Parallel and Distributed Single Source Shortest Path Algorithms.

Song Tung Nguyen (301354423), Darryl Julius Basri (301388030), Don Van (301338875)

1 Introduction

Graphs are commonly used in many computer science's fields (such as networking, data mining) to represent a variety of complex data relationships. A graph consists of a collection of nodes (vertices) that are connected by edges that represent the relationships between those two nodes. Moreover, a graph can also be directed / undirected, and weighted. A directed graph is a type of graph in which the edges represent the direction of the vertices, and an undirected graph does not. While a weighted graph means the edges have weight. These edges can represent different things in different situations, such as cost, distance, capacity, etc.

One of the most common problems in graph theory is the Single Source Shortest Path (SSSP) problem. The SSSP problem is about finding the most efficient algorithm to find the shortest path from a node (referred to as the source node) to every other node in the graph. This problem is particularly challenging when the graph is large and computing the shortest path will become inefficient. Many researchers have proposed solutions to the SSSP problem. Two of the most well-known algorithms to the SSSP problem that this paper will focus on are the Dijkstra, and Bellman-Ford. This paper will evaluate these algorithms under three different types of computing: serial, parallel and distributed computing with MPI. Furthermore, to simplify the problem, we will consider the case where the graph is undirected and has a weight of 1.

2 Background

This section reviews the serial version of Dijkstra's algorithm and Bellman-Ford. It is important to understand the basics of the serial version before proceeding to parallelize the algorithms.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that finds the shortest path between two vertices in either a directed or undirected graph whose weights are non-negative. This algorithm was first published in 1956 by computer scientist Edsger W. Dijkstra, and there have been many variants of this algorithm. The most used variant labels a node as the source node and finds the shortest path to every other node in the graph. Furthermore, Dijkstra's algorithm can also be used for finding the shortest path between two nodes by labelling a node as the source and destination node, and stopping the algorithm when the shortest path to the destination node has been computed.

The original Dijsktra's algorithm uses min-priority queue to store partial solutions of the distance between each node from the start. It has a time complexity of O((|V|+|E|) log |V|) where |V| is the number of vertices and |E| is the number of edges. While a more efficient algorithm can be implemented by using a Fibonacci heap as the data structures, as it allows us to insert a new element into the queue in time O(1), and extract the node that has the minimal distance in O(log |V|). So the overall complexity becomes O(|V| + |E| log |V|).

2.2 Bellman-Ford

The Bellman-Ford algorithm finds the shortest path between two vertices in either a directed or undirected graph whose weights can either be negative or non-negative. The idea of the algorithm is there could only be maximum of |V| - 1 (|V| is the number of vertices in the graph) edges in any simple path. Assuming there is no negative weight cycle, if the shortest paths with at most i edges have been calculated, then an interaction over all edges guarantees to give a shortest path with at most (i+1) edges. For negative weights, the algorithm checks at the last iteration for any edge that decreases the weight of the shortest path with |V|-1 edges. If found, then the graph must contain a negative weight as the path becomes |V| edges.

While the Bellman-Ford is well-known for being more versatile than the Dijkstra's algorithm because of its capability to handle negative weights, it is slower than Dijkstra's. On average, we update the weight of all edges every iteration. Since there are |V|-1 iterations, the time complexity becomes O(|V|.|E|). In the worst-case scenario, this complexity could become O(|V|^3) time.

3 Implementation Details

3.1 Serial Dijkstra's Algorithm

The algorithm starts off by marking the distance from the source node to itself zero, and the distance from the source node to all of the other nodes in the graph is marked as infinity. The algorithm keeps a list of visited nodes to prevent visiting a node twice and a priority queue to know which vertex to visit next. In each iteration, the algorithm finds an unvisited vertex with the smallest distance, marks it as visited, and sets it to be the current vertex. The distance from the source node to the current vertex is then calculated using all of the neighbors of the current vertex. If one of the calculated distances is smaller than the current distance associated with the current vertex, it will be then used as the distance from the source to the current vertex. The process repeats until the priority queue is empty.The algorithm is described in detail by the pseudocode written by Aderholdt et al. [1].

```
Require: $G$, a weighted graph
Require: $src$, a source vertex
function DIJKSTRA($G$, $src$)
    for $v$ in $V(G)$ do
        $distance[v] \leftarrow \infty$
        $path[v] \leftarrow null$
        $Queue.push(v, dst)$
    end for
    $Queue.decrease(src, 0)$
    while $Queue$ is not empty do
        $v \leftarrow Queue.pop()$
        for $u$ in $\Gamma(v)$ do
            RELAX($v, u$, edge($v, u$))
        end for
    end while
    return $distance$, $path$
end function
```

```
function RELAX($v$, $u$, $e$)
    $tmp \leftarrow distance[v]$ + weight($e$)
    if $distance[u] > tmp$ then
        $distance[u] \leftarrow tmp$
        $path[u] \leftarrow v$
        $Queue.decrease(u, distance[u])$
    end if
end function
```

Figure 1: Serial Dijkstra's Algorithm Pseudocode by Aderholdt et al. [1].

