15-418/618 Project

Parallel Shortest Path Algorithms

Xinqi Wang (Andrew ID xinqiw)
Yuou Lei (Andrew ID yuoul)

1 SUMMARY

In this project, we investigated multiple parallelized implementations of two classic algorithms for finding shortest paths to a single source in a give graph: the Dijkstra's Algorithm and the Bellman-Ford Algorithm. We parallelized both algorithms with Message Passing Interface (MPI) and Open Multi-Processing (OpenMP), and analyzed the performance of parallelized models with sequential models on the GHC and PSC machines. Graphs and findings of our evaluation are presented and discussed in the Results section.

2 BACKGROUND

2.1 Graphs and the Single Source Shortest Path Problem

A graph consists of vertices and edges connecting pairs of vertices. Graphs are applied in many real world applications to represent connections and relationships. For example, in social media networks, in video games, in mapping systems, etc. To model different characteristics of different applications, various types of graphs are introduced. In this project, we consider the weighted and undirected graph type. A weighted graph is a graph where each edge is associated with a weight value that is typically positive. This type of graphs are often seen in optimization problems like path planning applications. We assume all weights are positive in this project. An undirected graph is a graph where all edges are bidirectional. As shown in Figure 9, a weighted, undirected graph is a graph where all edges can be numerically ordered an are bidirectional.

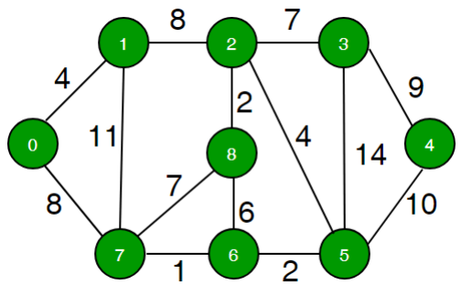

Figure 1: A weighted, undirected graph [4]

Graphs are typically represented by either adjacency matrices or adjacency lists ,as shown in Figure 2. As can be seen, an adjacency matrix has an $O(1)$ access time complexity and $O(V^2)$ space complexity, whereas adjacency lists have $O(V)$ access time and takes up $O(E)$ space. In this project, depending on the algorithm, different graph representations are implemented

to optimize performance resource usage.

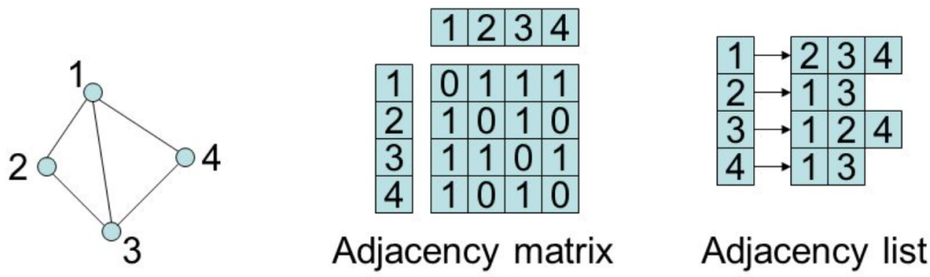

Figure 2: Graph Representation [1]

In this project, we aim to parallelize a well-known graph problem: the Single Source Shortest Path problem. Given a graph G and a source S in the vertex set V, the Single Source Shortest Path problem finds the shortest path from each of the vertices in V to S. For example, in Figure 9, given $S = 0$, the shortest paths from vertices {0,1,2,3,4,5,6,7,8} are {0,4,12,19,21,11,9,8,14}, respectively.

2.2 Bellman-Ford Algorithm

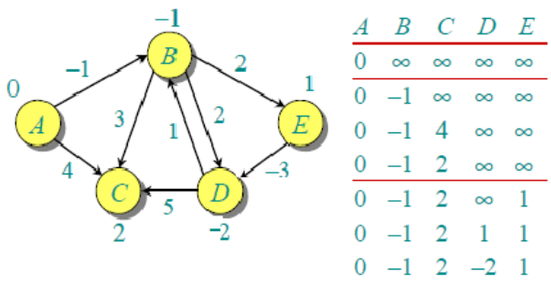

Figure 3: Bellman Ford Algorithm [2]

