All-Pairs Shortest Path: Parallizing The Floyd-Warshall Algorithm

Alice McCollum
Alvin Tsang

CMPT 431 Spring 2024
Github Repo: https://github.com/Alvicorn/CMPT_431_Final_Project

Introduction

The All-Pairs Shortest Path (APSP) algorithm is an abstraction for many real-world problems, thus many approaches exist. In general, APSP computes the shortest path between all pairs of vertices in a weighted graph. Following the Floyd-Warshall algorithm, the shortest path distances between pairs of vertices can be computed by iteratively updating a matrix of shortest paths.

The algorithm begins by initializing the distance matrix with direct edge weights between vertices. Then, it iterates through all vertices and considers each vertex as a potential intermediate node in the shortest path. For each pair of vertices, it checks if going through the intermediate vertex leads to a shorter path than the current path. If so, it updates the shortest path distance. This process continues until all vertices have been considered as intermediate nodes. The result is a distance matrix that holds the shortest path distances between all pairs of vertices.

The serial implementation of the Floyd-Warshall algorithm has a time complexity of O(V^3), where V is the number of vertices in the graph. This paper proposes a multi-threaded and multi-processing solution to reduce the time complexity of APSP.

Background

The APSP problem has multiple proven solutions. Johnson-Dijkstra algorithm uses the Dijkstra algorithm and the Bellman-Ford algorithm. This approach runs slightly faster on a sparse graph, but slower otherwise. Another solution is Seidel's algorithm, which only works on undirected graphs). Because Floyd-Warshall gives an acceptable performance in all use cases, we chose it as a more general approach.

The Floyd-Warshall algorithm also lends itself well to parallelization, as it primarily relies on a triply-nested loop, each of which runs V times. We can speed up the overall runtime by splitting the workload of one of these loops. In extremely performance-sensitive settings, there is a specialized variation of the algorithm known as the "blocked Floyd-Warshall algorithm" [1]. This variation performs the calculations in a different order, which minimizes the number of cache evictions the CPU must perform, increasing the performance further. However, to take advantage of this, we must know specifics about the CPU being used, most notably its cache size. Because we can't know that in this case, we opted for the more general approach.

Implementation Details

Serial Implementation

The Floyd-Warshall algorithm can be described with the following pseudocode:

```
// read as an input
Graph graph = Graph();

// array of edge weights initialized to ∞
Matrix min_weights = Matrix(graph.vertex_count());

for (Edge edge: graph.all_edges()) {
  int in_vertex = edge.in_vertex();
  int out_vertex = edge.out_vertex();
  min_weights[in_vertex][out_vertex] = edge.weight();
}

for (int v = 0; v < graph.vertex_count(); v++) {
  min_weights[v][v] = 0;
}

for (int k = 0; k < graph.vertex_count(); k++) {
  for (int i = 0; i < graph.vertex_count(); i++) {
    for (int j = 0; j < graph.vertex_count(); j++) {
      int intermediate_weight = min_weights[i][k]
                                + min_weights[k][j];
      if (intermediate_weight < min_weights[i][j] {
        minimum_weight[i][j] = intermediate_weights;
      }
    }
  }
}
```

Figure 1: Serial implementation of Floyd-Warshall algorithm. Adapted from an example found on Wikipedia [2]

The Graph object is composed of the input and consists of unique vertices and directed, weighted edges. Each vertex can have zero or more ingress/egress edges.

The Matrix object is a two-dimensional square array with the same dimensions as the number of vertices from the Graph object. For the Floyd-Warshall algorithm, the row index

indicates the input vertex, the column index indicates the output vertex and the value at the position `Matrix[row_index][column_index]` is the weight of the directed edge between the input vertex and output vertex. Initially, each position in the matrix is initialized to ∞. We used ∞ to indicate two vertices that do not share an edge.