3.2 Serial Bellman-Ford

Different from Dijkstra's algorithm, Bellman-Ford does not use a priority queue to gredily select the next smallest distance vertex that is unvisited. It simply relaxes all of the edges (|V| - 1) times, in which |V| means the number of vertices. After |V|-1 iterations, the number of vertices, whose distance to the source vector is correctly calculated, will grow and eventually all of the vertices in the graph will be correctly calculated. The algorithm is described in detail by the pseudocode written by Aderholdt et al. [1].

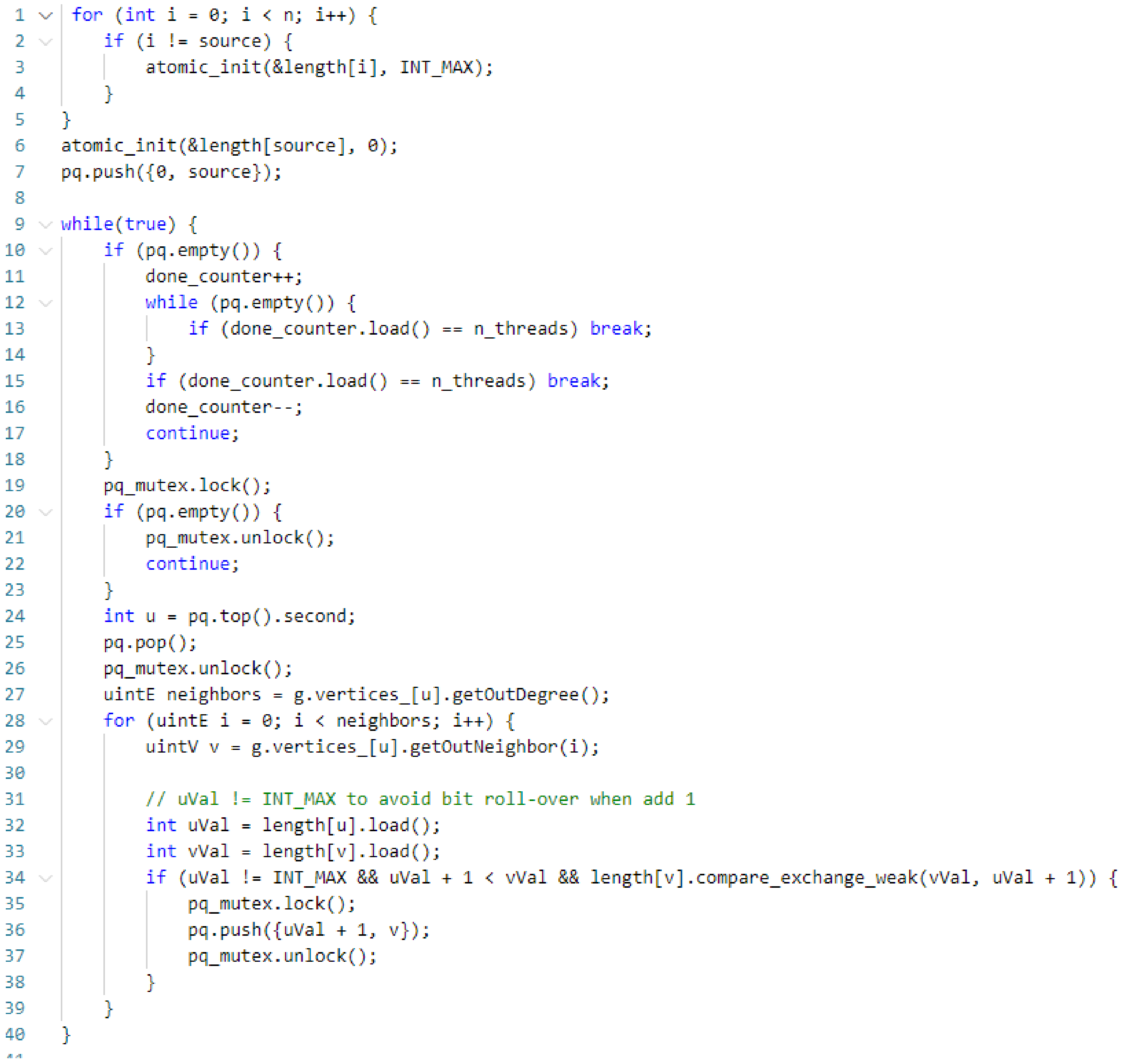
```
Require: $G$, a weighted graph
Require: $src$, a source vertex
function BELLMANFORD($G$, $src$)
    for $v$ in $V(G)$ do
        $distance[v] \leftarrow \infty$
        $path[v] \leftarrow null$
    end for
    $distance[src] \leftarrow 0$
    for $k = 1$ to $|V| - 1$ do
        for $v$ in $V(G)$ do
            for $u$ in $\Gamma(v)$ do
                RELAX($v, u$, edge($v, u$))
            end for
        end for
    end for
    return $distance$, $path$
end function
```

```
function RELAX($v$, $u$, $e$)
    $tmp \leftarrow distance[v]$ + weight($e$)
    if $distance[u] > tmp$ then
        $distance[u] \leftarrow tmp$
        $path[u] \leftarrow v$
    end if
end function
```