Algorithm 1 Bellman Ford(Graph, source)

```
    for vertex v in Graph.Vertices do
2:      dist[v] ← INFINITY
        prev[v] ← UNDEFINED
4:  end for
    dist[source] ← 0
6:  for vertex v in Graph.Vertices do
        for edge e in Graph.Edges do
8:          alt ← dist[u] + Graph.Edges(e)
            if alt < dist[v] then
10:             dist[v] ← alt
                prev[v] ← u
12:         end if
        end for
14: end for
    return dist[], prev[]
```

The Bellman Ford Algorithm, shown in Figure 3, is a popular single source shortest path finding algorithm that estimates the shortest paths from all other nodes to the source node

by iterating through all edges in the graph. By iteratively walking through paths of one edges, two edges, and so on, Bellman Ford algorithm guarantees the optimal results after $V - 1$ iterations.

2.3 Dijkstra's Algorithm

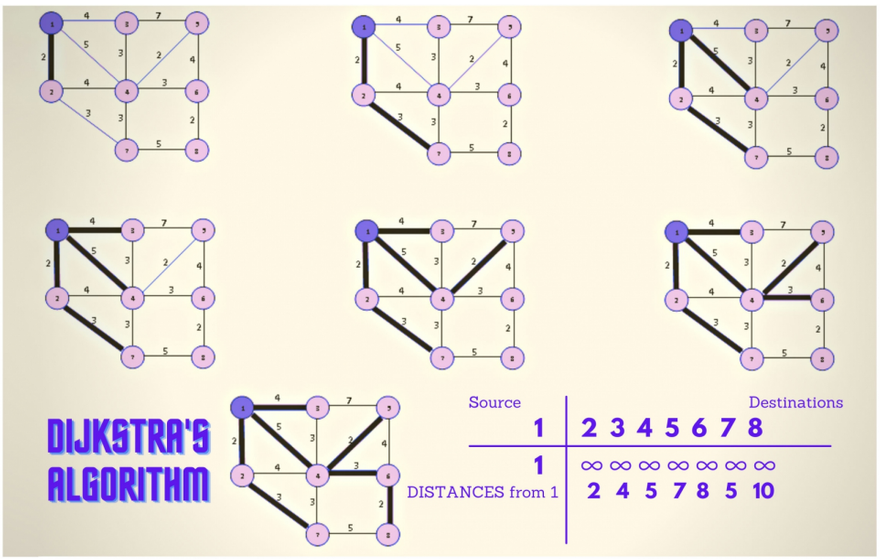

Figure 4: Dijkstra's Algorithm [3]

The Dijkstra's algorithm is a greedy algorithm for finding the shortest paths from a source node to all other nodes in a given graph, as illustrated in Figure 4. One inherent dependency in Dijkstra's Algorithm is the fact that each update to the shortest paths from each vertices to the source node must be made based on the addition/introduction of the node from previous iteration. Given the fact that Dijkstra's Algorithm is a greedy algorithm, the *while* loop must be executed sequentially to find the optimal solution and is a potential bottleneck.

Algorithm 2 Dijkstra(Graph, source)

```
1:  for vertex v in Graph.Vertices do
2:      dist[v] ← INFINITY
3:      prev[v] ← UNDEFINED
4:      add v to Q
5:  end for
6:  dist[source] ← 0
7:  while Q is not empty do
8:      u ← vertex in Q with min dist[u]
9:      remove u from Q
10:     for neighbor v of u still in Q do
11:         alt ← dist[u] + Graph.Edges(u, v)
12:         if alt < dist[v] then
13:             dist[v] ← alt
14:             prev[v] ← u
15:         end if
16:     end for
17: end while
18: return dist[], prev[]
```

3 APPROACH

Since both of the targeted shortest path finding algorithms optimize solutions over multiple iterations and a significant portion of the computation involves conditional operations and thus would cause constant divergence, we suspected that a CUDA parallelized version running on GPU will unlikely to utilize the available computing power to full extent to yield significant speedup. Since there is inherent dependence on results from a prior iteration, data communication between processors is inevitable and synchronization must be done to arrive at a correct optimal solution. Thus, we decided to focus our tasks and analysis on the parallelization of these two algorithms with two different parallel programming models, the shared-address space model and the message-passing model, on different data scales. To do this, we implemented the parallellized Dijkstra's and Bellman-Ford algorithms using both OpenMP and MPI and compare their performance against single threaded serial implementations on both GHC and PSC machines.

