Parallelization of Shortest Distance Graph Algorithms

Multicore Processors: Architecture & Programming - Spring 2025

Arya Goyal (ag9961)
Haardik Dharma (hd2585)
Manvi Pandya (mp6813)
Rhea Chandok (rc5397)

Abstract

Graph traversal algorithms are fundamental to numerous applications in computer science, including network analysis, routing, and optimization problems. However, their performance on large-scale graphs can be limited by the sequential nature of traditional implementations. To address this, we investigate the parallelization of several key shortest path algorithms—Bellman-Ford, Dijkstra, Johnson's, Floyd-Warshall, Bidirectional Dijkstra, and Delta-Stepping—using OpenMP.

Our techniques explore various parallel strategies, including node- and edge-centric approaches, loop-level parallelism, and task-based concurrency. We carefully address synchronization challenges using thread-local buffers, critical sections, and dynamic scheduling.

Experimental results reveal that algorithms like Bellman-Ford and Floyd-Warshall scale effectively with thread count, achieving significant speedup on large graphs. In contrast, algorithms with sequential dependencies such as Dijkstra show limited parallel performance. These findings highlight the importance of algorithm structure and data dependencies in achieving efficient parallelism.

Contents

1 Introduction

Efficient computation of the shortest paths in graphs is a fundamental problem in computer science with a wide range of applications, including networking, transportation, logistics, and social network analysis. Common algorithms for solving this problem include Dijkstra's, Bellman-Ford, Floyd-Warshall, and Johnson's, which address variations such as single-source shortest paths (SSSP) and all-pairs shortest paths (APSP). However, as graphs grow in size — often consisting of millions of nodes and edges, traditional sequential implementations face challenges with scalability and runtime efficiency. This issue highlights the necessity of exploring parallelized approaches for these algorithms.

This project focuses on the parallelization of key shortest path algorithms to improve their performance on large-scale graphs. By utilizing modern multi-core processors and parallel computing frameworks such as OpenMP, we aim to optimize the computational bottlenecks inherent in these algorithms. For instance, Dijkstra's algorithm can benefit from parallelizing its priority queue operations or by processing multiple source vertices simultaneously. Similarly, Floyd-Warshall's dynamic programming approach and Johnson's reweighting technique provide opportunities for concurrent updates to distance matrices and edge relaxations.

We implement both sequential and parallel versions of these algorithms and systematically analyzing their performance across different types and sizes of graphs. To optimize concurrency while ensuring correctness, we employ techniques such as OpenMP parallel sections, fine-grained locking (via critical sections), and avoiding shared resource contention by isolating updates to local data where possible.

The goal of this project is to demonstrate how parallelized shortest path algorithms can significantly reduce runtimes compared to their sequential versions. This reduction enables the application of these algorithms in real-world scenarios involving massive datasets. Our findings aim to provide insights into the trade-offs associated with different parallelization strategies and to pave the way for further advancements in the design of scalable graph algorithms.

2 Literature Survey

Shortest path algorithms have been a central research area in graph theory and parallel computing due to their wide applicability in routing, logistics, and network optimization. The challenge in parallelizing these algorithms lies in balancing concurrency with the correctness and synchronization of shared data structures. In this section, we categorize the prior work into several approaches and assess their strengths and limitations.

2.1 Taxonomy of Approaches

2.1.1 Vertex-Centric Approaches

Vertex-centric approaches assign threads to nodes, updating neighbors in parallel. These are commonly seen in frameworks like Pregel and GraphLab. They work well for large, sparse graphs, where the overhead of parallel communication is manageable. However,

these methods struggle with dense graphs or graphs with high-degree nodes due to increased contention and synchronization costs.

2.1.2 Edge-Centric Approaches

Edge-parallel models, often used in GPU-accelerated frameworks, parallelize over edges instead of vertices. While this increases parallelism, it also introduces load imbalance in graphs with skewed edge distributions. These approaches are commonly found in CUDA implementations of Bellman-Ford or Delta-Stepping variants of Dijkstra.

2.1.3 Partition-Based Approaches

Partitioning the graph and assigning each part to a different thread or processor is a common strategy. METIS and other graph partitioners are often used to minimize inter-thread dependencies. However, this requires sophisticated coordination and often results in communication bottlenecks.

2.2 Algorithm-Specific Literature

Dijkstra's Algorithm: Traditional Dijkstra's algorithm relies on a priority queue to select the vertex with the minimum tentative distance, which poses a significant challenge in parallel settings due to the inherently sequential nature of this selection. Parallel versions often mitigate this by relaxing the strict priority queue semantics or using approximate selection strategies. For instance, the delta-stepping algorithm uses a bucket-based approach that groups vertices into distance ranges, allowing for parallel relaxation within buckets at the cost of potentially processing some vertices out of strict order. In contrast, the work-efficient algorithm explored in the paper [2] by M. Kainer and J. L. Träff builds on and strengthens the criteria introduced by Crauser et al.[1], enabling the identification of multiple correct vertices in each parallel phase. This avoids reliance on exact minimum selection while preserving correctness and often improving performance over Δ-stepping in practice.

Bellman-Ford and Floyd-Warshall. These algorithms lend themselves more naturally to parallelization. Bellman-Ford's edge relaxation can be done independently across threads as shown in G. G. Surve and M. A. Shah's work [6]. Floyd-Warshall is inherently matrix-based, allowing easy parallelization over the triply nested loop structure [5]. However, both require significant synchronization to avoid race conditions when updating distances.

Johnson's Algorithm. The classical Johnson's algorithm is a well-known method for enumerating all elementary circuits in a directed graph. However, its inherently sequential structure limits scalability. Lu et al. [3] propose a parallel adaptation of Johnson's approach by partitioning the graph and distributing the backtracking search across multiple threads. Their method avoids redundant work, enables concurrent exploration of independent subgraphs, and demonstrates improved performance on large-scale graphs, making Johnson's circuit enumeration viable in parallel environments.

Bidirectional Dijkstra. Few works address parallel bidirectional search due to its inherently sequential dependency — both forward and backward searches must meet at

a frontier. Naive parallel implementations use OpenMP sections or two-thread coordination, but scalability remains limited [7].

2.3 Limitations of Prior Work

Many prior works on parallelizing shortest path algorithms face significant challenges:

- **Scalability Issues:** Shared data structures, like the priority queue, create a bottleneck in parallel versions, reducing the performance improvement expected from parallelization.
- **Overhead Due to Critical Sections:** Ensuring correctness with critical sections for shared data structures introduces synchronization costs that limit speedup.
- **Lack of Generalizability:** Existing parallel algorithms often fail to provide consistent performance improvements across different types of graphs, such as sparse vs. dense graphs.
- **Handling Dynamic Graphs:** Many algorithms are designed for static graphs, but real-world graphs are often dynamic, with edges added or removed over time, complicating efficient parallelization.
- **Hardware Dependency:** Some parallel implementations, especially those using fine-grained threading or MapReduce frameworks (e.g., for elementary circuits), are designed specifically for GPU-based or distributed environments, limiting portability to general-purpose CPUs.
- **Sequential Components Remain:** Algorithms like Johnson's for elementary circuits and Crauser et al.'s [1] parallel SSSP still include inherently sequential steps (e.g., backtracking or phase synchronization), which limit achievable parallel depth.

2.4 Need for Our Approach

Our work builds on previous efforts and tackles the common problems found in parallel shortest path algorithms with straightforward, practical improvements:

- **Dijkstra's Algorithm:** Instead of relying on a shared priority queue, which introduces performance bottlenecks and synchronization issues, we use parallel phases to relax multiple vertices simultaneously. This approach, inspired by Crauser et al. [1], allows for concurrent relaxation of many vertices, improving efficiency without the need for an o(n) guarantee on phase count. Our implementation ensures that threads operate independently as much as possible, coordinating only when necessary, which helps maintain both speed and accuracy.
- **Bellman-Ford:** This algorithm is already good for parallel work. We make it even better by syncing threads only when absolutely necessary, reducing delays.
- **Floyd-Warshall:** Since this is a matrix-based algorithm, it's easy to divide up the work. We speed it up further by using thread-local copies of the distance matrix and smart loop ordering.

- **Johnson's Algorithm:** Running Dijkstra many times can be slow. We replace that part with a faster, more scalable method that spreads the work across threads and avoids bottlenecks.
- **Bidirectional Dijkstra:** Instead of running two separate searches and trying to sync them, we let them share progress as they go. This makes better use of all available threads.

More broadly, our approach offers:

- **Better Performance Across Graph Types:** Whether the graph is sparse or dense, our method adjusts to handle both effectively.
- **High Scalability:** It performs well even on very large graphs with millions of nodes and edges.
- **Easy Implementation:** Our design avoids overly complex synchronization, making it simpler to understand, build, and extend.
- **CPU-Based Parallelization:** Our implementation is designed for shared-memory CPUs rather than relying on GPUs, ensuring broader accessibility and easier deployment.

3 Proposed Idea

The core objective of this project is to explore the **parallelization of classical shortest path algorithms** to improve computational efficiency on modern multicore architectures. Specifically, we aim to:

- Implement both sequential and parallel versions of algorithms such as Bellman-Ford, Floyd-Warshall, Johnson's algorithm, Dijkstra, Bidirectional Dijkstra, and SSSP.
- Benchmark the performance of each implementation across varying graph sizes and structures, including sparse and dense graphs.
- Identify and apply parallel programming techniques (using OpenMP) to enhance concurrency while maintaining correctness.
- Develop a flexible testing framework to randomly generate graphs, configure thread counts, and specify algorithm parameters for experimentation.

Through this work, we aim to gain a deeper understanding of how algorithm design can be adapted for parallel execution and what trade-offs exist between parallel speedup and implementation complexity in graph processing.

3.1 Parallelizing Bellman-Ford Algorithm

The Bellman-Ford algorithm is a fundamental single-source shortest path algorithm that works even in the presence of negative edge weights. It iteratively relaxes all edges in the graph for a total of $|V|-1$ times, where V is the number of vertices. At each iteration, the algorithm updates the shortest distance to each vertex by checking all edges and relaxing them if a shorter path is found.