Figure 2: Serial Bellman-Ford Pseudocode by Aderholdt et al. [1].

3.3 Parallel Dijkstra’s Algorithm

Since the algorithm is processing the vertices based on the order they are sorted inside the priority queue, it is impossible to determine the order of vertices that will get processed at the beginning. Hence, it is impossible to assign the vertices to each thread upfront. The only solution is dynamic decomposition, which means different threads will operate on the same priority queue and try to pop an element from it for processing when they are available. For our implementation, we assumed that standard priority queue implementations are not thread-safe so our pseudocode will include the use of mutex and atomic operations.



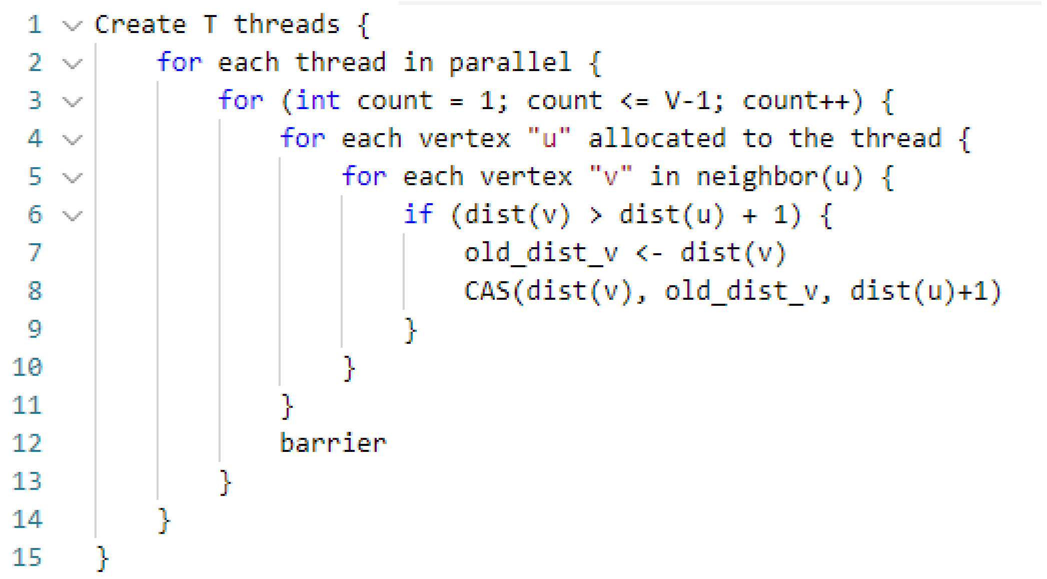
```
for (int i = 0; i < n; i++) {
    if (i != source) {
        atomic_init(&length[i], INT_MAX);
    }
}
atomic_init(&length[source], 0);
pq.push({0, source});

while(true) {
    if (pq.empty()) {
        done_counter++;
        while (pq.empty()) {
            if (done_counter.load() == n_threads) break;
        }
        if (done_counter.load() == n_threads) break;
        done_counter--;
        continue;
    }
    pq_mutex.lock();
    if (pq.empty()) {
        pq_mutex.unlock();
        continue;
    }
    int u = pq.top().second;
    pq.pop();
    pq_mutex.unlock();
    uintE neighbors = g.vertices_[u].getOutDegree();
    for (uintE i = 0; i < neighbors; i++) {
        uintV v = g.vertices_[u].getOutNeighbor(i);

        // uVal != INT_MAX to avoid bit roll-over when add 1
        int uVal = length[u].load();
        int vVal = length[v].load();
        if (uVal != INT_MAX && uVal + 1 < vVal && length[v].compare_exchange_weak(vVal, uVal + 1)) {
            pq_mutex.lock();
            pq.push({uVal + 1, v});
            pq_mutex.unlock();
        }
    }
}
```

Figure 2: Parallel Dijkstra’s Algorithm.