3.1 Sequential Dijkstra's Algorithm

We initially implemented the serial Dijkstra's algorithm with a graph represented as an adjacency matrix. However, our first attempts with graphs represented as adjacency matrices caused processes to be killed due to not enough memory with large graph sizes. Considering that it is rare that an adjacency matrix is full, meaning all nodes are connected to all or most of other nodes, we decided to modify our data structure and use adjacency lists to represent graphs more compactly. For both serial and parallelized Dijkstra's algorithms, the input graph includes all adjacency lists and corresponding weights for each adjacent pair of nodes.

3.2 Parallel Dijkstra with OpenMP

As mentioned in previous sections, the inherent dependence in Dijkstra's Algorithm makes it difficult to parallelize the outer loop since that would yield incorrect shortest paths. Therefore, our first attempt divides all independent work evenly among available thread workers. These tasks include initializing distance and parent node arrays, finding the node currently with the minimum distance to the source node, and finally iterating the neighbors of this node to update their shortest paths.

3.3 Parallel Dijkstra with MPI

We also parallelized Dijkstra's Algorithm using MPI for a message-passing model. The core idea of parallelism using MPI is similar to OpemMP. We evenly separate the work of computing the shortest path of each iteration to each process. The process will handle part of the nodes, and after each process finishes its work, our algorithm will use Allreduce API call to collect all the 'next node' candidates of each process, retrieves the node with shortest distance to the source node among the candidates and broadcasts this node to all other threads. Then, all threads proceed to update its portion of nodes that are connected to the selected node and update their distance to the source accordingly. Finally, when the algorithm successfully get all paths of the nodes, we will combine the previous metrics of each process to get a global previous metric. In this case, we can always find the global shortest next node, which make the parallelism can get the same result as sequential version.
One pitfall of the MPI version is that because this is a message-passing model, all cooperation are synchronized using messages. This makes workload balancing harder than in the OpenMP case where we assume a shared memory among processes, since our graph is represented as adjacency lists. When we statically assign each worker a specific range of nodes to update, some threads might make more updates in an iteration if it's responsible for more neighbors nodes of the selected node. Unlike in the OpenMP version, where the neighbor nodes are assigned evenly to all available nodes to update, dynamically assigning neighbor nodes will require frequent massage communication to all threads and likely generate more overheads.

3.4 Sequential Bellman Ford's Algorithm

We implemented the serial Bellman Ford algorithm with a pre-defined data structure consisting of the edges in the graph and the weights of each edge. Because Bellman Ford iterates through all edges, this data representation enables more efficient data manipulation and computation. Whereas Dijkstra's Algorithm has $O((|E| + |V|)log|V|)$ time complexity with a priority queue, Bellman-Ford algorithm has $O(|V||E|)$ complexity and thus is expected to have a less optimal performance. Still, because each iteration of the outer loop "relaxes" all edges once and since this task can be done independently for each edge, there is potential parallelism for optimizing the algorithm.

3.5 Parallel Bellman-Ford with OpenMP

The OpenMP parallelized Bellman Ford executes the inner for loop of the algorithm in parallel. During each iteration of the outer loop, all edges are assigned evenly to available

threads to process and update the corresponding nodes if needed. Notice that while Dijkstra's Algorithm is parallelized by assigning nodes to different workers, Bellman Ford Algorithm is parallelized by assigning all edges to different workers.

3.6 Parallel Bellman-Ford with MPI

The MPI parallelized Bellman Ford is more complicated than the OpenMP implementation, since there is no shared address space. By dividing up the edges to be iterated through among the thread workers, each worker updates its local copy of the distance array and parent array. One problem here is although it's easy to reduce all distance arrays to a global one using an Allreduce() operation in minimum mode at the end of each iteration, the parent arrays can not be easily reduced to the correct one using this operation. Each element in the correct parent array has to come from the process that has the minimum distance element for the corresponding node. To work around this, we defined a data structure that stores the distance and the parent, and a compare function that compares two elements of this data structure based on the distance value. We then deploy the Allreduce() call on our customized data structure with our customized compare function. This will reduce all local arrays of distance-parent pairs into a global one that stores the minimum distance and the corresponding parent node for each node. Still, one concern for this method is the possible overhead generated because our data structure and function are not as optimized as the API defined ones are to achieve the best performance.