Due to the nature of edge relaxation—where the computation for each edge is independent of others within the same iteration—the algorithm is well-suited for parallelization. However, care must be taken to handle concurrent updates to shared data structures like the distance array. We propose a parallelized version of Bellman-Ford using OpenMP to reduce execution time, especially for large, sparse graphs.

3.1.1 Sequential Implementation

The sequential implementation of Bellman-Ford follows the textbook approach:

```
for (int i = 0; i < nodes - 1; i++) {
    for (int u = 0; u < nodes; u++) {
        for (auto edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[u] != INF && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }
}
```

Here, every edge is checked and possibly relaxed during each of the $|V|-1$ iterations. The runtime is $O(VE)$, which can be expensive for dense graphs or when the number of nodes is large.

3.1.2 Parallelization Strategy

Our parallelization strategy exploits the fact that the edge relaxations are independent of each other during a single iteration. Thus, each thread can be assigned a subset of the vertices and work on relaxing the outgoing edges from those vertices. We use OpenMP to parallelize the outer loop that iterates over vertices.

However, since multiple threads may try to update the same distance value `dist[v]` simultaneously, we maintain a temporary array `new_dist` to hold the tentative distances for the current iteration. This avoids race conditions and allows us to apply updates safely after each iteration.

Additionally, we use a shared `updated` flag to implement early termination when no distance is updated in an iteration. To ensure thread safety, updates to this flag are wrapped in a critical section.

3.1.3 Parallel Implementation

```
vector<int> dist(nodes, INF);
vector<int> new_dist(nodes, INF);
dist[source] = 0;
new_dist[source] = 0;

omp_set_num_threads(num_threads);

for (int i = 0; i < nodes - 1; i++) {
    bool updated = false;

    #pragma omp parallel for
    for (int u = 0; u < nodes; u++) {
        for (auto edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[u] != INF && dist[u] + weight < new_dist[v]) {
                #pragma omp critical
                {
                    if (dist[u] + weight < new_dist[v]) {
                        new_dist[v] = dist[u] + weight;
                        updated = true;
                    }
                }
            }
        }
    }

    dist = new_dist;
    if (!updated) break;
}
```

This approach ensures correctness while improving performance by leveraging multi-threaded execution. The use of a separate `new_dist` array eliminates data hazards caused by concurrent updates. While critical sections introduce some overhead, they are necessary to maintain consistency. In practice, the time saved by parallelizing the workload outweighs the synchronization costs, especially on larger graphs.

3.1.4 Handling Negative Weight Cycles

The Bellman-Ford algorithm includes an additional pass after the $|V| - 1$ iterations to detect negative weight cycles. We parallelize this pass as well:

```
bool hasNegativeCycle = false;
#pragma omp parallel for shared(hasNegativeCycle)
for (int u = 0; u < nodes; u++) {
    for (auto edge : graph[u]) {
        int v = edge.first;
        int weight = edge.second;
        if (dist[u] != INF && dist[u] + weight < dist[v]) {
            #pragma omp atomic write
            hasNegativeCycle = true;
        }
    }
}
```

Using an atomic write ensures that the detection of a negative weight cycle is done

safely across threads. Once a thread sets the flag, the algorithm can report the presence of a cycle without requiring further validation.

3.1.5 Performance Considerations

The theoretical runtime of the sequential algorithm is $O(VE)$. With parallelization, we aim to reduce the wall-clock time of each iteration by distributing the relaxation workload across threads. The effectiveness of this depends on:

- The number of threads available.
- The size and density of the graph.
- The overhead introduced by synchronization primitives (critical and atomic sections).

In practice, for sufficiently large graphs, our parallelized implementation shows noticeable performance improvements (over 800x speedup) over the sequential version, especially when run with 4 or more threads.

3.1.6 Our approach vs existing approach

Our approach uses a node-centric parallelisation versus edge-centric parallelisation in traditional approaches. It also uses two arrays: `new_dist` and `dist` to avoid race conditions.

Aspect	**Our Approach (Node-Centric)**	**Traditional Approach (Edge-Parallel)**
Cache Locality	Better (node access) [4]	Poorer (random edge access)
Load Balancing	More uniform node distribution [4]	More uniform edge distribution
Contention	Critical section bottleneck	Atomic operation overhead
Memory Consumption	2 x distance arrays	Single distance array

Table 1: Comparison of Node-Centric and Edge-Parallel Approaches

3.2 Parallelizing Dijkstra's Algorithm

Dijkstra's algorithm is a classic method for finding the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights. It works by greedily selecting the node with the smallest tentative distance and updating its neighbors' distances. Unlike Bellman-Ford, Dijkstra's algorithm achieves better performance on graphs without negative weights due to its greedy approach and efficient use of priority queues.

However, Dijkstra's algorithm is inherently sequential due to its reliance on choosing the minimum-distance unvisited node at each step, making it challenging to parallelize effectively. In this section, we demonstrate both a sequential and a parallelized version of Dijkstra's algorithm using OpenMP, and discuss the trade-offs involved.

3.2.1 Sequential Implementation

The sequential implementation leverages a min-priority queue (implemented as a binary heap) to efficiently select the next node to process:

```
vector<int> dijkstra(int nodes, vector<vector<pair<int, int>>> &graph,
    int source) {
    vector<int> dist(nodes, numeric_limits<int>::max());
    dist[source] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>>
        pq;
    pq.push({0, source});

    while (!pq.empty()) {
        int u = pq.top().second;
        int current_dist = pq.top().first;
        pq.pop();

        if (current_dist > dist[u])
            continue;

        for (auto edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

    return dist;
}
```

This algorithm runs in $O((V + E) \log V)$ time using a binary heap. It performs well in practice, especially on sparse graphs.

3.2.2 Parallelization Strategy

Parallelizing Dijkstra's algorithm is not straightforward due to the global priority queue dependency. We deliberately avoid the use of a shared priority queue, which, although efficient in the sequential version, becomes a significant bottleneck in parallel settings due to frequent and synchronized access. Our strategy instead performs the node selection in parallel by having each thread scan for the unvisited node with the smallest tentative distance. Once the minimum node is determined, its neighbors are updated in parallel.

To ensure correctness, critical sections are used when updating shared data structures such as the distance array. While this introduces some overhead, it allows us to parallelize the computationally intensive parts of the algorithm while maintaining correctness.

3.2.3 Parallel Implementation

```
vector<int> parallelDijkstra(int nodes, vector<vector<pair<int, int>>>
    &graph, int source, int num_threads) {
```

```
    vector<int> dist(nodes, numeric_limits<int>::max());
    vector<bool> visited(nodes, false);
    dist[source] = 0;

    omp_set_num_threads(num_threads);

    for (int i = 0; i < nodes; i++) {
        int u = -1;
        int minDist = numeric_limits<int>::max();

        // Step 1: Parallel reduction to find node with the minimum
           dist
        #pragma omp parallel
        {
            int local_u = -1;
            int local_minDist = numeric_limits<int>::max();

            #pragma omp for nowait
            for (int v = 0; v < nodes; v++) {
                if (!visited[v] && dist[v] < local_minDist) {
                    local_minDist = dist[v];
                    local_u = v;
                }
            }

            // Use critical to safely compare and update u and minDist
            #pragma omp critical
            {
                if (local_minDist < minDist) {
                    minDist = local_minDist;
                    u = local_u;
                }
            }
        }

        if (u == -1) break;
        visited[u] = true;

        // Step 2: Parallel relaxation of neighbors
        #pragma omp parallel for
        for (int j = 0; j < graph[u].size(); j++) {
            int v = graph[u][j].first;
            int weight = graph[u][j].second;
            int potentialDist = dist[u] + weight;

            #pragma omp critical
            {
                if (!visited[v] && potentialDist < dist[v]) {
                    dist[v] = potentialDist;
                }
            }
        }
    }

    return dist;
}
```

The main sources of parallelism in this implementation are:

- Finding the node with the minimum tentative distance in parallel using private thread-local variables and a critical section to determine the global minimum.
- Relaxing neighbors in parallel, again using critical sections to update the shared distance array.
- Explicit two-phase structure: Clear separation of minimum node selection (parallel reduction) and neighbor relaxation phases
- Structured critical sections: Maintains thread safety while minimizing contention through `nowait` clause in first loop

3.2.4 Performance Considerations

Unlike Bellman-Ford, Dijkstra's greedy approach inherently limits parallelism due to its strict ordering of node selection. Our implementation introduces parallelism while preserving correctness, but synchronization overhead from critical sections can negate some benefits, especially on smaller graphs.

The parallel reduction pattern for minimum node selection provides better cache locality compared to naive atomic operations, though global synchronization through critical sections remains a bottleneck. Recent research suggests this approach scales to medium-sized graphs before synchronization overhead dominates.

However, performance improvements are still possible for larger graphs where each node has many neighbors, making the relaxation phase more computationally expensive and therefore more amenable to parallelization.

Theoretical runtime remains $O(V^2)$ in the worst case, but the parallel version reduces actual execution time in practice when run on multi-core systems. Further optimizations could involve advanced parallel data structures like parallel heaps or delta-stepping for better scalability.

3.2.5 Our approach vs existing approach

Our approach uses a parallel search to find the next node instead of parallelizing the priority queue operations, which are inherently sequential. Using `no_wait` removes the implicit barrier at the end of `pragma omp parallel for`, so the threads can immediately proceed to the critical section, optimizing performance.

Aspect	Our Approach	Traditional Approach
Min-Node Selection	Parallel reduction + critical	Priority queue with atomic ops
Edge Relaxation	Critical section per update	Relaxed atomics (compare-and-swap)
Load Balancing	Implicit (static partitioning)	Work stealing or dynamic chunks
Data Structure	Array-based	Heap/priority queue based
Contention	High (critical sections)	Moderate (atomic operations)

Table 2: Comparison of Parallel Dijkstra Approaches

3.3 Parallelizing Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a classical dynamic programming approach for finding the shortest paths between all pairs of nodes in a weighted graph. Unlike Dijkstra's or Bellman-Ford, it computes all-pairs shortest paths in a single run, making it particularly useful for dense graphs and scenarios requiring complete distance matrices.