3.4 Parallel Bellman-Ford

Since Bellman-Ford is processing each of the vertex (|V| - 1) times. There are three approaches for parallelizing Bellman-Ford:

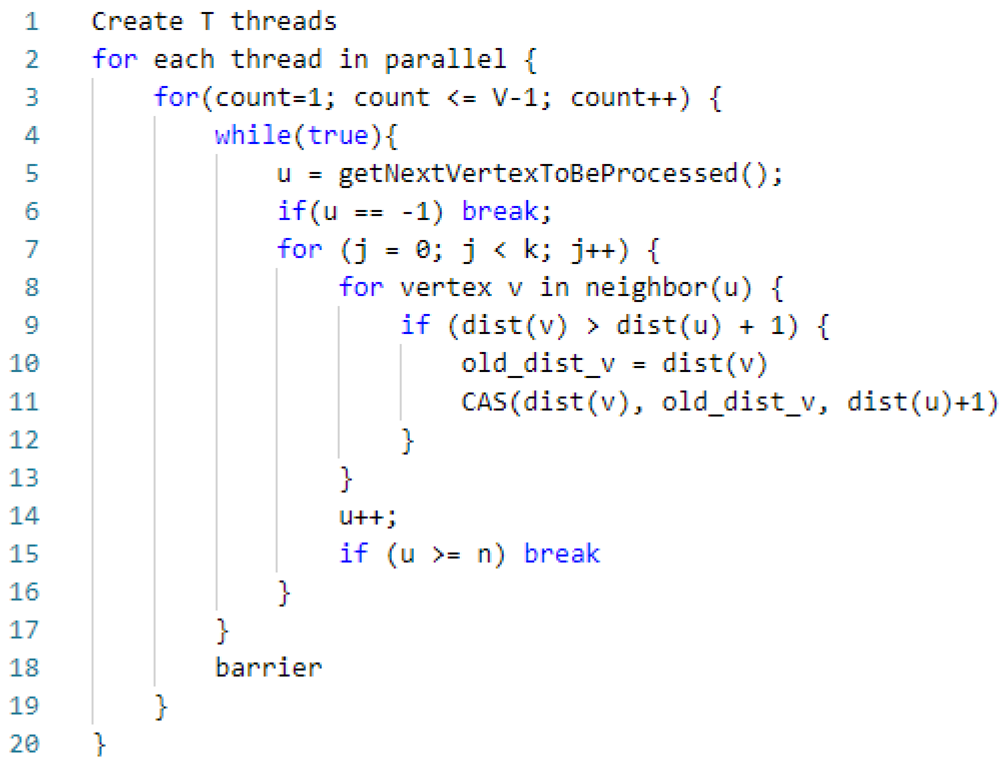
1. **Vertex-Based Decomposition:** This method divides the set of vertices in the graph by the number of threads. The details of this method is shown in the pseudocode.



```
Create T threads {
    for each thread in parallel {
        for (int count = 1; count <= V-1; count++) {
            for each vertex "u" allocated to the thread {
                for each vertex "v" in neighbor(u) {
                    if (dist(v) > dist(u) + 1) {
                        old_dist_v <- dist(v)
                        CAS(dist(v), old_dist_v, dist(u)+1)
                    }
                }
            }
            barrier
        }
    }
}
```

Figure 4: Bellman-Ford with Vertex-Based Decomposition.

2. **Edge-Based Decomposition:** This type of decomposition is skipped in this project due to the large overhead of modifying the Graph class just to make it to work. The Graph class does not have an indexed list of all the edges; it only has the list of edges that connects to a vertex, which makes assigning the edges to each thread harder.
3. **Vertex-Based Coarse Grained Dynamic Decomposition:** Each thread will ask for a new vertex to process once it has finished processing the previous vertex. To reduce the time spent by each thread getting a new vertex, we will add granularity to it so that each thread receives multiple vertices each time it requests a new set of vertices.



```
Create T threads
for each thread in parallel {
    for(count=1; count <= V-1; count++) {
        while(true){
            u = getNextVertexToBeProcessed();
            if(u == -1) break;
            for (j = 0; j < k; j++) {
                for vertex v in neighbor(u) {
                    if (dist(v) > dist(u) + 1) {
                        old_dist_v = dist(v)
                        CAS(dist(v), old_dist_v, dist(u)+1)
                    }
                }
                u++;
                if (u >= n) break
            }
        }
        barrier
    }
}
```

Figure 5: Bellman-Ford with Vertex-Based Coarse Grained Dynamic Decomposition.