3.7 Bellman Ford Algorithm with Early Termination

One optimization technique we deployed in our implementations is an early stopping criteria. Because the Bellman Ford algorithm updates shortest distances by repeatedly iterating over all edges multiple times, if the distance array is not updated during an iteration, then we can confirm that the current distance array is the shortest distance array. Therefore, we introduce a done flag to indicate whether any updates were made by this thread during this iteration. At the end of each iteration, if all done flags are still true, meaning no updates were made, the algorithm terminates early.

4 RESULTS

4.1 Evaluation

We evaluate our sequential and parallelized shortest paths algorithms on the dataset provided in [5]. Since there were no weighted graphs, we generate our own test data by extracting from the PA road network graph and randomly generating weights for each edge. We also made sure the extracted graph is connected by adding edges between nodes with adjacent labels. This will make sure the computations performed grows consistently with the number of nodes in each test graph.

We measure the compute time for each implementation on graphs of size 100, 1000, 10000, 30000, 50000, and 70000 nodes and record the total compute time in seconds in Figure 5. Since displaying all experimental results will take significantly more space, for parallelized versions, we only display here the execution times of thread counts that show improvement in performance from the previous thread counts. Later when discussing overall trends and specific cases, we will use graphs for better visualization.

	100	1000	10000	30000	50000	70000
Dij Seq	0.000035	0.001116	0.124092	1.049662	3.272296	6.968730
Dij OpenMP 4	0.000768	0.005897	0.082828	0.419291	1.060356	2.196711
Dij OpenMP 8	0.000970	0.008895	0.111924	0.392330	0.793979	1.556058
Dij MPI 4	0.001434	0.002780	0.042123	0.304660	0.920431	1.870061
Dij MPI 8	0.001957	0.004026	0.034584	0.186187	0.474450	0.876178
Dij MPI 16	0.003236	0.004062	0.038404	0.150001	0.313979	0.546564
Dij MPI 64	0.005280	0.009895	0.051327	0.149658	0.258482	0.396511
Bell Seq	0.000044	0.003513	0.384447	3.320095	9.320354	17.986922
Bell OpenMP 4	0.000429	0.004002	0.122059	0.916564	2.615442	4.945052
Bell OpenMP 8	0.000667	0.004372	0.083080	0.622684	1.350126	2.539595
Bell OpenMP 16	0.001160	0.005904	0.081985	0.374742	0.850444	1.564601
Bell OpenMP 64	0.110504	0.090854	0.167659	0.461632	0.980443	1.521513
Bell MPI 4	0.001133	0.011458	0.753334	6.335346	17.073723	34.224464
Bell MPI 8	0.001487	0.015885	0.949913	8.092773	21.207064	39.562006

Figure 5: Computation Time for all algorithms on all input graphs (in seconds)

4.2 Problem Size

Since we are targeting graph algorithms, the size of the problem(graph) can play significant roles in computation time. For each input graph size, we plot the speedup of each parallelized implementation against the corresponding baseline model in Figure 6. As expected, as the problem size grows, most of the parallelized implementations' performance also improve. More specifically, our parallelized implementations with 8 threads show positive speedup for input size of greater than 1000 nodes. This is because with smaller work loads, the benefit of parallelism is amortized by the cost of synchronization and initialization overhead.

