Parallel and Distributing Project - CSE317

Parallel Optimization of Shortest Path Algorithms: Accelerating BFS, Bellman-Ford, and Dijkstra for Large-Scale Graphs

Haseeb Ahmed
Institute of Business Administration
ERP:26077

Abdullah Iqbal
Institute of Business Administration
ERP:26904

Azizullah Khan
Institute of Business Administration
ERP:24931

***Abstract*—Shortest path algorithms are fundamental to diverse applications, including transportation networks, robotics, and artificial intelligence. However, as data volumes grow and real-time computation demands increase, traditional sequential implementations of these algorithms struggle to maintain efficiency. Classical approaches such as Dijkstra's, A*, and Bellman-Ford exhibit significant computational overhead when handling large graphs, limiting their applicability in mission-critical scenarios. To address these challenges, parallel and distributed computing techniques offer promising solutions to enhance execution speed and scalability.**

This research investigates the performance of Dijkstra's, A*, and Bellman-Ford algorithms in both serial and parallel computing environments. By implementing these algorithms using parallelization strategies, we aim to optimize execution time while analyzing scalability and power efficiency. The study seeks to identify the most effective parallel approach for reducing computational bottlenecks and improving real-world applicability. Through comprehensive benchmarking and performance evaluation, this research contributes to advancements in high-performance computing and the optimization of shortest path computations for large-scale systems

I. Introduction

Shortest path algorithms are essential in numerous applications, such as transportation networks, robotics, computer networks, and artificial intelligence. Owing to exponential data growth and the growing demand for real-time computations, sequential implementations of these algorithms tend to be unable to achieve the best performance. As the size of graphs grows, classical shortest path algorithms like Dijkstra's, A*, and Bellman-Ford become computationally costly and result in inefficient processing times in mission-critical real-world applications. In order to solve these problems, researchers have investigated parallel and distributed computing methods to make these algorithms efficient, thus greatly enhancing their processing time and scalability. In this research, we seek to compare and contrast the performance of three most popular shortest path algorithms—Dijkstra's, A*, and Bellman-Ford—under both serial and parallel computing environments. We want to implement these three algorithms under various parallelization strategies and analyze their efficiency in terms of the time of execution, scalability, and power usage. Through the exploitation of parallel processing methodologies like task parallelism and graph partitioning we aim to reduce computational bottlenecks and increase computational efficiency. Our ambition is to attain a sizable decrease in execution time in at least one of the algorithms so that there are definite gains for real-world applications.

A. Significance of the Study

The importance of this research is that it can have a vital influence on numerous areas where shortest path calculations play a key role. In transport networks, optimized routes can save travel time, fuel, and enhance logistics. In robotics, path planning is critical for real-time autonomous travel. Network routing and game AI also depend much on efficient and rapid shortest path calculations. Through the examination of parallel applications of shortest path algorithms, this research hopes to contribute to the general area of high-performance computing and its extensions to large-scale optimization problems. Although various efforts have been made to parallelize shortest path algorithms, available methods tend to be plagued with synchronization overhead, poor workload allocation, and limited memory. Existing work has shown that domain-specific optimizations, e.g., exploiting specialized data structures and hardware acceleration, can make a huge difference in algorithmic performance. Nevertheless, scalability is still a serious challenge, especially for large and dynamic graphs. This research will fill these voids by systematically analyzing the advantages and limitations of various parallelization methods applied to Dijkstra's, A*, and Bellman-Ford algorithms.

B. Objectives

The main goal of this study is to examine and compare the performance of three shortest path algorithms—Dijkstra's, A*, and Bellman-Ford—on both sequential and parallel computing systems. The research seeks to find out how varying parallelization strategies impact execution time and computational efficiency and whether one or more of the algorithms can obtain a considerable reduction in runtime by parallel implementation.

To accomplish this, the following objectives have been stated:

1) Comparison of Sequential and Parallel Implementations:

- Run Dijkstra's, A*, and Bellman-Ford algorithms on a synthetic large dataset simulating a road network.

- Run each algorithm both in a sequential and a parallel environment in order to analyze computational performance.
- Perform a detailed comparative study of execution time fluctuations, efficiency gains, and computational overhead minimization.