3.5 Distributed Dijkstra's Algorithm

The distributed version of Dijkstra was the hardest to implement as previously all of the threads shared the same priority queue, but in the distributed environment where there is no shared memory, it is impossible to do that. The solution is for each process to have its own copy of the priority queue. In each of the priority queues, there are only the vertices that are assigned to the process. In each iteration, each process sends the vertex with the minimum distance value to the master node for comparison, the master node will broadcast back the selected minimum to all of the process so that each process could update the value of that vertex in its own memory. Each process repeats the same actions until all of the priority queues are empty. The algorithm is described in detail by the pseudocode written by Aderholdt et al. [1].

```
Require: $G$, a weighted graph
Require: $src$, a source vertex
 1: function ParallelDijkstra($G$, $src$, $v_{start}$, $v_{end}$)
 2:     for $v_{start} \leq v \leq v_{end}$ in $V(G)$ do
 3:         $distance[v] \leftarrow \infty$
 4:         $path[v] \leftarrow null$
 5:         $Queue.push(v, dst)$
 6:     end for
 7:     $Queue.decrease(src, 0)$
 8:     while $Queue$ is not empty do
 9:         $v \leftarrow$ FindMin($rank$)
10:         for $u$ in $\Gamma(v)$ do
11:             Relax($v, u$, edge($v, u$))
12:         end for
13:     end while
14:     return $distance$, $path$
15: end function
```

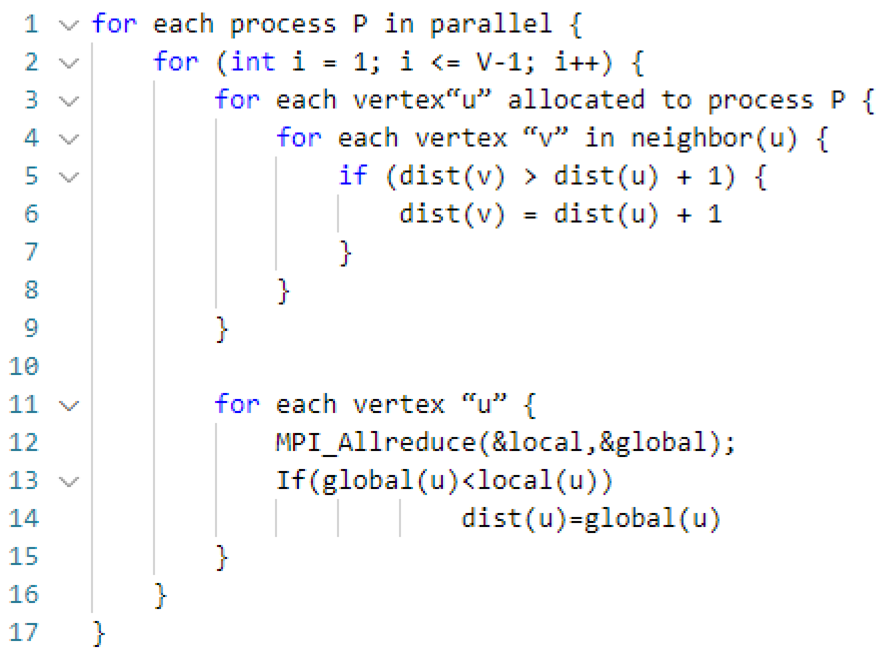
```
 1: function Relax($v$, $u$, $e$)
 2:     $tmp \leftarrow distance[v] +$ weight($e$)
 3:     if $distance[u] > tmp$ then
 4:         $distance[u] \leftarrow tmp$
 5:         $path[u] \leftarrow v$
 6:         $Queue.decrease(u, distance[u])$
 7:     end if
 8: end function
```

```
 1: function FindMin($rank$)
 2:     $p, v \leftarrow Queue.peek()$
 3:     put(shared[$rank$], ($p, v$), Master)
 4:     if $rank$ = MASTER then
 5:         $p, v \leftarrow$min($shared$)
 6:     end if
 7:     bcast($p, v$)
 8:     if $v = Qucuc.pcck()$ then
 9:         $Queue.pop()$
10:     end if
11:     return $v$
12: end function
```

Figure 6: Distributed Dijkstra's Algorithm.