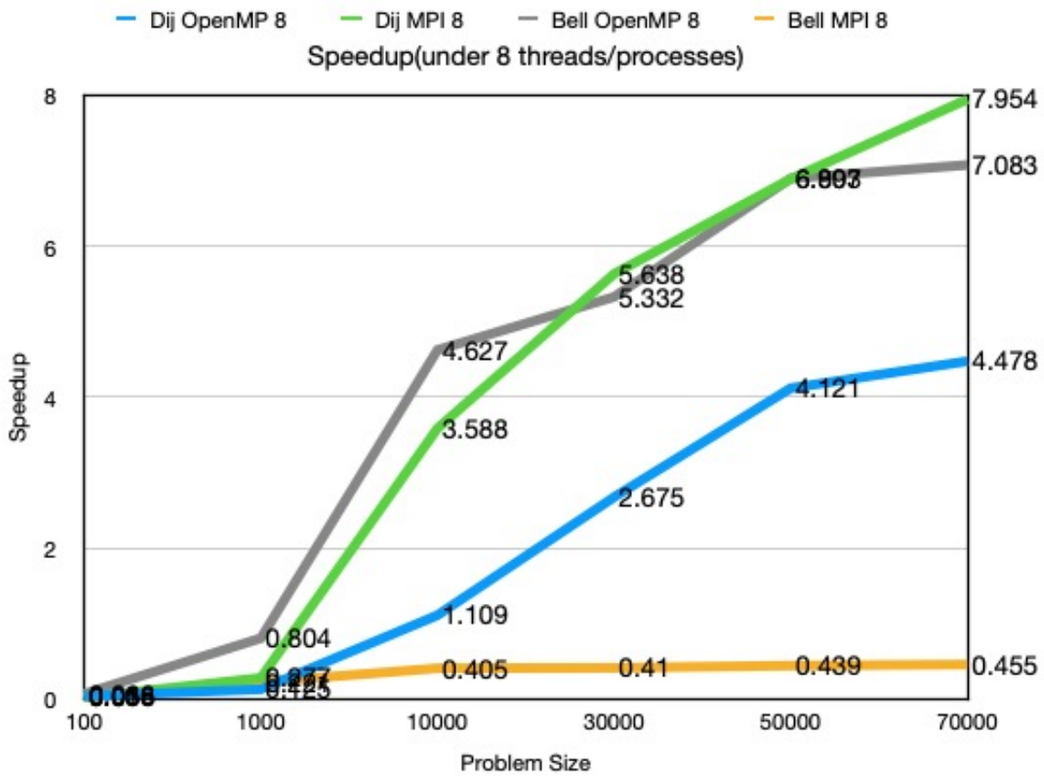

Figure 6: Speedup vs. Problem Size

4.3 Parallel Programming Models and Algorithms

4.3.1 OpenMP Dijkstra vs. Bellman Ford

The relative speedups for each algorithm vs. thread counts are shown in Figure 7. Under a shared-address space model implemented using OpenMP, the Bellman Ford algorithm showed better speedup from the baseline Bellman Ford model than Dijkstra's Algorithm. This is because our Dijkstra's OpenMP implementation required more synchronization to ensure the threads working together arrive at a correct solution when they work in a shared address. For example, at the beginning of each iteration, all threads must finish computing their local minimum-distance node before one of them proceeds to find the global minimum-distance node. Only after the global minimum has been computed will all worker threads proceed to updating neighbors. All threads must be synchronized with a barrier at the end of each iteration before any one can continue to the next round to find local minimum again with the updated shared distance array.

Dijkstra's OpenMP implementation also reaches peak speedup earlier at n=8. Because the work load distribution depends largely on the number of neighbors for the selected node, it is likely that for not strongly connected graphs where each node has only a few neighbors, Dijkstra's Algorithm with too much threads generate more overhead than the benefit in performance.