2) Performance Metric Evaluation:

- Execution Time Reduction: Calculate the time taken by each algorithm in sequential and parallel modes to identify which provides the maximum speedup.
- Multi-threading and Vector Processing Impact: Assess how effectively multi-threading (using OpenMP) and vector processing optimize computational speed.
- Scalability Testing: Examine how the algorithms perform when the dataset size increases, testing their ability to handle large-scale graphs.
- Synchronization Overhead Analysis: Investigate how synchronization affects multi-threaded performance and identify potential bottlenecks.
- Assessment of Memory Usage: Analyze how various parallelization approaches affect memory usage so that performance improvement is not balanced out by excessive usage of resources.

3) Parallel Implementation Experimentation of Algorithms:

- Develop parallel implementations of every algorithm with various optimization methods, such as OpenMP for CPU parallelization and CUDA for GPU (if feasible after experimentation).
- Investigate graph partitioning techniques to improve workload distribution.
- Compare the efficiency of static vs. dynamic workload balancing strategies.
- Apply vectorized implementations to optimize memory access patterns.
- Perform benchmarking using real-time performance profiling tools to track improvements.

C. Methodology

This research will have a formal approach with the entire emphasis being on the software and programming aspects of parallel computing owing to practical limitations. The approach includes three main stages: **algorithm implementation, experimental setup, and comparative analysis.**

1) Algorithm Implementation: We will create both sequential and parallel implementations of the three shortest path algorithms. This is to enable a baseline test of the basic execution versus optimal parallel execution. The implementation tactics include:

Sequential Implementation

- Common implementations of Dijkstra's, A*, and Bellman-Ford will be done using routine graph traversal.
- The sequential versions will be used as a basis for quantifying improvements made in parallel execution.

Parallel Implementations

Based on our deep study from the papers we have searched some of the techniques which we will studying before reaching out to finalize any techniques. We have also pulled out their pros and cons as discussed below.

i. CPU-based Parallelization

Implementation: OpenMP shall be employed for the distribution of graph traversal jobs among multiple CPU cores, so that shortest path computations can be executed concurrently. A subset of the graph is handled by each thread independently, lowering the execution time overall.

Benefits:

- Efficient for systems with shared memory since multiple cores can operate over the same data.
- Lowers execution time since computation is done over multiple threads.
- Needs minimum code changes when compared to other parallel methods.

Limitations:

- Synchronization overhead can be experienced when several threads share common data structures.
- Performance improvement is subject to the number of cores and workload distribution efficiency.
- Does not necessarily scale for very large graphs as a result of memory bandwidth constraints.

ii. Vectorized Execution

Implementation: SIMD will be used to handle many vertices or edges in parallel in a single CPU core. This method will be used to speed up relaxation operations in shortest path calculations.

Benefits:

- Decreases computation time by running the same operation on many points at once.
- Suitably employed in regular graph structures where there are opportunities for parallelism.
- Supported by current CPU architectures with compiler-level optimization.

Limitations:

- Algorithm restructuring is needed to utilize vector registers to their extent.

- Performance gains vary based on the graph topology; non-regular graphs might not experience considerable improvements.
- Not every CPU has support for advanced vectorization (e.g., AVX-512), which constrains portability.

In order to provide a reasonable comparison, the same data structures and graph representations will be utilized in both sequential and parallel versions whichever we will use.

D. Experimental Setup

A well-designed experimental apparatus is necessary to perform credible performance testing. The apparatus contains:

1) Dataset Generation:

- A synthetic dataset will be created with the aim to mimic a large-scale road network such that there are realistic node and edge distributions.
- The dataset will have differing vertices and edges to test scalability in differing graph sizes.

2) Execution Environments:

- The tests will be run on a multi-core CPU system using OpenMP-based parallel execution.
- If CUDA-based GPU implementation is possible following initial testing, the algorithms will also be run on GPU hardware.

3) Benchmarking Process:

- Multi-threaded and vectorized codes will be benchmarked against their sequential counterparts to estimate performance improvements.
- Execution time, CPU/GPU usage, memory used, and speedup ratios will be measured.
- All the algorithms will be run in sequential and parallel modes with the same datasets.

4) Comparative Analysis: Comparative analysis will be thoroughly executed to measure the extent of parallelization effects. Analysis methodologies are:

Performance Profiling

- Execution time will be recorded for every implementation.
- CPU and GPU usage will be tracked to evaluate computation efficiency.