The first for-loop updates the Matrix using the initial Graph. If an edge exists between two vertices, update the Matrix value at position `Matrix[input_vertex][output_vertex]`. The second for-loop updates all self-loops to weight 0.

The final for-loop minimizes the overall weight to go from one vertex to another. This is achieved by utilizing the transitive property. As per Figure 1, the *i-indexed* for-loop represents the input vertex, the *j-indexed* for-loop represents the output vertex and the *k-indexed* for-loop represents an intermediate vertex such that *vertex i* has a directed edge to *vertex k* and *vertex k* has a directed edge to *vertex j (i → k → j)*. If the sum of weights for *(i → k → j)* is less than the current weight, `Matrix[i][j]` will be updated to store the smaller weight.

For an input graph with V vertices, the serial implementation has the time complexity of $O(V^3)$ and the space complexity of $O(V^2)$.

Parallel Implementation

Before parallelizing the Floyd-Warshall algorithm, the computationally expensive areas need to be identified. The final for-loop in the algorithm is the most expensive, being triply-nested. The proposed parallel implementation will use barriers to assist in synchronization across multiple threads.

```
// read as an input
Graph graph = Graph(); // read as an input

// arrays of minimum weights initialized to ∞
Matrix current_min_weights = Matrix(graph.node_count());
Matrix previous_min_weigths = Matrix(graph.node_count());

for (Edge edge: graph.all_edges()) {
  int in_vertex = edge.in_vertex();
  int out_vertex = edge.out_vertex();
  current_min_distances[in_vertex][out_vertex] = edge.weight();
  previous_min_distances[in_vertex][out_vertex] = edge.weight();
```

```
}

for (int v = 0; v < graph.vertex_count(); v++) {
  current_min_weights[v][v] = 0;
  previous_min_weights[v][v] = 0;
}

for (int k = 0; k < graph.vertex_count(); k++) {
  for each thread in T {
    for each i allocated to the thread {
      for (int j = 0; j < graph.vertex_count(); j++) {
        // self-loop is always 0 and k can not be a boundary
        if  (i == j || i == k || j == k) {
          continue;
        } else if (current_min_weights[i][j] >
              prev_min_weights[i][k] + prev_min_weights[k][j]) {
           curr_min_weights[i][j] = prev_min_weights[i][k]
                                  + prev_min_weights[k][j]
       }
     }
  }

  // wait for all of curr_min_weights to be updated
  barrier()

  // copy curr_min_weights to prev_min_weights
  copy(prev_min_weights, curr_min_weights)

  // all threads wait for prev_min_weights to be updated
  barrier()
}
```

Figure 2: Parallel implementation of Figure 1 using barriers

Similar to the serial implementation, a graph and two Matrix objects that hold the weight of all edges are declared at the beginning of the algorithm. The two Matrix objects will be a shared data structure available to all the threads. `curr_min_weights` holds the weights calculated in the current k-iteration and utilizes values from `prev_min_weights` which comes from the (k-1)-iteration.

Parallelization occurs at the *i-indexed* for-loop, where the i-indexed for-loop represents the input vertices. Each thread will be allocated a certain number of input vertices (coarse-grain granularity) and be responsible for computing the shortest path between all other vertices. After computing and updating the smallest weights for iteration k, the thread will wait at the barrier and allow all other threads to catch up. Once all threads have finished their work on k, one designated thread will update `prev_min_weights` to be the same as `curr_min_weights` to prepare for iteration k+1.

For an input graph with V vertices, the parallel implementation has the time complexity of $O(V^2)$ + (overhead) and the space complexity of $O(V^2)$. However, a sufficiently small or sparse graph may run slower when parallelized, due to that communication and synchronization overhead.