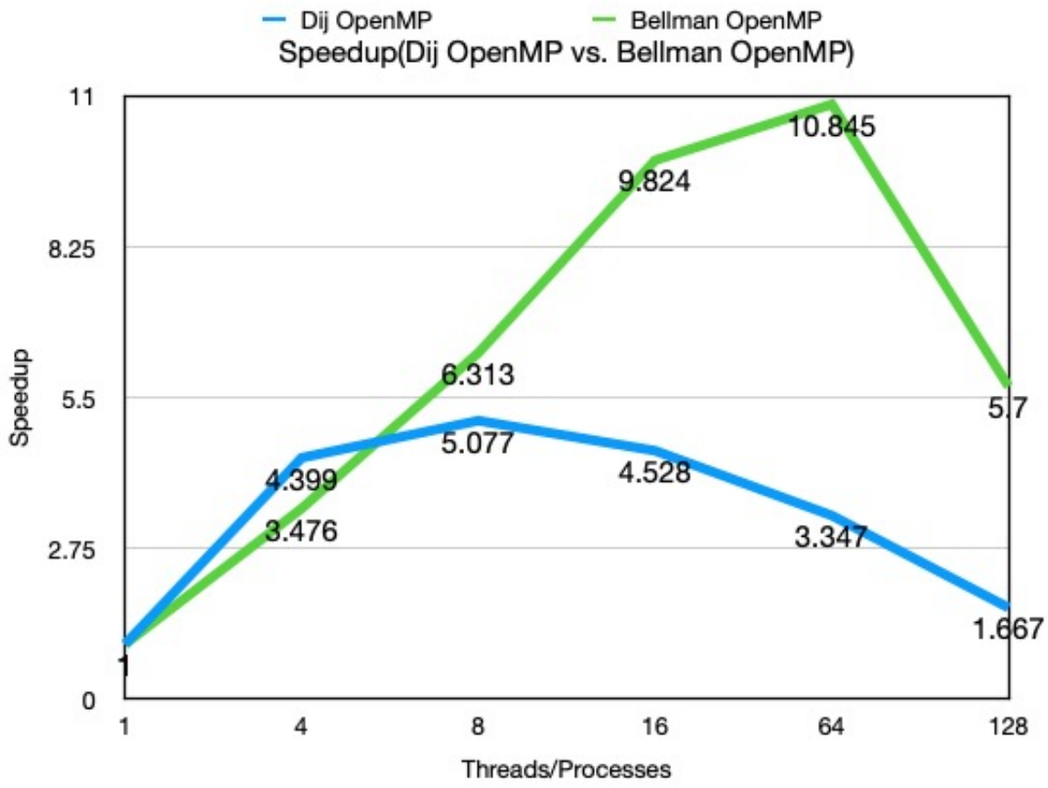

Figure 7: Speedup vs. Processor Count (OpenMP)

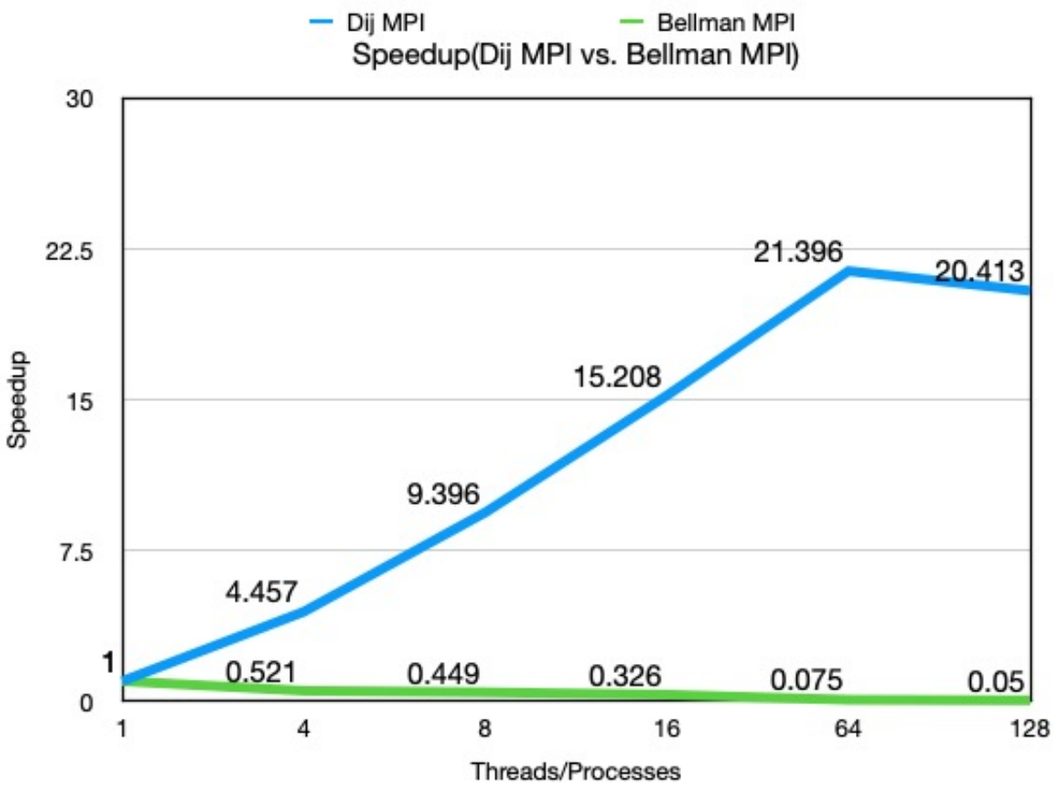

Figure 8: Speedup vs. Processor Count (MPI)