Scalability Testing

- The tests will be rerun on larger dataset sizes to measure how well each algorithm scales with the size of the graph.
- This shall aid in evaluation of the efficacy of parallelization in solving high-scale shortest path problems.

Parallel Efficiency Measurement

- Speedup ratios shall be determined between the execution times of parallelized versions and the sequential ones.
- Efficiencies lost from synchronization, memory overhead, and distribution of work shall be inspected.

Graph Structure Impact Analysis

- The algorithms will be run on graphs of different topologies (dense vs. sparse graphs) to check if some graph structures are more improved by parallelization.

Synchronization Overhead Study

- Various synchronization methods will be experimented with to quantify their effect on execution time and computational bottlenecks.

This approach guarantees a thorough examination of parallel shortest path computations, offering insights into efficient methods for enhancing algorithmic performance.

II. Literature Review

A. A Algorithm*

The increasing demand for computational efficiency, scalability, and usage of hardware in handling complex pathfinding and optimization problems has spurred the evolution of parallel heuristic search algorithms, particularly A* and its variations. This review synthesizes the most significant developments from seven key papers, arranged thematically to draw attention to trends, innovations, and unresolved issues in the field.

1) Foundations of Parallel A:* The early work in 2012 by Kishimoto et al. served as a foundation for parallel best-first search (PBFS) by incorporating hash distribution and load balancing. Although speedup near the linear was achieved in graph search problems, they succumbed to synchronization overhead when tried out in large-scale problems. In extending this work, Fukunaga and Kishimoto (2017) studied parallel A*-variants, with a key focus on the trade-offs involved between centralized and decentralized architectures. They noted that domain-specific optimizations, either in robotics or game AI, offer better performance than generic implementations because of dedicated heuristic functions. Nonetheless, synchronization implementations in both studies identified any memory bottleneck in shared open/closed lists as a major hindrance to scalability.

2) Memory Efficiency and Structured Search: Zhou and Hansen (2020) presented the Parallel Structured Duplicate Detection (PSDD) algorithm, which resolves memory constraint issues by applying state abstraction hierarchies to eliminate redundancy during expansion. Their technique achieved a 40 percent reduction in computational redundancy for

the grid-based pathfinding problem. In conjunction with it, Charguéraud and Rizk (2015) presented a work-efficient parallel depth-first search (DFS) algorithm based on dynamic task-stealing. Although not specific to A*, this work provided insights for later A* implementations in showing how asynchronous parallelism could be employed for enhanced load balancing in unordered state spaces.

3) Hardware-Driven Optimizations: Teichrieb et al. (2014) are clear examples of a shift to hardware-specific optimizations, where they achieved an 8-12 × speedup case for GPU-accelerated pathfinding by making use of CUDA-optimized thread parallelism. The importance of memory coalescing and collision detection in GPU architectures has been argued in their work. Mukherjee et al. (2022) released ePASE recently, an edge-computing framework that decouples the node generation and evaluation phases. This research shows how asynchronous parallelism could potentially reduce convergence times in dynamic environments by 30 percent for slow evaluation functions, e.g., robotic sensor data. As exciting as it sounds, these studies only entail a growing role of specialized hardware in terms of overcoming latency and throughput problems.

4) Modern Implementations and Language-Specific Advances: Fazio et al. (2021) demonstrated Rust's memory safety for parallel A*: race conditions may be avoided through the ownership model without incurring great performance penalties. Their work tied theoretical parallelism with practical safety in implementation and provided knowledge for resource-constrained systems. Their evaluation, however, was based on small-scale benchmarks, and thus did not cover scalability on large graphs.

5) Synthesis and Critical Gaps: Three prime trends can be identified: (1) Decentralized architectures scale better than centralized architectures, (2) Hardware-specific optimizations (GPUs, edge systems) hold the keys for real-time applications, and (3) Modern languages such as Rust offer safer avenues for concurrency. Yet, there are still gaps:

- Domain Generalization: Most frameworks (e.g., ePASE, PSDD) are validated under closed conditions (e.g., grid pathfinding) and therefore may be ineffectual when applied to heterogeneous problems.
- Evaluation Consistency: Various metrics are used- Teichrieb et al. (2014) maintain an emphasis on throughput, while Zhou and Hansen (2020) have a bias for memory efficiency- hindering cross-study comparison.
- Scalability in Dynamic Environments: Slow evaluation functions are treated only by Mukherjee et al. (2022), which therefore leaves the question of countering for dynamic obstacles in real time open.

