SCHOOL OF
COMPUTING SCIENCE

CMPT 431

FINAL PROJECT REPORT

Parallel Approaches to Single Source Shortest Path Problem using multithreading and MPI

Instructor: Alaa Alameldeen

Zhenao Li - 301424546

Lei Dong - 301179836

Canh Nhat Minh Le – 301384865

April 19, 2022

I. Introduction

The Single-Source Shortest Path (SSSP) problem is a central problem in graph theory. A graph includes a set of nodes, called vertices, together with a set of unordered pairs of these nodes for an undirected graph or a set of ordered pairs for a directed graph. These pairs are called edges or arcs for directed and undirected graphs. And graphs are usually implemented as data structures by adjacency list and adjacency matrix. SSSP is a problem that consists of finding the paths between a given vertex and all other vertices in the graph such that the sum of the edges in any given path is minimized. On top of that, based on the weights of edges and direction of the graph, the problem can then be split into SSSP with undirected graphs, unweighted graphs, directed acyclic graphs, directed graphs with nonnegative weights, directed graphs with arbitrary weights without negative cycles, directed graphs with arbitrary weights with negative cycles, and planar graphs with arbitrary weights (Shortest path problem, 2022). In this project, we will specifically use serial, parallel and distributed techniques to solve SSSP with directed graphs with equal weights with the value as one, which can be considered as unweighted graphs.

II. Background

Currently, the most popular and mentioned algorithms to solve the SSSP problem serially are the Dijkstra's algorithm, Bellman-Ford algorithm, and Breadth-first search (He, 2021). The Dijkstra's algorithm, proposed by computer scientist Edsger W. Dijkstra, is for solving the SSSP problem with non-negative edge weight (He, 2021). The Dijkstra's algorithm adopts Greedy algorithm and uses a data structure, originally min-priority queue, to store and query partial solutions sorted by distance from the start (Shortest path problem, 2022). On top of that, the algorithm can be optimized by using Fibonacci heap min-priority queue to the time complexity of O(E + VlogV) (Shortest path problem, 2022).

Compared to Dijkstra's algorithm, Bellman-Ford algorithm can solve not only the SSSP problem with non-negative edge weights, but also the problem if edge weights may be negative. The algorithm is able to detect and report the existence of a negative cycle reachable from the source vertex in the graph where any path that has a point on the negative cycle can be made shorter by one more walk around the negative cycle (Shortest path problem, 2022). Although the Bellman-Ford algorithm is more versatile than Dijkstra's, it has a higher time complexity of O(VE) (Shortest path problem, 2022).

Breadth-first search (BFS) algorithm is usually implemented to solve the SSSP problem with unweighted or equally weighted edges. It normally uses a queue to keep track of the adjacent vertices but is not yet explored (Shortest path problem, 2022). It determines the distance of the shortest path based on the layer the destination vertex is at. Because in BFS, vertices that take lesser edges to travel from the source are always visited earlier, and the edges are equally weighted, the correctness of BFS can be proven. BFS is also a fairly quick solution that has a time complexity of O(E + V).

On top of the serial implementations, a major popular algorithm for solving the SSSP problem in parallel is the Delta-stepping algorithm proposed by U.Meyer and P. Sanders (Meyer & Sanders, 1998). It is a label-correcting algorithm and the tentative distance of a vertex can be corrected several times through edge relaxations until the last step of the algorithm (Meyer & Sanders, 1998).

Another major popular algorithm is parallel Dijkstra's algorithm. It splits vertices to each process and then has each process to find the locally stored vertex with the shortest distance from the source (He, 2021). Then, the processes determine the vertex with the shortest distance and include it. After that, each process updates the distance array (He, 2021).