4.3.2 MPI Dijkstra vs. Bellman Ford

With the message-passing model MPI, Dijkstra's algorithm shows significantly better performance speedup than Bellman Ford's Algorithm. We suspected that this gap was results from the MPI Allreduce() calls made by the two implementation. After profiling our code, we confirmed that this was indeed a source of difference in compute time. The Bellman Ford MPI implementation, because the Allreduce() call uses a non-optimized, customized function to transfer V distance-parent pairs in the local array, it takes more time to complete. Meanwhile, our MPI Dijkstra's algorithm calls Allreduce() on only one integers pair(distance-node) for each process per iteration. Therefore, our MPI Bellman Ford implementation is bottle-necked by the fact that the local distance and parent pairs must be communicated and combined to find the correct values at the end of each iteration. Specifically, our profiling shows that the Allreduce() calls are responsibled for approximately 90% of the total compute time.

4.4 Bellman Ford Early Termination

Lastly, we also compare the speedups of Bellman Ford algorithms with and without early termination. We did not include the Bellman Ford algorithm with early termination in previous analyses because the performance of this optimized algorithm can fluctuate depending how densely connected the graph is and is thus not an appropriate choice for bench-marking and comparing with other algorithms. Still, with our sparsely connected input graphs, it can be seen below that Bellman Ford algorithm with early termination outperforms the one without early termination even in the sequential case.

Bellman (With Break)	OpenMP	Speed up	MPI	Speed up
1	0.080683	1	0.080683	1
4	0.036230	2.22696660226332	0.141608	0.569763007739676
8	0.022314	3.61580173881868	0.184681	0.436877643071025
16	0.016589	4.86364458376032	0.279529	0.288639103635043
64	0.010332	7.80903987611305	0.823588	0.097965244758301
128	0.853920	0.094485431890575	1.632813	0.049413496830316
Bellman	OpenMP	Speed up	MPI	Speed up
1	18.297137	1	18.297137	1
4	5.263214	3.47641897137377	35.131267	0.520822007358858
8	2.898375	6.31289498425842	40.761074	0.448887509686325
16	1.862575	9.82357059447271	56.158099	0.325814750246443
64	1.687196	10.8447015047452	245.404149	0.074559199893560
128	3.210036	5.69997875413235	367.516283	0.049785922002264

Figure 9: Bellman Ford Algorithm Speedup with and without Early Termination

5 REFERENCES

References

[1] *Adjacency List and Adjacency Matrix*. URL: https://bournetocode.com/projects/AQA_A_Theory/pages/graph.html.

[2] *Bellman Ford algorithm in Python*. URL: https://sites.google.com/site/rajboston1951/shortest-paths/3.

[3] *Dijkstra's algorithm in Python*. URL: https://www.stechies.com/dijkstras-algorithm-python/.

[4] *Graph*. URL: https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/.

[5] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

6 WORK DISTRIBUTION

Xinqi Wang (50%)

1. Sequential Dijkstra's and Bellman Ford algorithms
2. OpenMP parallelized Dijkstra with priority queue
3. OpenMP parallelized Bellman Ford
4. MPI parallelized Bellman Ford
5. Initial functional analysis on GHC
6. Final report
7. Final Video Recording

Yuou Lei (50%)

1. Evaluation dataset generator
2. OpenMP Dijkstra without priority queue
3. MPI parallelized Dijkstra
4. Final performance analysis on PSC
5. Final report
6. Presentation Slides
7. Final Video Recording

7 RECORD

Link: https://github.com/leiyuou/15618-Final-Project/blob/main/Presentation%26Record/Record_Xinqi