algorithm operates in an iterative manner, with each process executing the following steps:

Step 1: Local Processing. Each process calculates the shortest paths for its allocated edge subset. This step involves updating local distance tables by evaluating differences between vertex sets (denoted as V_1 and V_2) and performing set operations such as union, intersection, and difference to refine the computed paths.

Step 2: Data Synchronization. Upon completing its local computations in each iteration, each process shares intermediate results with others using Boost.MPI's all-reduce and point-to-point communication methods. This step ensures consistency across all processes while keeping communication overhead low.

Step 3: Convergence Check. Each process evaluates whether any updates to the distance tables occurred in the current iteration. This information is then shared among all processes, and the algorithm terminates if no updates are observed.

The implementation is designed to adapt to varying process counts, enabling the program to scale seamlessly from a single machine to a distributed cluster environment. For efficient data exchange, the *Boost.Serialization* library is employed to handle the serialization and deserialization of complex structures, such as graph edges and distance tables, facilitating smooth communication between processes. This approach provides a versatile and reliable framework for executing the parallel shortest path algorithm across diverse configurations, accommodating a number of graph sizes and computational resources.

4.2 Performance Evaluation

The performance, scalability, and advantages of the proposed parallel algorithm were assessed through a series of experiments. Synthetic graphs with vertex counts ranging from 100 to 100,000 were used, covering both sparse and dense graph structures. The evaluation was conducted in three distinct phases to thoroughly analyze the algorithm's effectiveness.

In the first phase, the focus was on identifying the optimal thread count for parallel execution of the algorithm. Results indicate that the execution time decreases as the number of threads increases from 2 to 4, achieving the best performance at 4 threads. However, as the thread count increases to 6, 8, and 10, the execution time rises again, particularly for larger graphs (100,000 vertices). This indicates that the algorithm does not scale linearly with the amount of threads and is affected by synchronization overhead and resource contention.

Table 1 and Fig. 1 presents the execution times of the proposed algorithm for different graph sizes and various numbers of threads. The sequential execution times listed in the

table were estimated based on the parallel execution times with two threads, considering typical parallelization efficiency and synchronization overhead.

Table 1. Execution Time with Different Numbers of Threads (ms)

Graph Size (|V|)	Sequential	Parallel				
		Threads (T = 2)	T = 4	T = 6	T = 8	T = 10
100	121	107	83	102.5	108	111.5
1,000	4323	3941	2982	3556	3928	4177
10,000	6783	7612	4503	6533	6698	6768
100,000	17320	14494	12459	14384	15465	16342

Figure 2 illustrates the speedup factor as a function of the number of threads. The speedup factor of the algorithm increases as the number of threads rises from 2 to 4, reaching its peak at 4 threads. Beyond this point, speedup decreases, especially for larger graphs (10,000 and 100,000 vertices). This confirms that after an optimal threshold, the cost of inter-thread communication and synchronization outweighs the benefits of parallelism, reducing efficiency. The speedup (S) was calculated using the formula: $S = \frac{T_{sequential}}{T_{parallel}}$, where $T_{sequential}$ is the execution time with one thread and $T_{parallel}$ is the execution time with multiple threads.

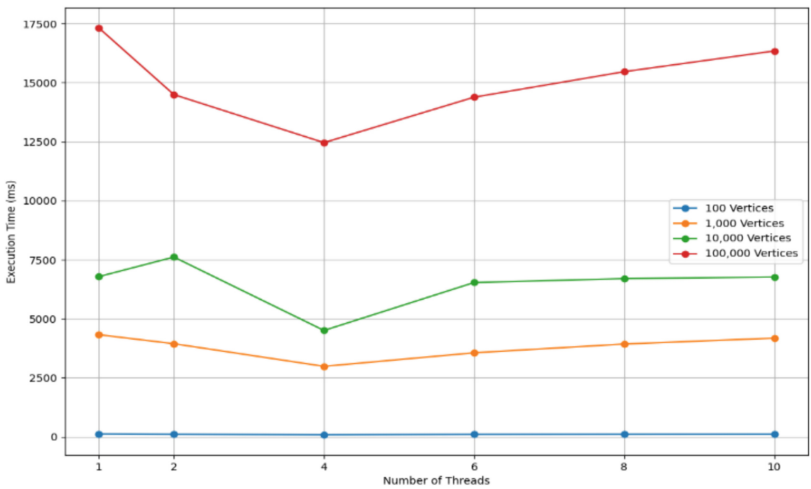


Fig. 1. Execution Time by Number of Threads

The results produced by the parallel version of the proposed algorithm are consistent with those obtained from the sequential execution. There is no observed discrepancy in the final output between the two versions, confirming the correctness and reliability of the parallel implementation.