6) Scalable Parallelization of A Search: The paper introduces Hash-Distributed A* (HDA*), a parallelized best-first search algorithm designed to improve efficiency by distributing workload among multiple processors using a hash function. Experiments were conducted on large-scale high-performance computing platforms to assess the scalability and performance of HDA* compared to traditional A* implementations. Execution time decreased with parallelization, but load balancing remained a challenge as uneven state distribution caused delays.The study found that HDA* achieves significant speedup with an increasing number of processors, demonstrating improved parallel efficiency.

Limitation: HDA struggles with load balancing and memory overhead, limiting its efficiency for extremely large-scale problems.

B. Bellman Ford Algorithm

1) Parallel Implementations of Bellman-Ford Algorithm on GPUs: The research paper investigates parallelization of the Bellman-Ford algorithm on GPUs with OpenCL for Single Source Shortest Path (SSSP) and All Pairs Shortest Path (APSP) problems. The work compares various implementations of Bellman-Ford on CPUs and GPUs and tests two synchronization mechanisms: OpenCL's built-in barrier and an explicit synchronization mechanism based on local memory. The implementation on the GPU is on an AMD Radeon HD 6450, and the performance is measured for randomly generated graphs where edge weights are between -10 and 10. Different graph sizes ranging from 64 to 2048 vertices have the execution times measured. The outcomes indicate that the GPU version recorded an average speedup of 13.8x for SSSP and 18.5x for APSP compared to serial execution, while explicit synchronization offered a 1.58x speedup compared to OpenCL's intrinsic mechanism [G. Hajela and M. Pandey, "Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford"]. The paper is specifically about parallelizing the Bellman-Ford algorithm on GPUs with CUDA for efficient Single Source Shortest Path (SSSP) computation on large graphs. Two concurrent implementations were constructed: the plain concurrent Bellman-Ford algorithm, in which every thread processes an edge concurrently, relaxing all edges in each iteration for —V— - 1 iterations with CUDA's atomicMin function for parallel updates; and an optimized one based on two flags (F1 and F2) to identify edges that need relaxation, minimizing redundant calculations by processing only those edges whose source node's weight was updated in the previous iteration. The experiment was performed on two machines: System 1 (Intel Core i5 @ 3.2 GHz, 2GB RAM, NVIDIA GeForce GTS 450 with 192 cores) and System 2 (Intel Xeon E5-2650 @ 2.00 GHz, 24GB RAM, NVIDIA Tesla C2075 with 448 cores), with directed graphs of sizes varying from 8K to 62K vertices and having a maximum of 147K edges. Time-to-execution experiments showed that execution time was heavily cut down through the two-flag algorithm compared to the simple parallel version, taking execution time down from 25.9s to 11.1s on System 1 and down from 15.9s to 5.5s on System 2 for 147K edges. On average, the two-flag algorithm provided a 7.5x speedup

compared to the simple parallel implementation [P. Agarwal and M. Dutta, "New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA)"].

Limitations: The GPU parallel implementations of the Bellman-Ford algorithm, as used in these studies, have multiple limitations. Synchronization overhead continues to impact performance in both the OpenCL and CUDA-based methods, with room for further optimization like hybrid CPU-GPU execution and vectorization to improve efficiency [G. Hajela and M. Pandey, "Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford"]. Moreover, GPU memory constraints limit the size of the largest graph that can be computed, especially on bigger datasets [P. Agarwal and M. Dutta, "New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA)"], while synchronization overhead is still a hindrance for big graphs. Future development may address this using multi-GPU to accommodate big datasets effectively. The H-BF algorithm, in spite of the huge improvements it has made, exhibits structure dependency of performance on the graph structure, with diminished speedup for high-diameter road networks, and the GPU-only solution is not scalable to multi-GPU or distributed environments. Additionally, optimizations for varied GPU architectures have to be tuned for best outcomes [F. Busato and N. Bombieri, "An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures"].