3.6 Distributed Bellman-Ford

The distributed version of Bellman-Ford algorithm is implemented by assigning |V| / N (where N is the number of processors) vertices to each processor. Then at every iteration, each processor relaxed its assigned edges, and broadcast it to all the other processors. The other processor that received a new value for that edge from all the other processors will find the minimum value and call it "global"value. Finally, the processor will check if the global value is less than the local value, if so an update will occur. A pseudocode of the algorithm is given below :



```
for each process P in parallel {
    for (int i = 1; i <= V-1; i++) {
        for each vertex“u” allocated to process P {
            for each vertex “v” in neighbor(u) {
                if (dist(v) > dist(u) + 1) {
                    dist(v) = dist(u) + 1
                }
            }
        }

        for each vertex “u” {
            MPI_Allreduce(&local,&global);
            If(global(u)<local(u))
                        dist(u)=global(u)
        }
    }
}
```

Figure 7: Distributed Bellman-Ford.

4 Evaluation

4.1 Test Method

To test the algorithm for correctness, we have decided to use a small example from GeekforGeeks, which is shown in the picture below [4]. The graph is located in ***input_graphs/GG_Test***.

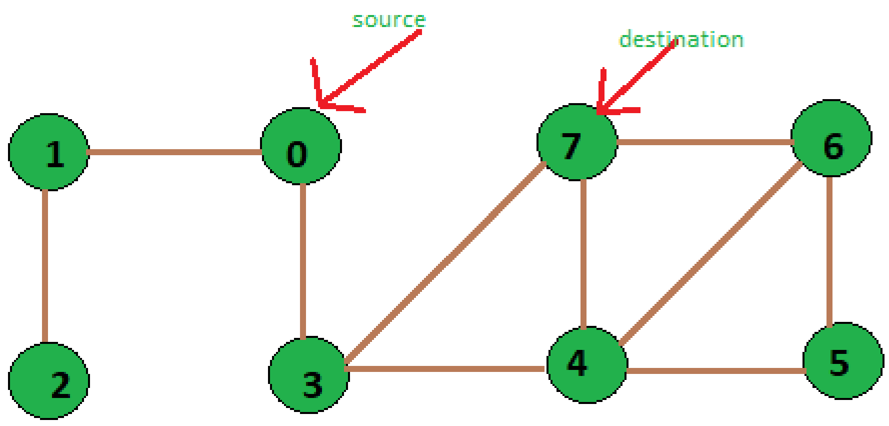
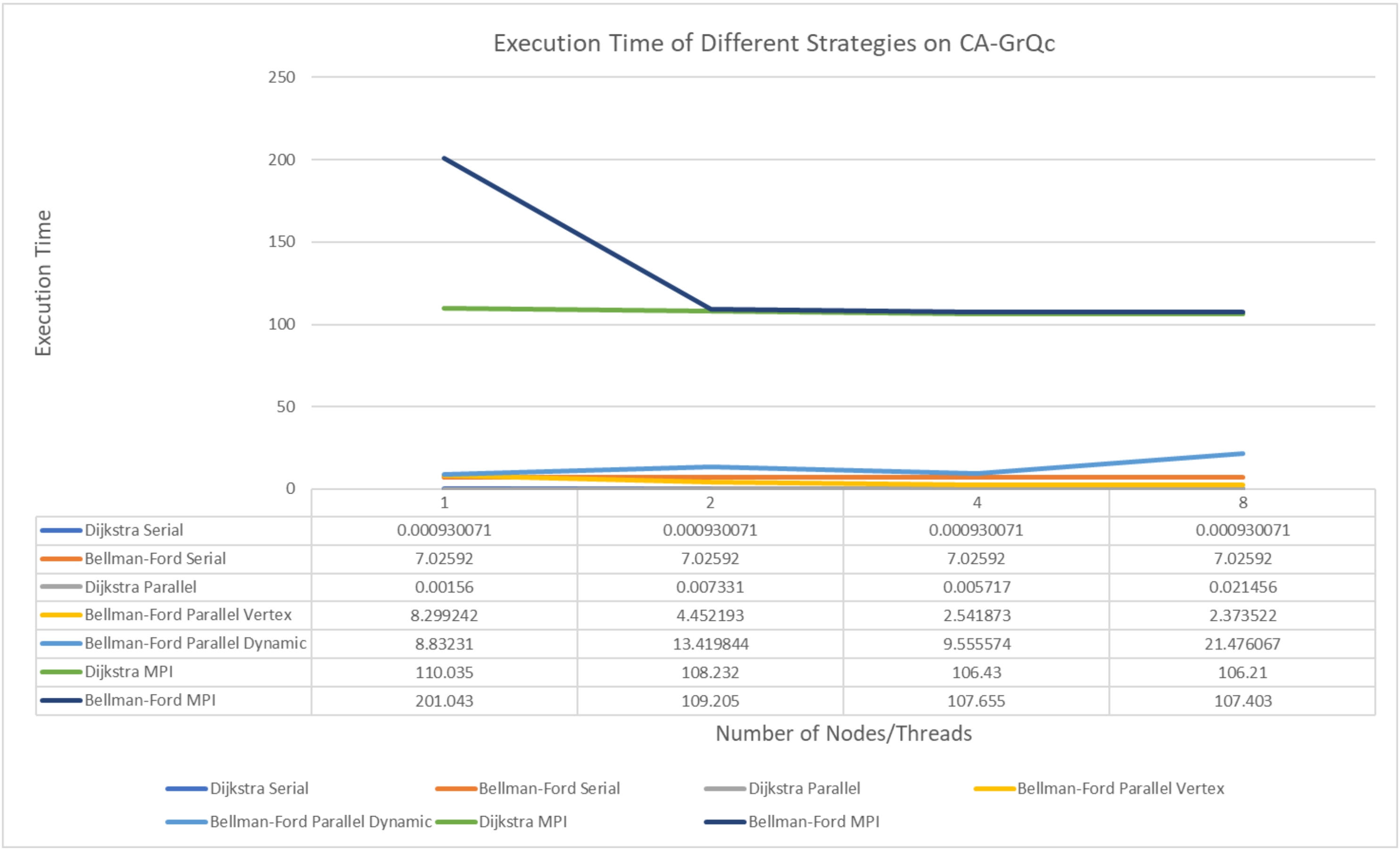