Distributed Implementation

In the distributed version, the workload is divided at *k*, the outer-most loop, rather than *i* or *j*, the inner one. Initially, when dividing the work like this, the algorithm will get a very inaccurate result. However, with a map-reduce function to send the best results of each function to the others, and repeat the process a certain number of times. If there is 1 process, there is no need to repeat it, as *k* was not divided, and the result is very similar to the serial implementation. If there are 2 processes, the computation will need to be repeated once more. This gives results identical to the serial implementation, as $i * j * k = 2(i * j * (k/2))$. If there are 3 processes, the computation will be repeated twice, and this again gives identical results to the serial implementation by similar math. If there are 4 or more processes, however, continue to only run the algorithm 3 times, and we finally begin to see some amount of speedup.

In pseudocode, this implementation looks like:

```
let curr_dist be a |V| × |V| array of minimum distances initialized
to ∞ (infinity)
let prev_dist be a |V| × |V| array of minimum distances initialized
to ∞ (infinity)
let P be the number of processes
let Pid be this process's ID number

for (u=0; u < |V|; u++} {
  for (v=0; v < |V|; v++) {
    If (u == v) { // self-loop
      curr_dist[u][v] = 0
    } else if (u has shares an edge with v) {
```

```
      curr_dist[u][v] = edge_weight(u, v)
    }
  }
}

step_size = |V| / P
start = step_size * Pid
end = start + step_size

for (i = 0; i < required_iterations; i++){
  for (k=start; k < end; k++) {
    for (i=0; i<|V|; i++) {
      for (j=0; j < |V|; j++) {
        if  (i == j || i == k || j == k) { // self-loop is always 0
          continue;
        } else if (dist[i][j] > prev_dist[i][k] + prev_dist[k][j]) {
           dist[i][j] = prev_dist[i][k] + prev_dist[k][j]
         }
      }
    }
  }

  // wait for all of dist to be updated
  barrier()
}
```

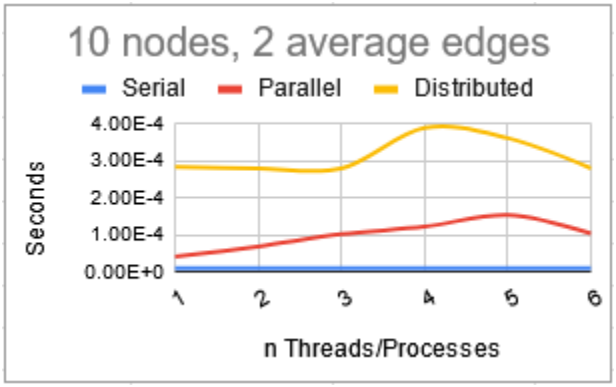
Figure 3: Distributed implementation of Floyd-Warshall using MPI

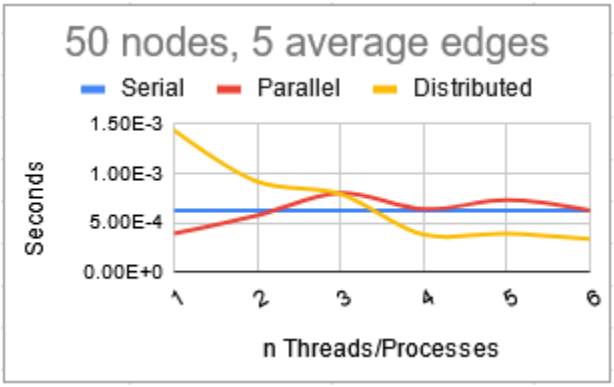
This implementation reduces the amount of communication between the processes. If the workload were distributed across many computers on a network, the communication overhead costs could add up very quickly if the work was distributed across *i* or *j*. While this method is less than ideal for a small number of processes, and has worse results than a multi-threaded solution for a moderately sized problem, with a sufficiently large number of processes and once accounting for networking overhead, this implementation may prove to be the better option in some cases.