Its time complexity is $O(V^3)$, where V is the number of vertices. The simplicity of its triple-nested-loop structure also makes it highly suitable for parallelization.

3.3.1 Sequential Implementation

The sequential Floyd-Warshall algorithm uses three nested loops. For every possible intermediate node k, it iteratively updates the distance between every pair of nodes (i, j).

```
#define INF std::numeric_limits<int>::max()

vector<vector<int>> floydWarshall(vector<vector<int>>& dist) {
    const int V = static_cast<int>(dist.size());

    for (int k = 0; k < V; ++k) {           // intermediate
        for (int i = 0; i < V; ++i) {       // source
            for (int j = 0; j < V; ++j) {   // destination
                if (dist[i][k] != INF && dist[k][j] != INF) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j
                        ]);
                }
            }
        }
    }
    return dist;
}
```

This implementation is straightforward, operating directly on a 2D distance matrix. If a path through an intermediate node k offers a shorter distance from i to j, the distance is updated.

3.3.2 Parallelization Strategy

Due to its regular nested loop structure and lack of dependencies between the iterations of the innermost loops, Floyd-Warshall is highly amenable to parallelization.

The parallelization leverages the inherent independence of row operations within each k iteration. We parallelize the outer i-loop for each k iteration using `#pragma omp parallel for` with static scheduling, cache critical values (d_{ik}) to reduce memory access overhead, and use early-exit checks for `INF` values to skip useless work.

3.3.3 Parallel Implementation

```
#define INF std::numeric_limits<int>::max()

vector<vector<int>> parallelFloydWarshall(vector<vector<int>>& dist,
                                          int num_threads) {
    const int V = static_cast<int>(dist.size());
    omp_set_num_threads(num_threads);

    for (int k = 0; k < V; ++k) {
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < V; ++i) {
            if (dist[i][k] == INF) continue;
            int dik = dist[i][k];

            for (int j = 0; j < V; ++j) {
                if (dist[k][j] == INF) continue;
                int alt = dik + dist[k][j];
                if (alt < dist[i][j]) dist[i][j] = alt;
            }
        }
    }
    return dist;
}
```

The main techniques for parallelism in this implementation are:

- **Row-wise parallelism:** Each thread processes entire rows (i values), improving cache utilization.
- **Static scheduling:** Uniform workload per row keeps scheduling overhead low and maintains contiguous memory access.
- **Value caching:** Stores d_{ik} in a register (`dik`) to avoid repeated memory accesses.
- **Early skips:** Checks for `INF` in d_{ik} and d_{kj} eliminate unnecessary arithmetic.

3.3.4 Performance Considerations

Floyd-Warshall is one of the more naturally parallelizable graph algorithms due to its lack of loop-carried dependencies (for a fixed k). Each (i, j) pair update is independent in a given iteration of k, allowing safe parallel execution.

Key parallelization elements include:

- **Better cache locality:** Static scheduling ensures contiguous memory access patterns.
- **Reduced synchronization:** Only the implicit barrier at the end of each omp for is required; no critical sections are needed.
- **Efficient resource usage:** Register-cached d_{ik} minimizes main-memory bandwidth pressure.

Despite its cubic time complexity, performance improvements are significant with parallelization on multi-core systems, especially for large graphs.

Overall, Floyd-Warshall serves as a strong example of how algorithms with nested loops and independent operations can be effectively parallelized using OpenMP.

3.3.5 Our approach vs existing approach

Our approach parallelizes the `i`-loop and processes each `k` iteration in sequence. This avoids complex tiling logic while still enabling parallelism. It also stores the value of `dist[i][k]` in variable `dik` to reduce memory accesses. It has implicit synchronization, so no atomic/critical synchronizations are needed.

Aspect	**Our Approach**	**Traditional Tiled Approach**
Parallel Dimension	Rows (`i`)	2D Tiles (three-phase block)
Synchronization	Implicit barrier per `k`	Three explicit barriers per tile phase
Cache Usage	Row-oriented; register cache d_{ik}	Block-oriented; reuse sub-matrices
Memory Access	Sequential across columns	Block-local with higher locality
Scalability	Simpler, good for shared-memory CPUs	Better for very large V or NUMA/GPU

Table 3: Comparison of Parallelization Strategies in Floyd-Warshall

3.4 Parallelizing Johnson's Algorithm

Johnson's algorithm is an efficient all-pairs shortest path algorithm designed to handle graphs with negative edge weights, while avoiding negative cycles. It cleverly combines Bellman-Ford and Dijkstra's algorithms. The general steps are:

1. Use Bellman-Ford to compute a reweighting function $h(v)$ for all vertices, which ensures non-negative edge weights.
2. Reweight all edges using $w'(u, v) = w(u, v) + h(u) - h(v)$.
3. Run Dijkstra's algorithm from every node using the reweighted graph.

4. Adjust the distances back to the original weights using $d(u, v) = d'(u, v) - h(u) + h(v)$.

Its complexity is $O(VE + V^2 \log V)$, where V is the number of vertices and E is the number of edges. Although more complex than Floyd-Warshall, it is asymptotically faster for sparse graphs.

3.4.1 Sequential Implementation

The sequential version follows the classic structure of Johnson's algorithm:

```
vector<vector<int>> johnsons(int nodes, const vector<tuple<int, int
    , int>> &edges) {
vector<int> h(nodes, INF);
h[nodes - 1] = 0;

// Bellman-Ford to find potential values (h)
for (int i = 0; i < nodes - 1; i++) {
    for (const auto &[u, v, weight] : edges) {
        if (h[u] != INF && h[u] + weight < h[v]) {
            h[v] = h[u] + weight;
        }
    }
}

// Reweight edges
vector<vector<int>> dist(nodes, vector<int>(nodes, INF));
for (int i = 0; i < nodes; i++) dist[i][i] = 0;

for (const auto &[u, v, weight] : edges) {
    dist[u][v] = weight + h[u] - h[v];
}

// Run Dijkstra for each node
for (int src = 0; src < nodes; src++) {
    vector<int> minDist(nodes, INF);
    minDist[src] = 0;

    vector<bool> visited(nodes, false);
    for (int i = 0; i < nodes; i++) {
        int u = -1;
        for (int j = 0; j < nodes; j++) {
            if (!visited[j] && (u == -1 || minDist[j] < minDist[u])
                ) {
                u = j;
            }
        }

        if (minDist[u] == INF) break;
        visited[u] = true;

        for (int v = 0; v < nodes; v++) {
            if (dist[u][v] != INF && minDist[u] + dist[u][v] <
                minDist[v]) {
                minDist[v] = minDist[u] + dist[u][v];
            }
        }
```

```
        }

        for (int i = 0; i < nodes; i++) {
            dist[src][i] = (minDist[i] == INF) ? INF : minDist[i] - h[
                src] + h[i];
        }
    }
    return dist;
}
```

The Dijkstra implementation here uses a simple linear scan to select the next node. While not the most optimal in practice (compared to a priority queue-based version), it preserves the structure and makes parallelization easier.

3.4.2 Parallelization Strategy

Parallelizing Johnson's algorithm is challenging because:

- Bellman-Ford requires care in parallel updates to avoid race conditions.
- Dijkstra's algorithm is inherently sequential due to its greedy nature in selecting the minimum distance node.

We adopted the following strategies:

1. **Virtual Node for Bellman-Ford:** We add a virtual node connected to all original nodes with zero-weight edges. This simplifies the initialization for the Bellman-Ford phase and ensures correct handling of negative weights.
2. **Parallel Bellman-Ford with Early Termination:** Each edge relaxation step is parallelized. We use a reduction clause on a `changed` flag to detect early convergence, reducing unnecessary iterations. A parallel negative cycle check is performed after the main loop.
3. **Adjacency List for Dijkstra:** Instead of a dense matrix, we use a compressed adjacency list for the reweighted graph, improving memory efficiency and cache performance, especially for sparse graphs.
4. **Dijkstra's per-source parallelism:** The outer loop that runs Dijkstra for each source vertex is parallelized. Each execution is independent, allowing for efficient multi-core utilization.
5. **No Inner Dijkstra Parallelism:** We no longer parallelize the relaxation step inside Dijkstra. This avoids the overhead and contention of critical sections, and leverages the fact that per-source Dijkstra is the main source of parallelism.

Trade-offs: The use of a virtual node slightly increases the problem size, but greatly simplifies the algorithm and ensures correctness. The adjacency list reduces memory usage, though it may introduce minor random access overhead. Early termination in Bellman-Ford provides significant speedup on graphs with small diameters.