Figure 8: Graph used for testing correctness.

For benchmarking different implementations, we decided to use ca-GrQc and ca-HepTh from Stanford Large Network Dataset Collection. Please note that both of the graphs are multi-graph, meaning there may not be a path between two random vertices. We decided to use this graph because they are reasonably large enough to see the difference between implementations. Moreover, it helps us verify that if there is no path between two vertices, the algorithm would not report a false positive.

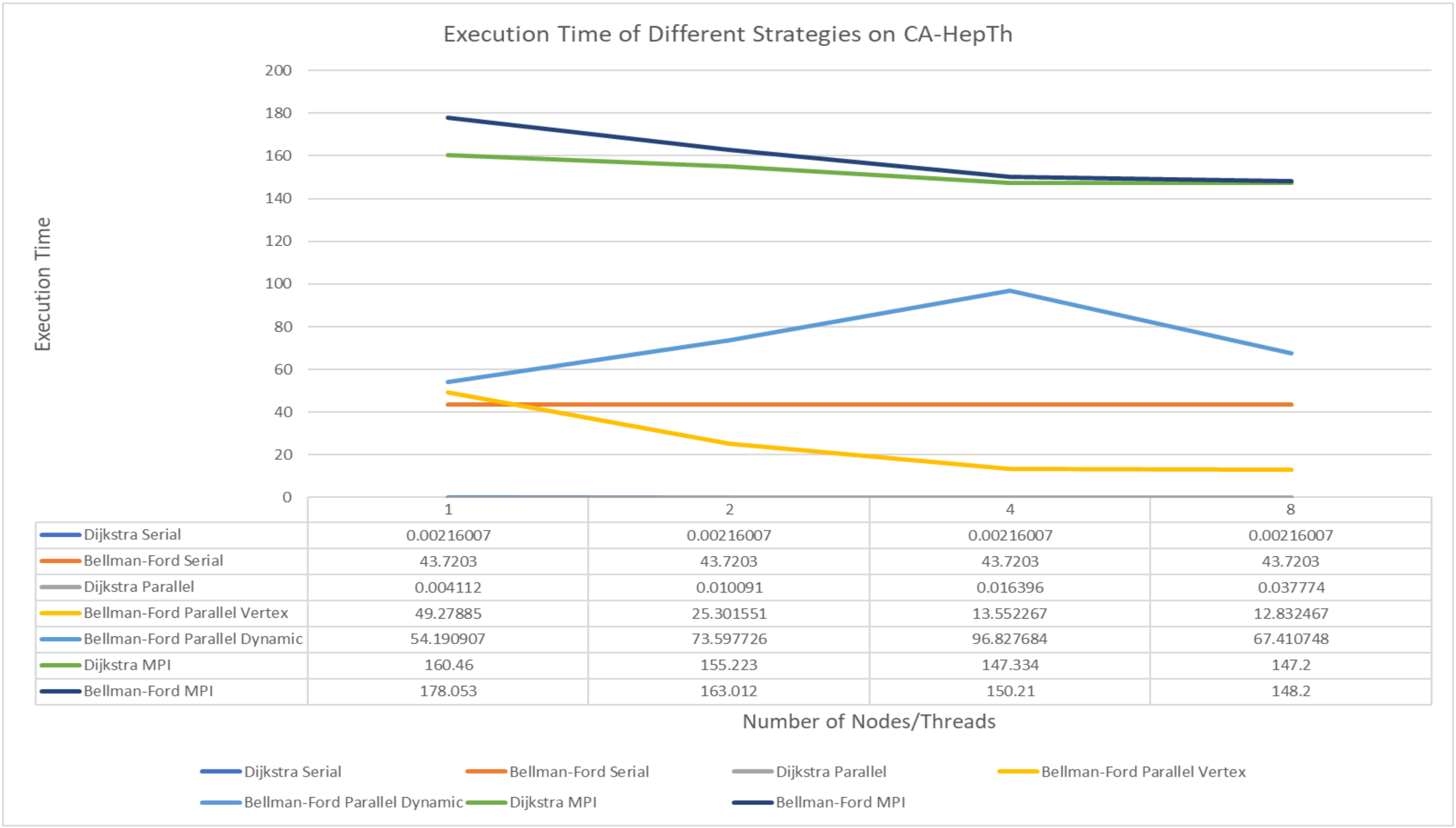
All of the benchmarks are run with slow CPU, 4GB of memory.