III. Implementation

1. Serial SSSP implementation:

Pseudocode:

```
void dijkstra_serial(graph, source_vertex)
// Initialization
   Queue queue
   vertex current_vertex
   int num_of_vertex = graph.num_of_vertex
   int* dist_array[num_of_vertex]
   int* prev_vertex_array[num_of_vertex]
   Initialize dist_array to INT_MAX for each element
   Initialize prev_vertex_array to INT_MAX for each element

// Enqueue the source vertex and set its distance to itself to 0
   queue.enqueue(source_vertex)
   dist_array[source_vertex] = 0
   prev_vertex_array[source_vertex] = source_vertex

   while (queue is not empty)
      current_vertex = queue.dequeue()
      for every vertex adjacent_vertex adjacent to current_vertex
         // Update if the adjacent vertex has not been visited
         if dist_array[adjacent_vertex] == INT_MAX
            dist_array[adjacent_vertex] = dist_array[current_vertex] + 1
            prev_vertex_array[adjacent_vertex] = current_vertex
            queue.enqueue(adjacent_vertex)
```

In this section, we will present our queue-based breadth-first search to solve the SSSP problem. Because we are dealing with directed graphs with equal weights of value 1, the shortest distance of a vertex to the source is the number of edges between them in a BFS path, hence the number of layers between them. We use STL Queue as the FIFO queue to store and query vertices. The queue is first initialized with the sourced vertex enqueued. We also use an array, dist_array, to store the shortest distance from the source to each vertex, and the value for each element is initialized as INT_MAX, except for the source vertex, which is set to 0. We then use another array, prev_vertex_array, to store the previous node adjacent to each vertex on the shortest path from the source. The process will then dequeue each vertex in the queue until the queue is empty. After the process dequeues each vertex, current_vertex, it gets the adjacent vertices, v, to that vertex and checks whether the vertex has been previously visited by checking whether the dist_array[v] equals to INT_MAX. If v has not been previously visited, we update dist_array[v]

= dist_array[current_vertex] + 1 and we set prev_vertex_array[v] = current_vertex. The work is completed once the queue is empty. Vertices not reachable from the source will have a distance of INT_MAX. The algorithm has a time complexity of O(V + E).

2. Parallel SSSP implementation:

Pseudocode:

```
Initialization:
    Create struct thread_obj:
        Pointer to g_queue_1
        Pointer to g_queue_2
        Pointer to the graph
        Pointer to dist_array
        Pointer to barrier,
        Pointer to prev_vertex_array
    Enqueue source_vertex, set dist_array[source_vertex] = 0,
    prev_vertex_array[source_vertex] = source_vertex
Parallel Execution:

    At each process in parallel:
        While check_if_empty == false:
          While g_queue_1.dequeue is successful:
            For every adjacent_vertex adjacent to current_ vertex:
              if (adjacent_vertex has not been visited):
                dist_array[adjacent_vertex] = dist_array[current_vertex]
                prev_vertex_array[adjacent_vertex] = current_vertex
                g_queue_2.enqueue(adjacent_vertex)
        barrier.wait()
        Swap g_queue_1 and g_queue_2
        if (pid == 0):
          if (g_queue_1.dequeue is true):
              Perform dequeuing and enqueuing like above
          else:
              check_if_empty = true
        barrier.wait()
```

Based on the serial implementation, we created a starter function that would initialize the necessary variables such as threads, queues, and arrays. In order to make the code more organized, we utilized the OOP concept by creating a struct called thread_obj to store the parameters that are required to be passed into the function that will be executed in parallel called dijkstra_task.

The tasks are dynamically distributed among each processes, with each thread handle one layer of the graph. Instead of using the C++ queue like the serial program, we decided to applied the non blocking concurrent queue that has already been implemented in the previous assignment. The usage of concurrent queues ensure that synchronization between threads are maintained. For the parallel implementation, we initialized two queues: g_queue_1 and g_queue_2. At each

processes, the g_queue_1 would be dequeued to get the current vertex, called current_vertex. After getting the neighbours of current_vertex，we then performed checking if each vertices have been visited or not. If any vertex has not been visited, we update the distance array dist_array[v], and set prev_vertex_array[v] =current_vertex. Finally, that vertex is enqueued to g_queue_2.