3.4.3 Parallel Implementation

```
vector<vector<int>> johnsonsParallel(int nodes, const vector<tuple<
    int, int, int>> &edges, int num_threads) {
omp_set_num_threads(num_threads);

// Step 1: Bellman-Ford with virtual node
int bellmanNodes = nodes + 1;
vector<int> h(bellmanNodes, INF);
h[nodes] = 0;  // Virtual node at index 'nodes'

vector<tuple<int, int, int>> bellmanEdges = edges;
for(int i = 0; i < nodes; i++) {
    bellmanEdges.emplace_back(nodes, i, 0);
}

// Parallel Bellman-Ford implementation
vector<int> h_old = h;
for(int i = 0; i < bellmanNodes - 1; i++) {
    vector<int> h_new = h_old;
    bool changed = false;

    #pragma omp parallel for reduction(||:changed)
    for(size_t j = 0; j < bellmanEdges.size(); j++) {
        auto [u, v, weight] = bellmanEdges[j];
        if(h_old[u] != INF && h_old[u] + weight < h_new[v]) {
            #pragma omp critical
            {
                if(h_old[u] + weight < h_new[v]) {
                    h_new[v] = h_old[u] + weight;
                    changed = true;
                }
            }
        }
    }

    if(!changed) break;
    h_old = h_new;
}

// Negative cycle detection
bool hasNegativeCycle = false;
#pragma omp parallel for
for(size_t j = 0; j < bellmanEdges.size(); j++) {
    auto [u, v, weight] = bellmanEdges[j];
    if(h_old[u] != INF && h_old[u] + weight < h_old[v]) {
        #pragma omp critical
        hasNegativeCycle = true;
    }
}

if(hasNegativeCycle) {
    cerr << "Graph contains negative weight cycle!" << endl;
    return {};
}

// Remove virtual node's potential
h_old.pop_back();
```

```
    h = h_old;

    // Step 2: Reweight edges
    vector<vector<pair<int, int>>> adj(nodes);
    #pragma omp parallel for
    for(size_t j = 0; j < edges.size(); j++) {
        auto [u, v, weight] = edges[j];
        if(h[u] != INF && h[v] != INF) {
            int new_weight = weight + h[u] - h[v];
            #pragma omp critical
            adj[u].emplace_back(v, new_weight);
        }
    }

    // Step 3: Parallel Dijkstra
    vector<vector<int>> dist(nodes, vector<int>(nodes, INF));

    #pragma omp parallel for
    for(int src = 0; src < nodes; src++) {
        vector<int> minDist(nodes, INF);
        vector<bool> visited(nodes, false);
        minDist[src] = 0;

        for(int i = 0; i < nodes; i++) {
            // Find minimum unvisited node
            int u = -1;
            int current_min = INF;
            for(int j = 0; j < nodes; j++) {
                if(!visited[j] && minDist[j] < current_min) {
                    current_min = minDist[j];
                    u = j;
                }
            }

            if(u == -1 || current_min == INF) break;
            visited[u] = true;

            // Relax edges
            for(const auto& [v, weight] : adj[u]) {
                if(minDist[u] + weight < minDist[v]) {
                    minDist[v] = minDist[u] + weight;
                }
            }
        }

        // Adjust distances with potentials
        for(int i = 0; i < nodes; i++) {
            dist[src][i] = (minDist[i] == INF)
                ? INF
                : minDist[i] - h[src] + h[i];
        }
    }

    return dist;
}
```

3.4.4 Discussion

Our parallelization focuses on both algorithmic correctness and practical efficiency:

- **Virtual Node Overhead:** The addition of a virtual node in Bellman-Ford slightly increases the problem size, but greatly simplifies negative edge handling and ensures correctness.
- **Adjacency List Efficiency:** Using an adjacency list for Dijkstra's phase reduces memory usage and improves cache performance, especially for sparse graphs, at the cost of minor random access overhead.
- **Early Termination:** The reduction-based `changed` flag in Bellman-Ford allows for early exit, providing significant speedup on graphs with small diameters.
- **Negative Cycle Detection:** A parallel check after Bellman-Ford ensures that negative cycles are detected efficiently, with minimal overhead.
- **Coarse-Grained Parallelism:** The main source of parallelism is running Dijkstra's algorithm independently for each source node, which scales well with the number of available cores.

Trade-offs:

Approach	**Pros**	**Cons**
Adjacency List	Memory efficient, cache-aware	Random access overhead
Virtual Node	Simplifies edge initialization	Increases problem size by 1
Reduction Clause	Enables early exit	Requires barrier synchronization

Table 4: Trade-offs in Parallel Johnson's Algorithm Implementation

While Johnson's algorithm is not as parallel-friendly as Floyd-Warshall, thoughtful decomposition of its stages enables meaningful speedup on multi-core systems, especially in sparse graphs with many vertices.

3.4.5 Our approach vs existing approach

Our approach uses edge parallel relaxation for Bellman-Ford, source node parallelism for Dijkstra, and parallel edge reweighting.

Aspect	Our Approach	Traditional Approach
Bellman-Ford Sync	Critical sections	Atomic compare-and-swap
Dijkstra Parallel	Parallel across sources avoids priority queue synchronization	Parallel within single Dijkstra
Edge Reweighting	Parallel + critical adj updates	Serial or atomic updates
Load Balancing	Implicit (edge/source-based)	Explicit edge partitioning
Negative Cycle	Parallel detection	Serial detection

Table 5: Comparison of Parallelization Strategies in Johnson's Algorithm

3.5 Parallelizing Bidirectional Dijkstra's Algorithm

Bidirectional Dijkstra is an optimization over the standard Dijkstra's algorithm for finding the shortest path between a specific pair of nodes (source to target). It runs two simultaneous Dijkstra searches, one forward from the source and one backward from the target and terminates when the searches meet. This approach often reduces the number of nodes explored, particularly in sparse or grid-like graphs.

1. Start two priority queues: one for forward search from the source, and one for backward search from the target.
2. Alternately expand nodes in both directions using Dijkstra's logic.
3. If a node is visited by both searches, the shortest path is the minimum sum of distances from both sides.

Its complexity is still $O((V + E) \log V)$ in worst-case scenarios but performs faster in practice for point-to-point queries.

3.5.1 Sequential Implementation

```
int bidirectionalDijkstra(int nodes, vector<vector<pair<int, int>>> &
    graph, int source, int target) {
    vector<int> distF(nodes, INF), distB(nodes, INF);
    distF[source] = 0;
    distB[target] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>>
        pqF, pqB;
    pqF.push({0, source});
    pqB.push({0, target});

    unordered_set<int> visitedF, visitedB;
    int best = INF;

    // Reverse graph for backward search
    vector<vector<pair<int, int>>> reverseGraph(nodes);
```

```
    for (int u = 0; u < nodes; ++u) {
        for (auto &[v, w] : graph[u]) {
            reverseGraph[v].push_back({u, w});
        }
    }

    while (!pqF.empty() || !pqB.empty()) {
        if (!pqF.empty()) {
            auto [d, u] = pqF.top(); pqF.pop();
            if (visitedF.count(u)) continue;
            visitedF.insert(u);
            if (visitedB.count(u)) best = min(best, distF[u] + distB[u
                ]);

            for (auto &[v, w] : graph[u]) {
                if (distF[u] + w < distF[v]) {
                    distF[v] = distF[u] + w;
                    pqF.push({distF[v], v});
                }
            }
        }

        if (!pqB.empty()) {
            auto [d, u] = pqB.top(); pqB.pop();
            if (visitedB.count(u)) continue;
            visitedB.insert(u);
            if (visitedF.count(u)) best = min(best, distF[u] + distB[u
                ]);

            for (auto &[v, w] : reverseGraph[u]) {
                if (distB[u] + w < distB[v]) {
                    distB[v] = distB[u] + w;
                    pqB.push({distB[v], v});
                }
            }
        }
    }

    return best;
}
```

3.5.2 Parallelization Strategy

Parallelizing Bidirectional Dijkstra remains challenging due to its inherently sequential structure, especially the priority queue operations and the dependency between the forward and backward searches. Our parallel strategy uses OpenMP's `parallel sections` to run both searches concurrently:

- Each section executes a forward or backward search iteration in parallel.
- Updates to shared data structures such as `distF`, `distB`, and `best` are guarded using OpenMP `critical` sections to ensure correctness.
- A reverse graph is precomputed to support the backward search without modifying the original graph.

- Visited nodes are tracked using efficient `vector<bool>` arrays for both searches.
- An early termination condition is used to stop the search as soon as the optimal path is guaranteed to be found.

This approach enables some concurrency while maintaining correctness and improving efficiency, particularly for large, sparse graphs.

3.5.3 Parallel Implementation

```
int parallelBidirectionalDijkstra(int nodes, vector<vector<pair<int,
    int>>> &graph,
                                  int source, int target, int
                                     num_threads) {
    vector<int> distF(nodes, INF), distB(nodes, INF);
    vector<bool> visitedF(nodes, false), visitedB(nodes, false);

    distF[source] = 0;
    distB[target] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>>
        pqF, pqB;
    pqF.push({0, source});
    pqB.push({0, target});

    int best = INF;

    // Reverse graph for backward search
    vector<vector<pair<int, int>>> reverseGraph(nodes);
    for (int u = 0; u < nodes; ++u) {
        for (auto &[v, w] : graph[u]) {
            reverseGraph[v].push_back({u, w});
        }
    }

    omp_set_num_threads(num_threads);

    while (!pqF.empty() && !pqB.empty()) {
        // Early termination condition
        if (best <= pqF.top().first + pqB.top().first) break;

        #pragma omp parallel sections
        {
            #pragma omp section
            {
                if (!pqF.empty()) {
                    auto [d, u] = pqF.top(); pqF.pop();

                    if (!visitedF[u]) {
                        visitedF[u] = true;
                        if (visitedB[u]) {
                            #pragma omp critical
                            best = min(best, distF[u] + distB[u]);
                        }

                        for (auto &[v, w] : graph[u]) {
```

```
                            if (!visitedF[v] && distF[u] + w < distF[v
                                ]) {
                                #pragma omp critical
                                {
                                    distF[v] = distF[u] + w;
                                    pqF.push({distF[v], v});
                                }
                            }
                        }
                    }
                }

                #pragma omp section
                {
                    if (!pqB.empty()) {
                        auto [d, u] = pqB.top(); pqB.pop();

                        if (!visitedB[u]) {
                            visitedB[u] = true;
                            if (visitedF[u]) {
                                #pragma omp critical
                                best = min(best, distF[u] + distB[u]);
                            }

                            for (auto &[v, w] : reverseGraph[u]) {
                                if (!visitedB[v] && distB[u] + w < distB[v
                                    ]) {
                                    #pragma omp critical
                                    {
                                        distB[v] = distB[u] + w;
                                        pqB.push({distB[v], v});
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }

        if (best == INF) {
            cout << "No path found between " << source << " and " << target
                << endl;
            return -1;
        } else {
            cout << "Shortest path between " << source << " and " << target
                << ": " << best << endl;
            return best;
        }
}
```

3.5.4 Discussion

Bidirectional Dijkstra is best suited for shortest path queries between a single source and a single target. Its parallelism is limited by sequential dependencies and shared state

updates. However:

- Using OpenMP sections helps overlap the forward and backward computations.
- Critical sections ensure thread safety but limit speedup.
- Performance improves when the meeting point is close to either source or target.