4.2 Results



	1	2	4	8
Dijkstra Serial	0.000930071	0.000930071	0.000930071	0.000930071
Bellman-Ford Serial	7.02592	7.02592	7.02592	7.02592
Dijkstra Parallel	0.00156	0.007331	0.005717	0.021456
Bellman-Ford Parallel Vertex	8.299242	4.452193	2.541873	2.373522
Bellman-Ford Parallel Dynamic	8.83231	13.419844	9.555574	21.476067
Dijkstra MPI	110.035	108.232	106.43	106.21
Bellman-Ford MPI	201.043	109.205	107.655	107.403

Figure 9: Execution Time of Different Strategies on Ca-GrQc.



	1	2	4	8
Dijkstra Serial	0.00216007	0.00216007	0.00216007	0.00216007
Bellman-Ford Serial	43.7203	43.7203	43.7203	43.7203
Dijkstra Parallel	0.004112	0.010091	0.016396	0.037774
Bellman-Ford Parallel Vertex	49.27885	25.301551	13.552267	12.832467
Bellman-Ford Parallel Dynamic	54.190907	73.597726	96.827684	67.410748
Dijkstra MPI	160.46	155.223	147.334	147.2
Bellman-Ford MPI	178.053	163.012	150.21	148.2

Figure 10: Execution Time of Different Strategies on Ca-HepTh.

From Figure 9 and 10, the communication overhead that is involved in the parallel and distributed implementations becomes apparent. We see that MPI, which has the highest cost of communication, suffers far more compared to the other algorithms because the actual work needed to be done on the nodes is very light in comparison to the overhead caused by sending and receiving data. There is an exponential speedup that can be seen in the Bellman Ford parallel algorithm that is also evident in the MPI implementation, but is overshadowed by the overhead caused from communication. There is also evidence of a memory bottleneck when dealing with Bellman-Ford algorithms because it has to retain data on all of the vertices. In contrast, since the implementation of Dijkstra's algorithm uses a priority queue, it will not benefit as much from more threads or cores because there is a bottleneck presented by popping from the priority queue and retaining correctness. This downside is negated by the fact that Dijkstra's algorithm requires a relatively miniscule amount of memory to function smoothly when the Bellman Ford algorithms could barely function with 1 GB. Due to our testing conditions such as using the cluster to reduce variation, the Bellman Ford algorithms were handicapped with the limited cpu cores and memory that could be allocated to any process.

5 Conclusion

In conclusion, Bellman-Ford's algorithm runs exponentially slower on both the serial and parallel implementations when compared to Dijkstra's algorithm. This result is to be expected since Bellman-Ford's run on $O(|V|.|E|)$ time complexity, while Dijkstra's run in $O(|V|+|E| \log |V|)$. Thus, the Bellman-Ford algorithm should only be used with negative weight edges on the graph.

6 References

[1] F. Aderholdt, J. A. Graves, and M. G. Venkata, "Parallelizing single source shortest path with OpenSHMEM," *Lecture Notes in Computer Science*, pp. 65–81, 2018.

[2] A. D. Kshemkalyani and M. Singhal, "Distributed computing," 2008.

[3] J. B. Orlin, "A faster algorithm for the single source shortest path problem with few distinct positive lengths" *Journal of Discrete Algorithms,* pp. 189-198, 2010.

[4] "Shortest path in an unweighted graph," *GeeksforGeeks*, 09-Nov-2021. [Online]. Available: https://www.geeksforgeeks.org/shortest-path-unweighted-graph/. [Accessed: 16-Apr-2022].

[5] M. Alshammari, A. Rezgui,"A single-source shortest path algorithm for dynamic graphs" *AKCE International Journal of Graphs and Combinatorics,* pp 1063-1068, 2020.