We keep doing so until the first queue, g_queue_1 is empty. Next, the two queues are swapped.

Two avoid any processes dequeueing from an empty queue, we set a condition for the first process, with pid = 0 to check if g_queue_1 is empty. If g_queue_1 is empty, we set the atomic bool variable check_if_empty to true, notifying other processes that the g_queue_1 is empty, and thus all the nodes have been reached.

3. MPI implementation:

Pseudocode:

1. Distribute all the N vertices equally to each process
2. Initialization:
   - 2.1 for each process:
     - local_min_dist [0, …, local_n] = Infinity
     - local_pred[0,…,local_n] = source
     - visited[0,….,local_n] = false
   - 2.2 at corresponding process:
     - local_min_dist[source] = 0
     - local_min_dist[neighbor of source] = 1
     - visited[source] = true
3. For step = 0 to N-2:
   - 3.1 In each process, find local_min_v such that local_min_dist[local_min_v] is smallest. Call it local_min_dist.
   - 3.2 find global_min_v such that its local_min_dist is smallest compared to all the other local_min_dist. call this min_dist tobe global_min_dist
   - 3.3 visited[global_min_v] = true at corresponding process
   - 3.4 at corresponding process:
     - let nb_v = neighbors of global_min_v
     - if (global_min_dist+1 < local_min_dist[nb_v]):
       - local_min_dist[nb_v] = global_min_dist+1
       - local_pred[nb_v] = global_min_v

The distributed version of SSSP using MPI is implemented based on Dijkstra's algorithm. Each process is assigned equally with (total/number of processes) vertices, and the remaining vertices are assigned to the last process. Each process maintains two int arrays which are called local_dist and local_pred, respectively. Local_dist array records the minimum distance of each local vertex to the source vertex and local_pred array records each local vertex's predecessor on its shortest path to the source vertex. In addition, each process maintains a bool array called marked which records if a vertex has been visited. If a vertex has been visited, its minimum distance and predecessor will not be changed. The minimum distance and predecessor of the source vertex are initialized to be 0 and itself respectively. It is also marked as visited. The minimum distance and

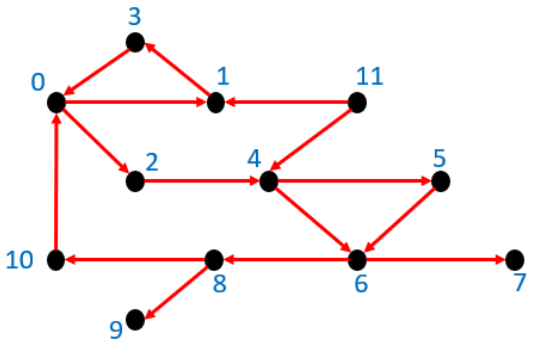
predecessor of each source vertex's neighbor are initialized to be 1 and the source vertex, respectively.

The minimum distance and predecessor of all the vertices except for the source are determined in a for-loop of (total-1) steps. In each loop, the vertex with the global minimum distance is found out and marked as visited. Each process finds out its own vertex with the local minimum distance. The pair of the local minimum distance and the corresponding global vertex number is passed to MPI_Allreduce function using MPI_MINLOC operation. This function finds out the global minimum distance among all the local minimum distances and broadcasts a pair of global minimum distance called global_min_dist and the corresponding global vertex number to all the processes. This vertex with the global minimum distance called global_min_v is marked as visited. After receiving global_min_v, the process containing the neighbors of this vertex checks if the neighbor vertices are visited. If they are not visited, then the new distance which is (global_min_dist + 1) is compared with their current minimum distance. If the new distance is smaller, then the current minimum distance is updated to be the new distance and the predecessor is updated to be global_min_v. This process is repeated (total-1) times and all the vertices are marked as visited at the end. After finishing the for-loop, each process send their local_dist array and local_pred array to the root process through MPI_Gatherv function. At the end, the root process will print out the minimum distance to the source vertex and predecessor on the path for each vertex.