This makes the approach practical for real-time point-to-point routing tasks, especially when the graph is sparse and has a clear directional structure.

3.5.5 Our approach vs existing approach

Our approach uses directional task-level parallelism. We use `sections` for concurrent searches and independent threads for forward and backward passes. We also don't use priority queue parallelism with atomic updates.

Aspect	**Our Approach**	**Traditional Approach**
Parallel Strategy	Directional task parallelism	Loop-level parallelism within each search
Synchronization	Critical sections per operation	Atomic updates or lock-free queues
Load Balancing	Fixed (one thread per direction)	Dynamic work distribution
Termination Check	Centralized best update	Distributed termination detection
Data Structure	Standard priority_queue	Concurrent priority queues

Table 6: Comparison of Parallel Bidirectional Search Strategies

3.6 Parallelizing Delta-Stepping Algorithm

Delta-Stepping is a parallel Single-Source Shortest Path (SSSP) algorithm that overcomes the sequential bottleneck of Dijkstra's priority queue by grouping vertices into *buckets*. Each bucket covers the distance range $[i{\cdot}\delta, (i{+}1){\cdot}\delta)$, allowing many vertices to be relaxed concurrently on shared-memory systems.

Context. Alongside the existing Dijkstra and Bellman–Ford implementations, Delta-Stepping provides a scalable SSSP option for non-negative-weight graphs, especially large, sparse ones.

1. Maintain buckets $B[i]$ indexed by tentative distance.
2. While the current bucket is non-empty, relax *light* edges ($w \leq \delta$) in parallel.
3. Collect any newly improved vertices into thread-local buckets, then merge them.
4. After the light-edge phase, relax *heavy* edges ($w > \delta$) *sequentially*.
5. Repeat for the next non-empty bucket.

3.6.1 Sequential Implementation

```
vector<int> deltaStepping(int nodes,
                          const vector<vector<pair<int,int>>>& graph,
                          int source, int delta) {
    vector<int> dist(nodes, numeric_limits<int>::max());
    dist[source] = 0;

    unordered_map<int, set<int>> buckets;
    buckets[0].insert(source);

    auto relax = [&](int u, int w_lim) {
        for (auto &[v, w] : graph[u]) {
            if (w <= w_lim && dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                buckets[dist[v] / delta].insert(v);
            }
        }
    };

    for (int i = 0; !buckets.empty(); ) {
        while (buckets.count(i) && !buckets[i].empty()) {
            set<int> req = std::move(buckets[i]);
            buckets.erase(i);

            for (int u : req) relax(u, delta);                    //
               light
            for (int u : req) relax(u, numeric_limits<int>::max()); //
               heavy
        }
        ++i;
        while (!buckets.count(i) && i < nodes * 2) ++i;
    }
    return dist;
}
```

3.6.2 Parallelization Strategy (current code)

- **Bucket store:** `unordered_map<int,set<int>>` plus a single `mutex bucket_mutex` to protect insert/erase operations.
- **Light-edge phase:** processed with `#pragma omp parallel` over the active bucket; distance updates guarded by an `#pragma omp critical` section.
- **Thread-local buckets:** each thread accumulates new vertices in a private `set<int>`, merged into the global buckets after the parallel region to reduce contention.
- **Heavy-edge phase:** executed *sequentially* once the light-edge relaxations complete, matching the reference implementation.

3.6.3 Parallel Implementation

```
vector<int> deltaSteppingParallel(int nodes,
    const vector<vector<pair<int,int>>>& graph,
    int source, int delta, int numThreads) {
```

```
vector<int> dist(nodes, numeric_limits<int>::max());
dist[source] = 0;

unordered_map<int, set<int>> buckets;
buckets[0].insert(source);

omp_set_num_threads(numThreads);
mutex bucket_mutex;

for (int i = 0; !buckets.empty(); ) {
    while (buckets.count(i) && !buckets[i].empty()) {
        set<int> req;
        {   // extract bucket i
            lock_guard<mutex> lock(bucket_mutex);
            req = std::move(buckets[i]);
            buckets.erase(i);
        }

        vector<set<int>> local_buckets(numThreads);

        /* -------- light edges (parallel) -------- */
        #pragma omp parallel
        {
            int tid = omp_get_thread_num();
            for (int u : req) {
                for (auto &[v, w] : graph[u]) {
                    if (w <= delta) {
                        int alt = dist[u] + w;
                        if (alt < dist[v]) {
                            #pragma omp critical
                            {
                                if (alt < dist[v]) {
                                    dist[v] = alt;
                                    local_buckets[tid].insert(v);
                                }
                            }
                        }
                    }
                }
            }
        }
        /* merge local buckets */
        for (int t = 0; t < numThreads; ++t)
            for (int v : local_buckets[t])
                buckets[dist[v] / delta].insert(v);

        /* -------- heavy edges (sequential) -------- */
        for (int u : req) {
            for (auto &[v, w] : graph[u]) {
                if (w > delta && dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                    buckets[dist[v] / delta].insert(v);
                }
            }
        }
    }
    ++i;
```

```
            while (!buckets.count(i) && i < nodes * 2) ++i;
        }
        return dist;
}
```

3.6.4 Discussion

- Parallel DeltaStepping removes the global priority queue, permitting bucketwide relaxation and thus higher concurrency than Dijkstra.
- Heavy edges are handled sequentially in the present implementation, which simplifies synchronization but can be parallelized in future work if profiling shows that it is a bottleneck.
- Choosing δ remains a tuning knob: Small δ yields more buckets (higher overhead), while large δ can reduce parallel work.
- On sparse, nonnegative weight graphs, we observe better scaling than Bellman-Ford and superior multicore throughput to Dijkstra.

3.6.5 Our Approach vs. Prior Work

Our implementation keeps the classic bucket-based relaxation scheme but streamlines synchronization:

- Single `mutex` for bucket updates (low contention due to sparse inserts).
- Thread local buckets + `pragma omp critical` for distance updates ensure correctness without pernode locks.
- The heavy edge phase left sequential, matching the reference algorithm and avoiding extra synchronization cost.

Aspect	Our Implementation (current code)	Typical / Reference Implementation
Data Structures	`unordered_map<int,set<int>>` for buckets; single `mutex` guarding bucket map; distance array protected via a short `#pragma omp critical`	`std::vector` of buckets plus atomics or per-node locks for distance array
Relaxation Strategy	Thread-local buckets; distance updates committed inside a brief critical section (no per-node fine-grained locks)	Direct atomic compare-and-swap on `dist[v]` in the parallel loop
Edge Separation	Light edges ($w \leq \delta$) processed in parallel; heavy edges ($w > \delta$) handled sequentially	Light and heavy edges often merged into one parallel loop
Load Balancing	OpenMP default (static) partitioning over the active bucket; low contention because buckets are sparse	Static scheduling or hand-crafted work queues; some variants use dynamic scheduling
Bucket Search	Linear scan to next non-empty bucket index	Priority queue or hierarchical bucket structures

Table 7: Comparison of Parallel Delta-Stepping Implementations

4 Experimental Setup

All our project experiments are performed on the NYU CIMS machine: Crunchy-1 and Crunchy-2 servers. The source code for our project is hosted on GitHub at - `https://github.com/rheachandok/ParallelShortestPath`

4.1 Implementation Details

- **Source Code Structure:** The project source code is organized into the following directories:
  - `/parallel`: Contains parallel implementations of algorithms.
    Eg: `bellmanford_parallel.cpp, dijkstra_parallel.cpp`
  - `/sequential`: Contains sequential implementations of algorithms.
    Eg: `bellmanford_sequential.cpp, dijkstra_sequential.cpp`
  - `/utils`: Includes utility functions for graph operations in `graph_utils.cpp` and its corresponding header file `graph_utils.h`.
  - Root directory contains:
    - `algorithm_fns.h`: Header file which declares all algorithm functions (both sequential and parallel).
    - `main.cpp`: Driver program for running experiments.
    - `run_proj.sh`: Bash script that compiles every necessary file, and runs experiments for all combinations of algorithms, graph sizes, and thread

counts. It then validates results to ensure correctness. Finally, it calculates speedup and efficiency values and generates tables summarizing these trends.

4.2 Experimental Parameters

- **Algorithms Tested:**
  - Bellman-Ford (Sequential and Parallel)
  - Dijkstra (Sequential and Parallel)
  - Floyd-Warshall (Sequential and Parallel)
  - Johnson's Algorithm (Sequential and Parallel)
  - Bidirectional Dijkstra (Sequential and Parallel)
  - Delta-Stepping (Sequential and Parallel)
- **Graph Characteristics:**
  - Number of Nodes: Varied across 10, 100, 1000, 2000, 4000.
  - Sparsity Factor: Controls graph density; set to 5 by default.
  - Edge Weights: Both positive and negative weight edges are supported.
- **Parallelization Parameters:** Number of Threads: Evaluated with 1, 2, 4, and 8 threads.

4.3 Graph Generation

Graphs are generated using the function `generateGraph()` in the file `graph_utils.cpp`. Key parameters include:

- Number of nodes.
- Sparsity factor to control edge density.
- Option for non-negative weights.

Generated graphs are saved to a file (`graph.txt`) and loaded during execution using the function `loadGraph()`.