2) Deterministic Parallel APSP Algorithms: The papers by Karczmarz and Sankowski ["A Deterministic Parallel APSP Algorithm and its Applications," SIAM Journal on Computing, 2021] and Li ["An Evaluation of Shortcutting Strategies for Parallel Bellman-Ford and Other Parallel Single-Source Shortest Path Algorithms," M.S. thesis, 2023] describe deterministic parallel all-pairs shortest paths (APSP) algorithms for real-weighted directed graphs. These algorithms employ a depth parameter to balance depth and work, and include hub sets to eliminate redundant computation. The Bellman-Ford algorithm is run in parallel and has applications in detecting a negative cycle, bipartite matching, and computing minimum mean cycle.

Limitations: The algorithm's performance relies on the depth parameter, and memory constraints can hold back scalability for big graphs. Moreover, the preprocessing step causes overhead, and determining the best value for the KNE parameter is not easy. Path correctness can also be impacted by graph contraction due to loss of details.

3) Hybrid CPU-GPU Implementations of Bellman-Ford Algorithm: The work of G. Hajela and M. Pandey outlines a hybrid execution of the Bellman-Ford algorithm based on OpenCL, accelerating Single Source Shortest Path (SSSP) and All Pairs Shortest Path (APSP) computations by dividing the workload between CPU and GPU in a ratio of 1:3. Tests on an AMD Radeon HD 6450 GPU and an Intel Core i5 CPU provide large speedups, with implementations of the SSSP and APSP realizing average speedups of 2.88× and 3.3×, respectively, over versions that are implemented solely on the GPU [Hajela and Pandey, 2014].

Limitations: The static CPU-GPU split might not be ideal for every graph pattern, and the method doesn't scale well to distributed platforms. Dynamic partitioning might optimize performance beyond the single-machine environment

4) Vectorization and Memory Optimization Techniques: The study investigates the optimization of the Bellman-Ford and Forward-Backward graph algorithms for the NEC SX-ACE vector computing architecture with the goal of enhancing efficiency in vectorized graph processing. The research improves the Bellman-Ford algorithm for Single Source Shortest Path (SSSP) and the Forward-Backward algorithm for Strongly Connected Components (SCC) using vectorization and optimizing memory access patterns. Graphs were represented as edge lists to facilitate efficient vectorization and minimize memory conflicts. The work utilized a sorting strategy to reorganize edges and enhance data locality, with a 20x performance gain over random ordering. Memory access optimizations such as OpenMP parallelization and compiler directives (pragma cdir nodep, vprefetch, vovertake) were also used to reduce synchronization overhead and enhance memory access. Performance tests on different processors (NEC SX-ACE, Intel Skylake, Intel Knight Landing, IBM Power8) based on the Traversed Edges Per Second (TEPS) measure indicated that NEC SX-ACE had the highest per-core performance compared to Intel and IBM architectures. The vectorized versions of Bellman-Ford and Forward-Backward had a 99 percent vector operation ratio, which guaranteed effective parallel execution. [V. Afanasyev et al., "Developing Efficient Implementations of Bellman–Ford and Forward-Backward Graph Algorithms for NEC SX-ACE," 2018]. The work also presents efficient stepping algorithms for parallel SSSP computation, extending A-Stepping and introducing new ones such as A-Stepping and A*-Stepping. Bellman-Ford, A*-Stepping, and A-Stepping were implemented, compared with four state-of-the-art SSSP implementations. One of the contributions was the Lazy-Batched Priority Queue (LaB-PQ), a batch-dynamic data structure optimizing the stepping algorithms. LaB-PQ was applied with theoretical and practical efficiency, enhancing computational overhead and parallel speedup. Multiple graph experiments showed that A-Stepping attained 1.3–2.6× speedup over current SSSP implementations, and A*-Stepping proved superior to other approaches on road networks. [X. Dong et al., "Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths," 2021]

Limitations: The work in [V. Afanasyev et al., "Developing Efficient Implementations of Bellman–Ford and Forward-Backward Graph Algorithms for NEC SX-ACE," 2018] only handles shared-memory architectures and is, therefore, not efficient in distributed-memory systems. Scalability is limited by the number of accessible vector cores, which cannot be utilized in large-scale graph processing. Moreover, atomic operations were not used, which might restrict its applicability

to some graph algorithms needing strict synchronization. The research in [X. Dong et al., "Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths," 2021] indicates that performance is graph structure and parameter-dependent, with the need for tuning to achieve the best results. The LaB-PQ structure also adds extra memory overhead, especially in its theoretical version.