IV. Evaluation of the implementations:

1. Correctness of MPI and parallel implementations:

Sample graph:



SSSP_MPI Sample result:

```
Source vertex : 4
Vertex_id,          min_distance,         Predecessor
0,      4,      10
1,      5,      0
2,      5,      0
3,      6,      1
4,      0,      4
5,      1,      4
6,      1,      4
7,      2,      6
8,      2,      6
9,      3,      8
10,      3,      8
11         No path to source vertex
Source vertex : 0
Vertex_id,          min_distance,         Predecessor
0,      0,      0
1,      1,      0
2,      1,      0
3,      2,      1
4,      2,      2
5,      3,      4
6,      3,      4
7,      4,      6
8,      4,      6
9,      5,      8
10,      5,      8
11         No path to source vertex
```

SPPP_parallel Sample Output:

```
Vertex_id,    min_distance,     predecessor
0,   0,   0
1,   1,   0
2,   1,   0
3,   2,   1
4,   2,   2
5,   3,   4
6,   3,   4
7,   4,   6
8,   4,   6
9,   5,   8
10,   5,   8
11 No path,    No previous vertex

Vertex_id,    min_distance,     predecessor
0,   4,   10
1,   5,   0
2,   5,   0
3,   6,   1
4,   0,   4
5,   1,   4
6,   1,   4
7,   2,   6
8,   2,   6
9,   3,   8
10,   3,   8
11 No path,    No previous vertex
```

2. Efficiency of three implementation:

Table 1. The execution time in seconds for three implementations for different graphs

Number of vertices/edges	Number of threads/processors	11 / 15	7115 / 103689	77360 / 905468	1.8 million
Serial	N/A	0.000001920	0.000332832	0.0113461	0.218929
Parallel	2	0.000341892	0.000272036	0.0366838	0.939442
MPI	2	0.0000720024	0.049	N/A	N/A
Parallel	4	0.000555992	0.000391006	0.0550761	0.930443
MPI	4	0.000598192	0.0161421	N/A	N/A
Parallel	8	0.00120497	0.000746965	0.0554721	0.925677
MPI	8	0.000458002	0.00	N/A	N/A

The MPI implementation does not work properly with the graph with more than 30000 vertices. The segmentation fault is thrown with large graphs and the issue has not been resolved. The guess is that it is related to memory allocation. According to the result of MPI implementation from the graphs with 11 vertices and 7115 vertices, the running speed is slower with more processors. This result is expected because the overhead caused by the communication between the distributed systems can not be overcome if the workload is small. The parallel implementation follows the same trend with small graphs. However, the result from the graph with 1.8 million vertices indicates that the overhead of communication can be overcome if the workload is sufficiently large. The serial implementation has a consistent excellent performance for the tested graphs. It indicates that the BFS algorithm is an efficient way to solve single source shortest path for unweighted graphs.

V. Conclusion:

In graph theory, the Single-Source Shortest Path (SSSP) is one of the most central problems that are addressed regularly. It requires finding the shortest paths between a given source vertex and

other vertices in the graph such that the sum of the edges in any given path is minimized. Our project focuses on solving the problem using the Breadth-First Search algorithm, implemented in three different techniques: serial, parallel, and distributed with MPI. After finishing programming, we tested each program on four different graphs with three distinct number of processes and timed the execution time for each case. Based on the result, we can conclude that for the Breadth First Search algorithm, the serial implementation is the fastest and most efficient, whereas our distributed implementation appears to take the longest time.

VI. References

He, M. (2021). *Parallelizing Dijkstra's Algorithm*.

Meyer, & Sanders. (1998). Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 114-152.

Shortest path problem. (2022, April 19). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Shortest_path_problem#Single-source_shortest_paths