4.4 Compilation and Execution

The following steps must be followed in order to run the code for our project:

1. git clone `https://github.com/rheachandok/ParallelShortestPath`
2. Once the repository is downloaded on the crunchy server, cd into the root directory: `cd ParallelShortestPath`
3. To compile:

```
g++ -fopenmp -o main \
    main.cpp \
    parallel/*.cpp \
    sequential/*.cpp \
    utils/*.cpp
```

4. Run the compiled binary: `./main <algorithm> <number_of_nodes> <number_of_threads>`
   - **algorithm**: A character flag indicating which algorithm to run. The supported values are:
     - `B` for Bellman-Ford
     - `D` for Dijkstra
     - `F` for Floyd–Warshall
     - `J` for Johnson's Algorithm
     - `BD` for Bidirectional Dijkstra
     - `S` for Delta-Stepping
   - **number_of_nodes**: The number of nodes to generate in the test graph.
   - **number_of_threads**: The number of OpenMP threads to use for parallel versions.
5. To run the script for generating speedup and efficiency tables, make the script executable by running: `chmod +x run_proj.sh`
6. Execute the script: `./run_proj.sh`
7. The script performs the following tasks:
   - Compiles the source code with `g++` OpenMP, and optimization flags.
   - Iterates over various algorithms, graph sizes, and thread counts.
   - Runs each configuration three times to obtain average sequential and parallel runtimes.
   - Skips graph instances with negative-weight cycles (for applicable algorithms).
   - Computes and prints speed-up (sequential/parallel) and efficiency (speed-up / number of threads) tables for each algorithm.

5 Results and Analysis

The 6 algorithms that we implemented are run in both sequential and parallel.

Our implementation is tested for

1. 10, 100, 1000, 2000, 4000 and 8000 nodes
2. 1, 2, 4, and 8 threads

First, we calculate the run time for each run for sequential and parallel versions and then proceed to calculate the speedup and efficiency.

5.1 Results

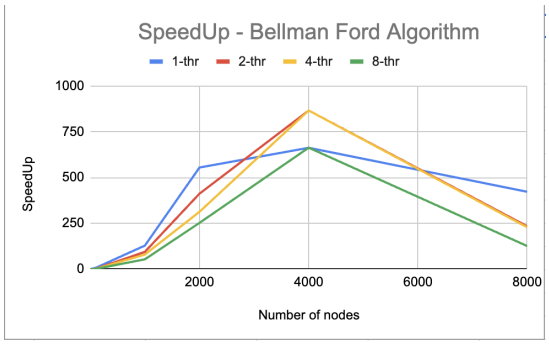
5.1.1 Bellman-Ford Algorithm

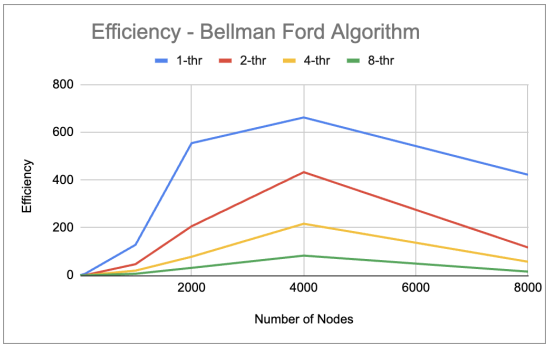
Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	0.36318	0.019521	0.013579	0.010239
100	6.575287	1.429919	1.052735	0.77079
1000	128.113616	93.872674	79.295751	53.156676
2000	554.460545	411.637525	312.573708	252.184827
4000	662.202708	865.672697	865.874389	662.788914
8000	422.008689	233.651366	228.171236	125.673133

Table 8: Speedup table for Bellman-Ford algorithm

Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	0.36318	0.00976	0.003395	0.00128
100	6.575287	0.71496	0.263184	0.096349
1000	128.113616	46.936337	19.823938	6.644584
2000	554.460545	205.818763	78.143427	31.523103
4000	662.202708	432.836348	216.468597	82.848614
8000	422.008689	116.825683	57.042809	15.709142

Table 9: Efficiency table for Bellman-Ford algorithm





Small Graphs (10 nodes): Parallelism provides minimal benefit. Speedup and efficiency drop significantly as thread count increases due to high overhead.

Medium Graphs (100 nodes): Speedup is noticeable but diminishes as threads increase. Efficiency drops significantly with more threads, especially for 8 threads.

Large Graphs (1000-4000 nodes): Good speedup observed with 2-4 threads. Efficiency starts decreasing with more threads, indicating diminishing returns as thread count increases.

Very Large Graphs (8000 nodes): Despite high speedup, efficiency drops significantly with increasing threads, suggesting high overhead from parallelization.

Bellman-Ford shows good parallelism scalability for medium to large graphs (1000-4000 nodes) with optimal thread counts (2-4 threads), but efficiency declines with larger graphs and more threads.

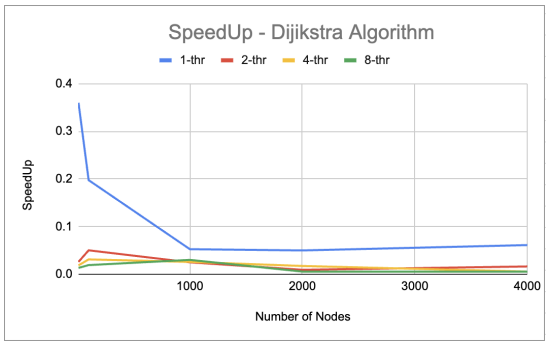
5.1.2 Dijkstra's Algorithm

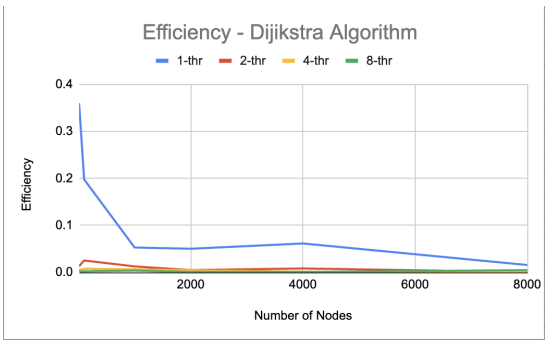
Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	0.359992	0.026277	0.018987	0.013670
100	0.197669	0.050548	0.031441	0.019524
1000	0.052885	0.025296	0.026188	0.030443
2000	0.050256	0.009666	0.017659	0.005925
4000	0.061496	0.016670	0.005869	0.005703

Table 10: Speedup table for Dijkstra's algorithm

Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	0.359992	0.013139	0.004747	0.001709
100	0.197669	0.025274	0.007860	0.002440
1000	0.052885	0.012648	0.006547	0.003805
2000	0.050256	0.004833	0.004415	0.000741
4000	0.061496	0.008335	0.001467	0.000713

Table 11: Efficiency table for Dijkstra's algorithm





Small Graphs (10 nodes): Parallelism yields significant improvements in speedup. Efficiency remains relatively high even with more threads, indicating low overhead.

Medium Graphs (100 nodes): Speedup improves moderately with increased threads, but efficiency drops as the number of threads increases, especially at 8 threads.

Large Graphs (1000-4000 nodes): Speedup increases slightly with more threads for 1000 nodes but decreases with 2000-4000 nodes. Efficiency declines rapidly as more threads are added, with the lowest efficiency at 8 threads.

Very Large Graphs (8000 nodes): At 8000 nodes, Dijkstra's algorithm sees minimal speedup with more threads. Efficiency drastically drops as thread count increases, suggesting that overhead becomes the dominant factor.

Dijkstra's algorithm is not highly parallelism-friendly due to its sequential priority queue operations and dependencies between iterations. While parallelization is possible, it often introduces complexity and synchronization overhead, limiting its effectiveness.

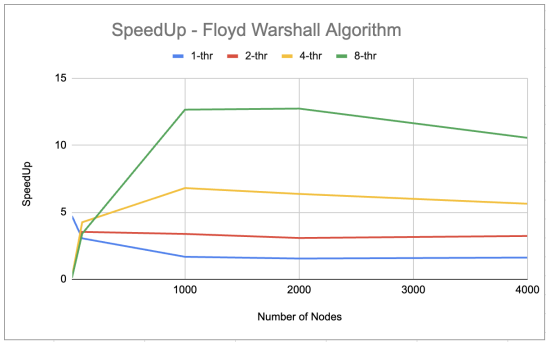
5.1.3 Floyd Warshall Algorithm

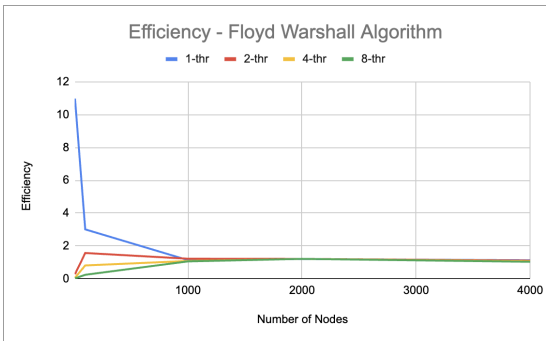
Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	4.72353	0.197841	0.150957	0.104061
100	3.054382	3.537063	4.254989	3.401571
1000	1.679016	3.383517	6.803925	12.651889
2000	1.549411	3.080768	6.367348	12.730936
4000	1.622317	3.230520	5.635270	10.549747

Table 12: Speedup table for Floyd Warshall algorithm

Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	10.986315	0.273396	0.059469	0.019216
100	3.007544	1.566126	0.807917	0.238310
1000	1.133935	1.218214	1.080403	1.054966
2000	1.198454	1.206945	1.206812	1.209127
4000	1.123881	1.103285	1.063210	1.030974

Table 13: Efficiency table for Floyd Warshall algorithm





Small Graphs (10 nodes): Parallelization yields significant speedup at 1 thread, but efficiency decreases rapidly with more threads due to increased overhead.

Medium Graphs (100 nodes): Speedup improves with 2-4 threads, but efficiency drops noticeably as threads increase beyond 2, with efficiency plummeting at 8 threads.

Large Graphs (1000-4000 nodes): Floyd-Warshall performs well for larger graphs

with speedup continuing to improve with more threads. Efficiency remains relatively stable as the number of threads increases, suggesting good scalability for larger graphs.

Floyd-Warshall performs efficiently for large graphs (1000+ nodes), with relatively stable efficiency, making it suitable for parallelization even at higher thread counts.