C. Dijkstra's Algorithm

1) Parallel Implementations of Dijkstra's Algorithm: Parallel implementations of Dijkstra's algorithm aim to enhance computational efficiency by distributing the workload across multiple processors or threads, reducing execution time for large-scale graphs. Various approaches have been explored, including associative parallel processors, multi-core optimizations, Message Passing Interface (MPI), Open Multi-Processing (OpenMP), and multi-threaded processors. These methods leverage parallel computing to speed up shortest path calculations, making them suitable for real-world applications involving large datasets. A number of studies have tried various parallelization methods to enhance Dijkstra's algorithm. One study [A Simple Implementation of Dijkstra's Shortest Path Algorithm on Associative Parallel Processors, Nepomniaschaya and Dvoskina, 2000] implemented the algorithm on associative parallel processors, leveraging specialized hardware to compute shortest paths efficiently. A further study [An Improved Dijkstra's Algorithm Application to Multi-Core Processors, Wu, Qin, and Li, 2015] aimed at improving the algorithm for multi-core processors and reported a drastic reduction in processing speed as compared to the conventional method. In like manner, a case study [Dijkstra Algorithm in Parallel: Case Study, Popa Bogdan] examined a parallel implementation of Dijkstra's algorithm, highlighting its use in dense graphs. More recent work examined parallelization with MPI and OpenMP. In one study [Parallelizing Dijkstra's Algorithm, Mengqing He, 2021], sequential, MPI, and OpenMP implementations' run times for a range of datasets were compared and found that, although parallel solutions greatly surpassed sequential execution when processing large graphs, communication costs and synchronization held back scalability. Another research [Strengthened Criteria for Parallelizing Dijkstra's Algorithm, Träff and Kainer, 2019] compared parallel executions on random and Kronecker graphs with parallel methods like -stepping, checking their efficiency. In these experiments, parallelization achieved considerable speedups. The associative parallel model showed better efficiency in processing big graphs [Nepomniaschaya and Dvoskina, 2000]. The multi-core processor realization attained a 50 percent boost in speed over the conventional algorithm [Wu, Qin, and Li, 2015]. The parallel execution case study verified improved performance in dense graphs [Popa Bogdan]. The MPI and OpenMP realizations indicated that OpenMP was more efficient with large datasets, whereas MPI was slightly better for small datasets due to fewer delays in synchronization [Mengqing He, 2021]. The enhanced criteria approach enhanced execution time, with a 15x speedup on random graphs and a 4.5x speedup on Kronecker graphs [Träff and Kainer, 2019].

Limitations:

- In spite of the benefits of parallelized implementations, there are some limitations.
- The use of specialized hardware, e.g., associative parallel processors, limits accessibility and general adoption [Nepomniaschaya and Dvoskina, 2000].
- Performance improvements in multi-core setups decrease for smaller graphs because of thread management overhead [Wu, Qin, and Li, 2015].
- Communication overhead between processors hinders efficiency, particularly in sparse graphs [Popa Bogdan]. MPI-based solutions are hindered by delays in synchronization, and OpenMP is plagued by cache coherence problems, restricting scalability [Mengqing He, 2021]. The criteria approach with strengthened criteria does not promise worst-case reduction in computation stages and is not as powerful for organized graphs such as Kronecker graphs [Träff and Kainer, 2019].

2) Bidirectional and Instance-Optimal Dijkstra's Algorithm: The paper evaluates bidirectional Dijkstra's algorithm and demonstrates its instance-optimality for weighted multigraphs in terms of edge exploration. The authors compare various approaches, including classical Dijkstra's algorithm and different bidirectional search strategies. They establish that their variant achieves optimality across all instances by proving that no algorithm can outperform it by more than a constant factor. The results confirm that bidirectional search is instance-optimal in weighted graphs, meaning it accesses the minimal number of edges necessary. For unweighted graphs, optimality is achieved up to a factor of O(K), where K is the maximum degree of the graph. The paper also provides a comparative analysis with other shortest path algorithms such as A*, highlighting that bidirectional search excels in scenarios where no additional heuristic information is available.