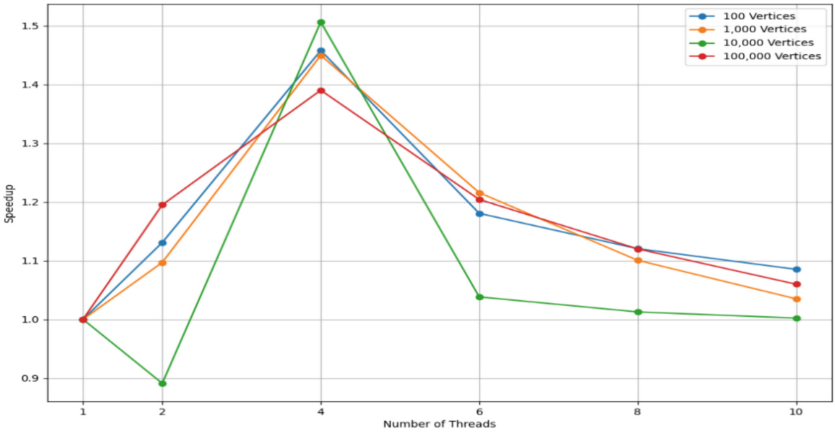


Fig. 2. Algorithm Speedup as a Function of the Number of Threads

Figure 3 compares the execution time of the Bellman-Ford, Dijkstra, and the proposed algorithm, all implemented with 4 parallel threads, across different graph sizes. Experimental results compare the execution times of Bellman-Ford, Dijkstra, and the proposed algorithm across different graph sizes using 4 threads.

- For 100 vertices, execution times of all three algorithms are relatively similar.
- As the graph size increases to 1,000, 10,000, and 100,000 vertices, the proposed algorithm consistently achieves the lowest execution time, whereas Bellman-Ford has the highest due to its higher computational complexity.
- Dijkstra performs better than Bellman-Ford but remains slower than the proposed algorithm, particularly for larger graphs.

Experiments show that the proposed algorithm achieves optimal performance with 4 threads, significantly reducing execution time compared to traditional algorithms (Figs. 1 and 2). However, increasing the thread count further leads to performance degradation due to synchronization overhead and memory contention, indicating that scalability depends on data size and hardware architecture.

Compared to Bellman-Ford and Dijkstra (Fig. 3), the proposed algorithm is superior, especially on large graphs. Bellman-Ford has the highest execution time. The proposed algorithm reduces execution time and maintains better efficiency as graph size increases.

It is a parallelized extension of our previously introduced sequential method, which was capable of handling graphs with cycles, negative weights, and even negative cycles. Unlike Dijkstra's algorithm, which fails with negative edge weights, and Bellman-Ford's algorithm, which suffers from high time complexity, the proposed parallel algorithm maintains both correctness and efficiency in these challenging cases. This ensures better scalability and applicability to complex real-world graph problems.

To further optimize performance, improvements in synchronization mechanisms and adaptive thread management can help enhance scalability for real-world applications such as intelligent transportation, social network analysis, and geographic information systems.

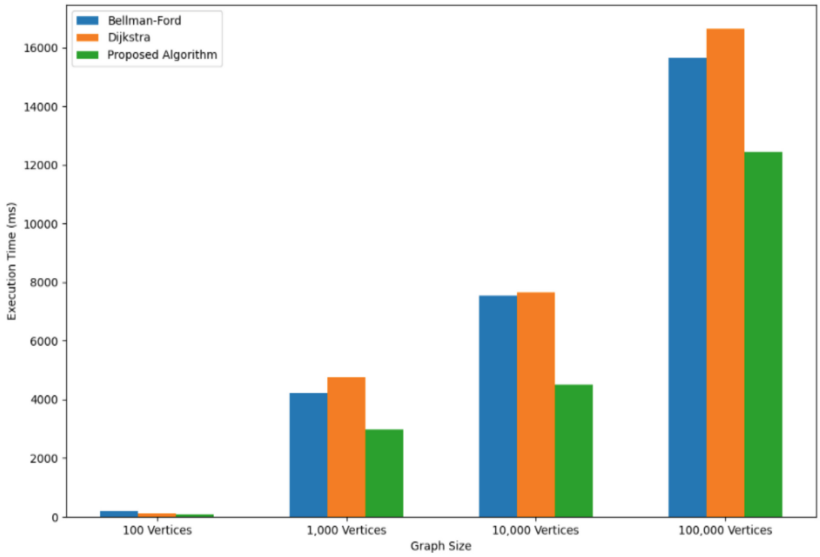


Fig. 3. Execution Time Comparison: Bellman-Ford, Dijkstra, and Proposed Algorithm (4 Threads)

4.3 Load Balancing Strategy

In the parallel implementation we propose, a fixed load distribution approach was implemented. The graph is divided among the available threads by allocating vertices (or edges) as uniformly as possible across them. Each thread handles a unique set of vertices along with their associated outgoing edges. This method reduces the need for synchronization by allowing threads to work independently during the relaxation process as much as possible. Although this static division works well for graphs with even distribution, adapting dynamic load adjustment techniques could improve performance when dealing with highly irregular graphs or those with uneven degree distributions.