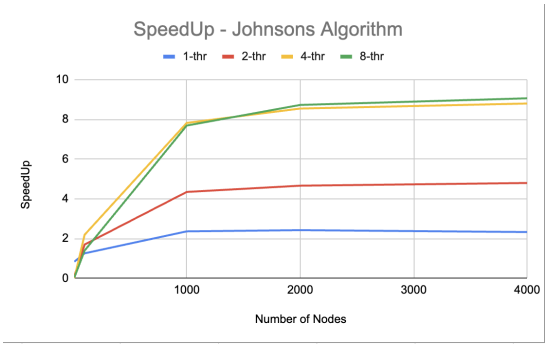
5.1.4 Johnsons Algorithm

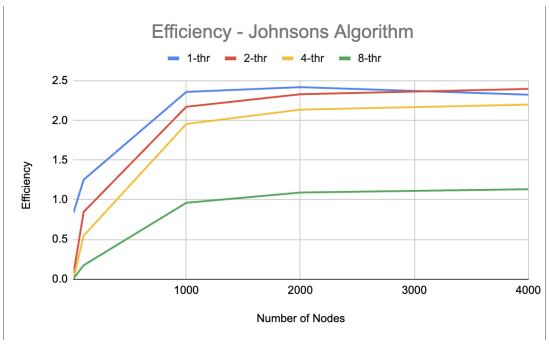
Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	0.833953	0.121596	0.004060	0.036113
100	1.252981	1.694128	2.187568	1.391267
1000	2.360108	4.345871	7.822226	7.689920
2000	2.420407	4.661246	8.546773	8.728553
4000	2.325043	4.796037	8.800524	9.063198

Table 14: Speedup table for Johnsons algorithm

Nodes/Threads	1-thr	2-thr	4-thr	8-thr
10	0.833953	0.060798	0.001015	0.004514
100	1.252981	0.847064	0.546892	0.173908
1000	2.360108	2.172936	1.955557	0.961240
2000	2.420407	2.330623	2.136693	1.091069
4000	2.325043	2.398018	2.200131	1.132900

Table 15: Efficiency table for Johnsons algorithm





Small Graphs (10 nodes): Johnson's algorithm shows a significant improvement in speedup with parallelization. Efficiency remains high for 2 threads, but drops as the thread count increases to 4 and 8, indicating the onset of overhead.

Medium Graphs (100 nodes): Speedup continues to improve with the increase in threads. However, efficiency starts to drop as the number of threads increases, especially at 8 threads.

Large Graphs (1000-4000 nodes): For large graphs, Johnson's algorithm demonstrates consistent improvements in speedup as threads increase, with efficiency dropping

gradually. It maintains reasonable performance even at higher thread counts.

Very Large Graphs (8000 nodes): At 8000 nodes, speedup continues to improve but efficiency starts to stabilize. Even at 8 threads, the performance gain becomes limited, suggesting that overhead is increasing.

Johnson's algorithm benefits from parallelization for graphs of all sizes, with consistent speedup and reasonable efficiency. However, efficiency decreases at higher thread counts, particularly for smaller graphs, and overhead becomes a limiting factor as the graph size grows.

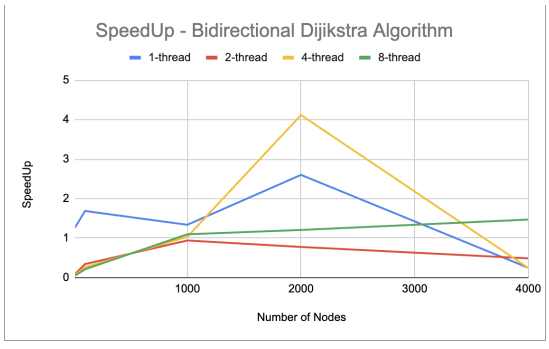
5.1.5 Bidirectional Dijkstra Algorithm

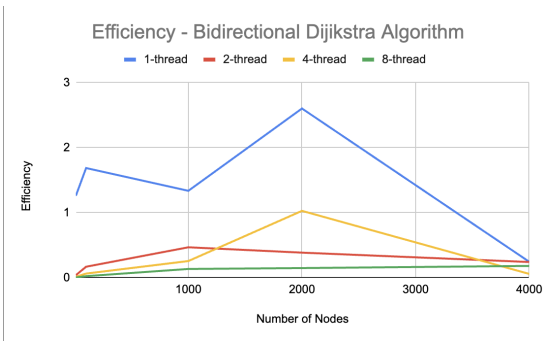
Nodes/Threads	1-thread	2-thread	4-thread	8-thread
10	1.261805	0.082771	0.061970	0.044699
100	1.687899	0.342796	0.264387	0.211826
1000	1.338019	0.940159	1.035264	1.097514
2000	2.602413	0.775823	4.119104	1.207328
4000	0.247645	0.487702	0.244251	1.470394

Table 16: Speedup table for Bidirectional Dijkstra algorithm

Nodes/Threads	1-thread	2-thread	4-thread	8-thread
10	1.261805	0.041385	0.015493	0.005587
100	1.687899	0.171398	0.066097	0.026478
1000	1.338019	0.470080	0.258816	0.137189
2000	2.602413	0.387911	1.029776	0.150916
4000	0.247645	0.243851	0.061063	0.183799

Table 17: Efficiency table for Bidirectional Dijkstra algorithm





Small Graphs (10 nodes): Bidirectional Dijkstra demonstrates significant speedup with parallelization, though efficiency drops sharply as the number of threads increases. At 8 threads, the algorithm's efficiency is minimal, indicating increasing overhead.

Medium Graphs (100 nodes): Speedup continues to improve with additional threads, but efficiency shows a noticeable decline as the thread count increases, especially at 8 threads. The performance is still reasonable for 2 and 4 threads.

Large Graphs (1000-2000 nodes): For larger graphs, bidirectional Dijkstra shows a consistent improvement in speedup with 2-4 threads. However, efficiency starts to degrade significantly at 8 threads, especially for the 2000-node graph.

Very Large Graphs (4000 nodes): At 4000 nodes, speedup behavior becomes erratic. While speedup improves for 8 threads, efficiency drops substantially across all thread counts, indicating a higher overhead that diminishes the benefits of parallelism.

Bidirectional Dijkstra benefits from parallelization for small and medium-sized graphs, showing good speedup for up to 4 threads. However, as the graph size grows and thread count increases, efficiency significantly drops, with overhead becoming a dominant factor, especially for very large graphs.

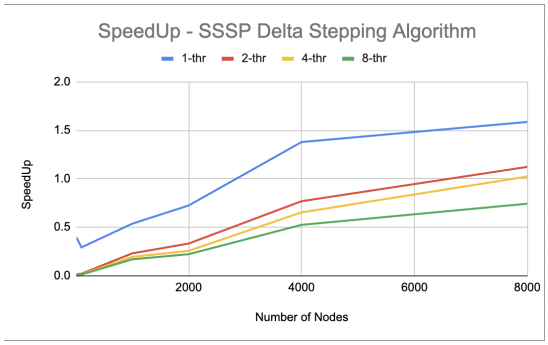
5.1.6 SSSP Delta Stepping Algorithm

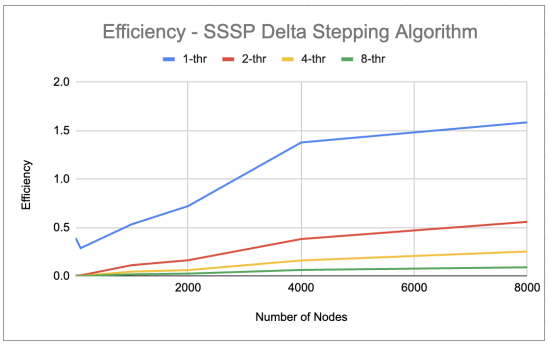
Threads	1-thr	2-thr	4-thr	8-thr
10	0.395049	0.016025	0.012830	0.007895
100	0.290323	0.019254	0.013248	0.010114
1000	0.535172	0.228890	0.192160	0.167343
2000	0.723361	0.330202	0.254705	0.220340
4000	1.378595	0.767489	0.652086	0.523250
8000	1.586021	1.121847	1.022227	0.741658

Table 18: Speedup table for SSSP Delta Stepping Algorithm

Threads	1-thr	2-thr	4-thr	8-thr
10	0.395049	0.008013	0.003207	0.000987
100	0.290323	0.009627	0.003312	0.001264
1000	0.535172	0.114445	0.048040	0.020918
2000	0.723361	0.165101	0.063676	0.027542
4000	1.378595	0.383745	0.163021	0.065406
8000	1.586021	0.560924	0.255557	0.092707

Table 19: Efficiency table for SSSP Delta Stepping Algorithm





Small Graphs (10 nodes): SSSP Delta Stepping Algorithm shows significant speedup with the addition of threads, but efficiency sharply drops as the thread count increases. At 8 threads, the efficiency is very low, indicating a high overhead from parallelization.

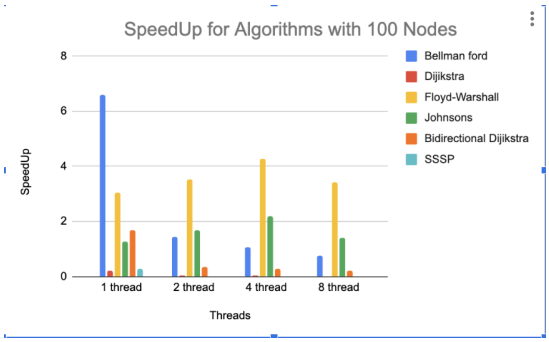
Medium Graphs (100 nodes): Speedup continues to improve with additional threads, but efficiency remains low, particularly at 8 threads. The algorithm benefits from parallelization up to 4 threads but exhibits diminishing returns beyond that.