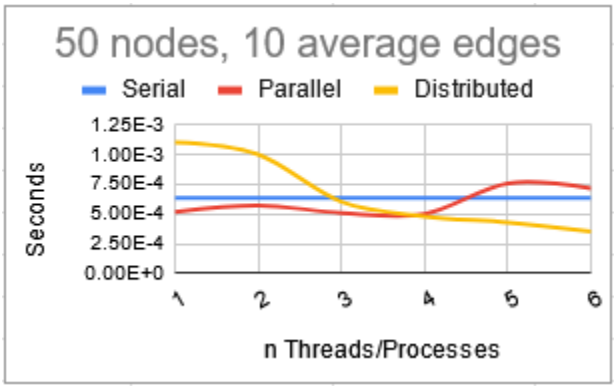
Evaluation

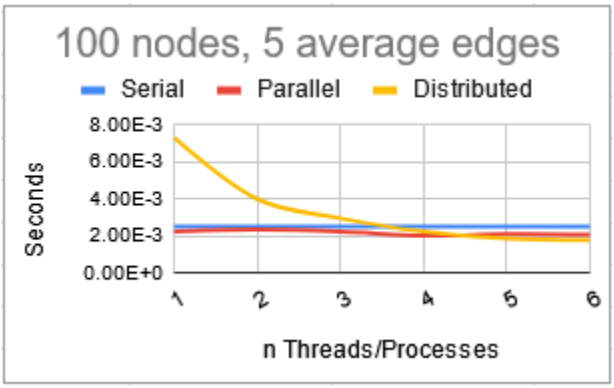
The three configurations (serial, parallel, and distributed) were tested on a variety of different graphs, each with a different number of nodes and edges. The parallel and distributed

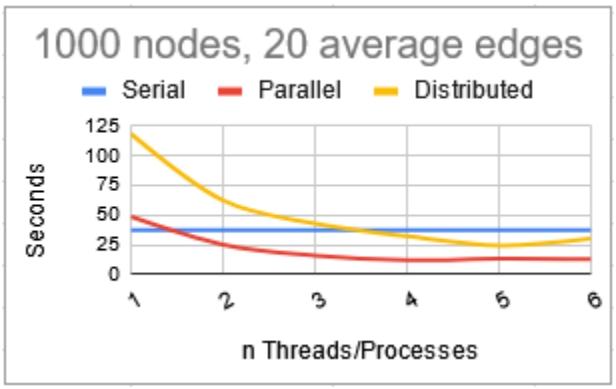
variants were tested with up to 6 threads/processes, on the provided Slurm cluster. Due to Slurm job time limits, only graphs with up to 1000 vertices were tested, and the speeds beyond this point are uncertain. Each test three times, and the average results of those tests were used to generate the graphs below. Only one graph was tested, though each implementation used the same graph for a given size.











Conclusions

The above graphs make it clear that a sufficiently large graph, as well as a sufficient level of parallelization, is needed to see any appreciable speed increase. On a small graph, the serial

version performs better. This is expected, as the serial version would not need to do additional overhead work that turns out to be disproportionately time-consuming in small graphs.

Because the distributed version is run entirely within SFU's network, it's hard to say if these results would be replicated if it were distributed across a global network. In that case, the effects of network overhead would be more significant. However, the technique used in that method is designed to mitigate the effects of communication overhead, and in large graphs (such as the 1000-node graph) the majority of the time is spent on computing, rather than communication. Thus, we expect the results to be very similar for any sufficiently large graph, assuming there are no issues with the network causing major delays.

We learned multiple ways to approach the all-pairs-shortest-path problem, based on the method of parallelization we plan to use. We also saw that, as expected, parallelizing problems does not always lead to faster results, especially with sufficiently small problems.

References

[1] A. A. Prihozhy, “Simulation of direct mapped, K-way and fully associative cache on all pairs shortest paths algorithms,” *«System analysis and applied information science»*, no. 4, pp. 10–18, Dec. 2019. doi:10.21122/2309-4923-2019-4-10-18

[2] “Floyd–Warshall algorithm,” Wikipedia, https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm#Pseudocode (accessed Apr. 14, 2024).