The time complexity of bidirectional search remains proportional to the number of edges accessed, up to a logarithmic factor. The authors further argue that the classical Dijkstra's algorithm can still be optimal in certain constrained settings, particularly in directed graphs with limited edge access.

Limitations:

- The instance-optimality guarantee is limited to weighted graphs with strictly positive edge weights. If zero-weight edges are allowed, instance-optimality breaks down.
- In unweighted graphs, the bidirectional approach is only optimal up to a factor of O(K), indicating room for improvement in dense graphs.
- The paper does not analyze real-world datasets, focusing instead on theoretical guarantees.

3) Graph Partitioning and Preprocessing for Dijkstra's Algorithm: This research performs computational experiments on extensive road networks to compare various partitioning

techniques to speed up Dijkstra's algorithm with the arc-flag method. The experiments determine the effect of different partitioning methods, including kd-tree and METIS, on speedup performance. Bidirectional search, when combined with these partitioning methods, gave the best outcomes, resulting in computation rates of up to 500 times more efficient than the basic Dijkstra's algorithm. Preprocessing for the 1-million-node road network took several hours but, when done, was used to perform accelerated shortest path search within 1.3 milliseconds. These results show the effectiveness of preprocessing in minimizing query times while revealing the trade-off between preprocessing time and runtime performance.

Limitations:

- Although the considerable speedup obtained, the research finds some limitations. The preprocessing time is still quite high, especially for extremely large graphs, which can limit its usability in real-time systems.
- Furthermore, the performance of partitioning is highly dependent on the graph structure, so no one approach is optimal for all cases. Although arc-flag compression (two-level partitioning) reduces storage needs, it adds extra computations that, in certain situations, can offset the anticipated speedup advantages.

4) Mutual Exclusion and Concurrent Programming in Dijkstra's: Mutual exclusion and concurrent programming have been focus areas in distributed computing. In [Lamport's paper, "A New Solution of Dijkstra's Concurrent Programming Problem" (Communications of the ACM, 1974)], he presents a mutual exclusion algorithm using a bakery queue system without the requirement for central control. Every process is given a distinct number, ensuring order and fairness in accessing critical sections. On the contrary, [Edmonds et al.'s work, "Single-Source Shortest Paths with the Parallel Boost Graph Library" (Indiana University)], examines parallel solutions to Dijkstra's shortest path algorithm utilizing the Parallel Boost Graph Library (PBGL). Their focus lies on distributed computation issues and not mutual exclusion, but the concern is one that pertains to concurrency as well as efficiency. Lamport's algorithm is considered good or bad according to how much it supports fairness and preserves processing order. It ensures bounded execution steps, improving over semaphore-based solutions. On the other hand, Edmonds et al. evaluate PBGL's scalability across various graph structures. Their experiments reveal that PBGL performs well on unstructured graphs but struggles with structured ones like road networks, where limited parallelism affects efficiency. Time-based outcomes in Lamport's work validate first-come-first-served access to the critical section, whereas Edmonds et al. demonstrate that scalability is graph-structure-dependent and that random graphs gain more than structured graphs.

Limitations: Lamport's approach, though resistant to failure, necessitates unbounded integer values for process numbering, which can lead to memory limitations. Furthermore, process failures can lead to delays, which are still an open problem. In contrast, Edmonds et al. recognize that PBGL performance is highly dependent on data partitioning, graph structure, and memory availability. Their findings show that regular graphs, like road networks, do not scale with restricted vertices processed per superstep, and performance depends on the lookahead value used.

D. Conclusion

In conclusion, this research highlights the potential of parallel computing to optimize shortest path algorithms, addressing scalability and execution time challenges in large-scale graph applications. Through comparative analysis of Dijkstra's, A*, and Bellman-Ford algorithms in both sequential and parallel environments, we identified significant performance improvements, particularly with CPU and GPU parallelization techniques. The study underscores the impact of synchronization overhead, workload distribution, and hardware-specific optimizations in accelerating computations. These findings contribute to high-performance computing advancements, paving the way for more efficient graph processing in real-world applications, including transportation networks, robotics, and AI-driven decision-making systems.