5 Conclusion

This study focuses on the implementation and optimization of an algorithm in a parallel computing environment to solve the shortest path problem on large graphs. By leveraging the power of modern multi-threaded systems, the proposed parallel algorithm has demonstrated its ability to efficiently handle graphs ranging from thousands to hundreds of thousands of vertices, including both sparse and dense graphs. Through the partitioning of adjacency lists and the execution of computation steps concurrently across multiple threads, the algorithm not only reduces execution time but also maintains high accuracy in the results.

Experimental findings indicate that the parallel algorithm not only outperforms conventional algorithms such as Dijkstra and Bellman-Ford but also demonstrates near-linear scalability with an increasing number of processing threads. Notably, the algorithm

capitalizes on the characteristics of parallel environments to optimize computational resources, especially for dense graphs.

Acknowledgment. This research has been supported by the Ministry of Science and Higher Education (assignment No. FSEE-2025-0015).

References

1. Shichkina, Y., Nguyen, X.H., Ha, M., Tran, D.M.: Shortest path search method on a graph with cycles. In: Gervasi, O., Murgante, B., Garau, C., Taniar, D., C. Rocha, A.M.A., Faginas Lago, M.N. (eds.) Computational Science and Its Applications – ICCSA 2024 Workshops. ICCSA 2024. Lecture Notes in Computer Science, vol. 14825. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-65343-8_26
2. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959). https://doi.org/10.1007/BF01386390
3. Bellman, R.: On a routing problem. Q. Appl. Math. **16**(1), 87–90 (1958)
4. Qiu, L., Li, Z., Ke, X., Chen, L., Gao, Y.: Accelerating biclique counting on GPU. In: Proceedings of the 40th IEEE International Conference on Data Engineering (ICDE 2024), pp. 3191–3203. IEEE, Utrecht (2024). https://doi.org/10.1109/ICDE60146.2024.00247
5. Verma, D., Messon, D., Rastogi, M., Singh, A.: Comparative study of various approaches of Dijkstra algorithm. In: Proceedings of the 2021 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS 2021), pp. 328–336. IEEE, Greater Noida (2021). https://doi.org/10.1109/ICCCIS51004.2021.9397200
6. Shyma, P.V., Sanil, S.K.P.: Degree-based search: a novel graph traversal algorithm using degree-based priority queues. Int. J. Adv. Comput. Sci. Appl. **15**(7), 1366–1371 (2024). https://api.semanticscholar.org/CorpusID:271699515
7. Sridhar, U., Blanco, M.P., Mayuranath, R., Spampinato, D.G., Low, T.M., McMillan, S.: Delta-stepping SSSP: from vertices and edges to GraphBLAS implementations. In: Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2019), pp. 241–250. IEEE, Rio de Janeiro (2019). https://doi.org/10.1109/IPDPSW.2019.00047
8. Zhang, B., Hu, D.J.: RETRACTED ARTICLE: Research on the construction and simulation of PO-Dijkstra algorithm model in parallel network of multicore platform. J. Wirel. Commun. Netw. **2020**, 85 (2020). https://doi.org/10.1186/s13638-020-01680-x
9. Safari, M., Oortwijn, W., Huisman, M.: Automated verification of the parallel Bellman–Ford algorithm. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. (eds.) Static Analysis. SAS 2021. Lecture Notes in Computer Science(), vol. 12913. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88806-0_17
10. Aher, S.N., Walunj, S.M.: Accelerate the execution of graph processing using GPU. In: Satapathy, S., Joshi, A. (eds.) Information and Communication Technology for Intelligent Systems. Smart Innovation, Systems and Technologies, vol. 106. Springer, Singapore (2019). https://doi.org/10.1007/978-981-13-1742-2_13
11. Marcucci, T., Umenberger, J., Parrilo, P., Tedrake, R.: Shortest paths in graphs of convex sets. SIAM J. Optim. **34**(1), 507–532 (2024). https://doi.org/10.1137/22M1523790
12. Bhatia, J., Dave, R., Bhayani, H., Tanwar, S., Nayyar, A.: SDN-based real-time urban traffic analysis in VANET environment. Comput. Commun. **149**, 162–175 (2020). https://doi.org/10.1016/j.comcom.2019.10.011

13. Mostafaei, H.: Energy-efficient algorithm for reliable routing of wireless sensor networks. IEEE Trans. Industr. Electron. **66**(7), 5567–5575 (2019). https://doi.org/10.1109/TIE.2018.2869345
14. Joshi, S., Shanmuganathan, M., Aljuhani, A., Albalawi, U., Aljaedi, A.: Energy-efficient and secure wireless communication for telemedicine in IoT. Comput. Syst. Sci. Eng. **43**, 1111–1130 (2022). https://doi.org/10.32604/csse.2022.024802
15. Šumak, B., Pušnik, M.: Analysis of the Shortest Path Method Application in Social Networks. In: Information Modelling and Knowledge Bases XXXIV, pp. 169–182. IOS Press (2023). https://doi.org/10.3233/FAIA220500
16. Zuo, C., Pal, A., Dey, A.: New concepts of picture fuzzy graphs with application. Mathematics **7**(5), 470 (2019). https://doi.org/10.3390/math705047