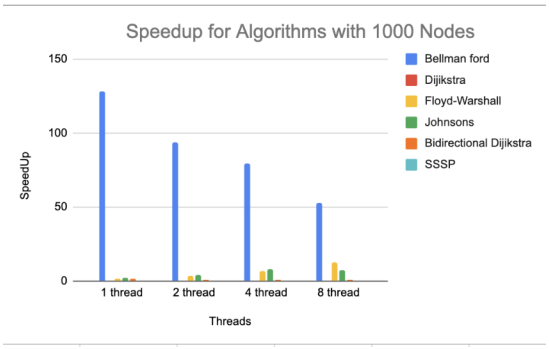
Large Graphs (1000-4000 nodes): For larger graphs, speedup improves with increasing thread count, although it plateaus beyond 4 threads. Efficiency continues to decline as more threads are added, especially for the 4000-node graph, where overhead becomes significant.

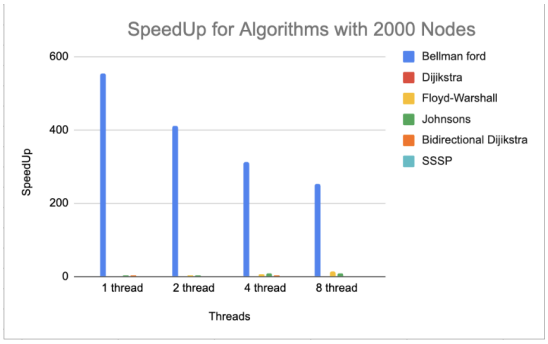
Very Large Graphs (8000 nodes): At 8000 nodes, speedup increases with the addition of threads up to 4 threads, but efficiency decreases sharply as the number of threads increases, indicating that parallelization does not provide significant benefits beyond 2 threads due to increased overhead.

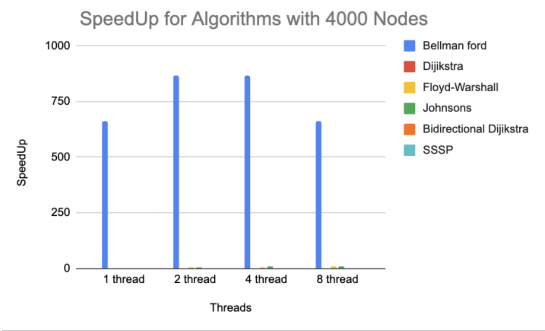
SSSP shows good speedup and reasonable efficiency for smaller graphs, particularly with up to 4 threads. However, as graph size increases and thread count exceeds 4, efficiency degrades significantly, indicating that the algorithm faces overhead challenges in parallelization.

5.2 Cross-Algorithm Speedup Comparison









The bar chart above compares the speedup achieved by each algorithm using 4 threads across varying graph sizes (100 to 4000 nodes). This visualization highlights significant differences in parallelization efficiency among algorithms.

Bellman-Ford stands out with the highest speedup values across all node sizes. Its performance scales exceptionally well with increasing graph size, achieving a speedup exceeding 800x at 4000 nodes. This is due to the algorithm's inherently parallel structure, as each iteration updates all edges independently.

Floyd-Warshall and **Johnson's Algorithm** demonstrate moderate to strong speedup, especially on larger graphs. Floyd-Warshall, in particular, benefits from data-parallelism in its triple nested loop structure, allowing effective work distribution among threads.

SSSP (Delta Stepping) shows a consistent but modest speedup trend, reflecting the complexity of managing dynamic work queues and bucket updates in parallel. Its gains taper off at larger node counts, indicating growing overhead.

Bidirectional Dijkstra and **Dijkstra** perform the worst in terms of parallel speedup. These algorithms involve sequential operations like priority queue updates, which are hard to parallelize without introducing significant synchronization costs. In fact, for large graphs, speedup stagnates or even drops for Dijkstra variants.

Key Takeaways:

- **Best Parallel Scaling:** Bellman-Ford and Floyd-Warshall algorithms show the highest speedup, especially with larger graphs.
- **Moderate Efficiency:** Johnson's Algorithm and SSSP (Delta Stepping) demonstrate reasonable speedup, benefiting from parallelism while facing some overhead.
- **Poor Parallel Performance:** Dijkstra and Bidirectional Dijkstra suffer due to their reliance on sequential structures like priority queues, limiting their scalability.
- **Scalability Trends:** Algorithms with minimal data dependencies and regular computation patterns scale better with thread count.
- **Thread Count Impact:** While 2-4 threads show consistent improvement, higher thread counts (e.g., 8) often introduce diminishing returns due to synchronization overhead.

5.3 Analysis

- **Data Dependencies:** Algorithms like Dijkstra and Bidirectional Dijkstra rely on priority queues and dynamic updates, which introduce sequential bottlenecks that hinder parallel execution. These dependencies make it difficult to efficiently divide work across threads.
- **Computation Structure:** Bellman-Ford and Floyd-Warshall benefit from regular loop-based structures that allow for easy distribution of computation among threads. In particular, Floyd-Warshall's triple-nested loop structure is well-suited to parallelism using loop collapse or tiling strategies.
- **Synchronization Overhead:** Algorithms that require frequent synchronization (e.g., to update shared structures or manage dynamic worklists) experience reduced speedup and efficiency as thread count increases. Delta-Stepping, while parallel in nature, suffers from this due to bucket updates.
- **Scalability Limits:** Some algorithms exhibit good speedup up to 2 or 4 threads, but show diminishing returns or degraded performance beyond that due to increased contention and overhead. This is observed in SSSP and Johnson's algorithms at 8-thread execution.
- **Input Size Sensitivity:** Smaller graphs (e.g., 10 or 100 nodes) often do not provide enough computational workload to offset the overhead of thread management and synchronization. In contrast, large graphs (1000+ nodes) allow better amortization of parallel overhead, yielding higher speedup.
- **Workload Distribution:** Algorithms with evenly distributed and statically known workloads (like Bellman-Ford) achieve better performance, whereas algorithms with dynamic behavior (like Bidirectional Dijkstra) face challenges in load balancing.

6 Conclusion

The performance of graph traversal algorithms under parallel execution varies significantly based on their structural properties and data dependencies. Algorithms like **Bellman-Ford** and **Floyd-Warshall** exhibit strong parallel scalability due to their regular computation patterns and minimal data interdependencies, making them well-suited for high thread-count environments. In contrast, **Dijkstra** and **Bidirectional Dijkstra** demonstrate poor parallel performance, primarily limited by their reliance on inherently sequential structures such as priority queues.

Moderately parallelizable algorithms like **Johnson's** and **Delta-Stepping (SSSP)** strike a balance—benefiting from parallel execution, yet constrained by synchronization overhead and dynamic workload management. Scalability tends to plateau beyond 4 threads for most algorithms, with diminishing returns caused by contention and overhead.

Additionally, **input size plays a critical role** in determining parallel efficiency. Larger graphs provide sufficient workload to leverage parallelism effectively, while smaller graphs often suffer from overhead dominating the performance gains.

Ultimately, designing or selecting graph algorithms for parallel execution requires careful consideration of **data dependencies**, **computation regularity**, and **synchronization patterns**. Algorithms with statically distributable workloads and fewer inter-thread dependencies consistently deliver superior performance and scalability on multicore systems.

7 Future Scope

While current parallel implementations of graph traversal algorithms demonstrate promising performance improvements, there remain several avenues for future exploration:

- **Adaptive Parallel Strategies:** Investigating adaptive methods that dynamically switch between node-centric and edge-centric approaches based on graph characteristics and runtime metrics can improve performance across diverse datasets.
- **Hybrid Models:** Combining parallel and distributed computing paradigms (e.g., OpenMP with MPI or GPU acceleration using CUDA/OpenCL) may enable scalability beyond the limits of shared-memory architectures.
- **Improved Data Structures:** Designing concurrent and lock-free variants of data structures such as priority queues or buckets can reduce contention and enhance performance in algorithms like Dijkstra and Delta-Stepping.
- **Graph Preprocessing Techniques:** Introducing preprocessing steps such as edge reordering or graph partitioning to improve cache locality and load balancing could significantly enhance parallel execution efficiency.
- **Energy-Efficient Computing:** Exploring power-aware scheduling and energy-efficient algorithm design can help in deploying these graph algorithms on resource-constrained or embedded systems.
- **Benchmarking and Real-World Applications:** Further studies on real-world, large-scale graphs (e.g., social networks, road networks) and standardized benchmarking frameworks will provide deeper insights into practical performance and scalability.
- **Compiler and Runtime Optimizations:** Leveraging advanced compiler techniques or runtime libraries for parallelism (e.g., task-based runtimes, work-stealing schedulers) may offer better load distribution and reduced overhead.

In conclusion, continued research in algorithm design, system-level optimization, and heterogeneous computing will play a crucial role in realizing the full potential of parallel graph traversal on modern and emerging hardware platforms.

References

[1] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science 1998: 23rd International Symposium, MFCS'98 Brno, Czech Republic, August 24–28, 1998 Proceedings 23*, pages 722–731. Springer, 1998.

[2] Michael Kainer and Jesper Larsson Träff. More parallelism in dijkstra's single-source shortest path algorithm. *arXiv preprint arXiv:1903.12085*, 2019.

[3] Wenbo Lu, Qianchuan Zhao, and Cangqi Zhou. A parallel algorithm for finding all elementary circuits of a directed graph. In *2018 37th Chinese Control Conference (CCC)*, pages 3156–3161. IEEE, 2018.

[4] Shailendra W. Shende, Abhijit N. Pimple, Bhushan Gajbhiye, and Sheetal R. Radke. Parallelism of graph traversing algorithm using openmp. *International Journal of Engineering Applied Sciences and Technology*, 1(5):149–151, 2016. ISSN 2455-2143.

[5] Thanukrishnan Srinivasan, R Balakrishnan, SA Gangadharan, and V Hayawardh. A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment. In *2007 International Conference on Parallel and Distributed Systems*, pages 1–8. IEEE, 2007.

[6] Ganesh G Surve and Medha A Shah. Parallel implementation of bellman-ford algorithm using cuda architecture. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, volume 2, pages 16–22. IEEE, 2017.

[7] Gintaras Vaira and Olga Kurasova. Parallel bidirectional dijkstra's shortest path algorithm. In *Databases and Information Systems VI*, pages 422–435. IOS Press, 2011.