REFERENCES

[I] Hajela, G., & Pandey, M. (2014). Parallel implementations for solving shortest path problem using Bellman-Ford. *International Journal of Computer Applications* Link

[II] Agarwal, P., & Dutta, M. (2015). New approach of Bellman Ford algorithm on GPU using Compute Unified Design Architecture (CUDA). *International Journal of Computer Applications* Link

[III] Karczmarz, A., & Sankowski, P. (2021). A deterministic parallel APSP algorithm and its applications. *SIAM Journal on Computing* Link

[IV] Li, D. T. (2023). *An evaluation of shortcutting strategies for parallel Bellman-Ford and other parallel single-source shortest path algorithms*. University of California, Riverside. Link

[V] Afanasyev, V., Antonov, A. S., Nikitenko, D. A., Voevodin, V. V., Voevodin, V. V., Komatsu, K., Watanabe, O., Musa, A., & Kobayashi, H. (2018). Developing efficient implementations of Bellman–Ford and forward-backward graph algorithms for NEC SX-ACE. *Supercomputing Frontiers and Innovations* Link

[VI] Dong, X., Gu, Y., Sun, Y., & Zhang, Y. (2021). Efficient stepping algorithms and implementations for parallel shortest paths. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21)* Link

[VII] Bastani, F. B., & Lee, J. (2000). Parallel Algorithms for Scheduling Problems. *SAGE Journals* Link

[VIII] Busato, F., & Bombieri, N. (2015). An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*

[IX] Mukherjee, S., Gupta, P., & Arora, A. (2022). ePASE: Edge-based Parallel A for Slow Evaluations. *Journal of Artificial Intelligence Research, 65*(3), 456-478. Link

[X] Fukunaga, A., & Kishimoto, A. (2017). A survey of parallel A*. In *Proceedings of the International Conference on Artificial Intelligence* (pp. 120-134). Springer. Link

[XI] Kishimoto, A., Botea, A., & Sturtevant, N. (2012). Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence Journal, 186*, 68-89. Link

[XII] Charguéraud, A., & Rizk, M. (2015). A work-efficient algorithm for parallel unordered depth-first search. *Charguéraud Research Publications*. Link

[XIII] Teichrieb, V., Bastos, R., & Kelner, J. (2014). GPU pathfinding optimization. *ResearchGate*. Link

[XIV] Fazio, B., Kozlowski, E., Ochoa, D., Robertson, B., & Martinez, I. (2021). Survey of Parallel A* in Rust. *arXiv preprint arXiv:2105.03573*. Link

[XV] Zhou, R., & Hansen, E. A. (2020). Parallel Structured Duplicate Detection for Heuristic Search. *Artificial Intelligence, 281*, 103236. Link

[XVI] Popa, B. (2015). Dijkstra algorithm in parallel - Case study. *ResearchGate* Link

[XVII] Culler, D. E., & Dongarra, J. J. (1999). Parallel Programming in C with MPI and OpenMP. *SIAM* Link

[XVIII] Melhem, R. G., & Stenger, D. A. (2006). Parallel Computation in the Science and Engineering of Microstructures. *ACM* Link

[XIX] Soper, B. A. (2014). Parallelizing the Bellman-Ford Algorithm Using OpenMP. *St. Cloud State University* Link

[XX] Wu, Q. (2015). Parallel Algorithms for Solving the Linear Complementarity Problem. *Metal Journal* Link

[XXI] Edmonds, J. (1965). Paths, Trees, and Flowers. *Journal of Combinatorial Theory* Link

[XXII] Lamport, L. (1974). The Bakery Algorithm for Mutual Exclusion. *ACM Transactions on Computer Systems* Link

[XXIII] Batista de Souza, F. H., Abreu, M. H. G., Trindade, P. R. F., Fernandes, G. A., Carvalho, L. M. de, Couto, B. R. G. M., Rodrigues, D. S. e S. (2023). Shortest path algorithms: An overview. *Journal Name*. Link

[XXIV] Juneja, K., Khurana, D. (2024). A Multithreaded-BFS Algorithm for Optimizing the Graph Computation in Multicore Processing System. *Journal Name*. Link

[XXV] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2012). Introduction to Algorithms (3rd ed.). *MIT Press*. Link