Concurrent Graph Algorithms

Jacob Steinebronn Daniel West William Quiroga Alex Rutledge

***Abstract*—This project does a shallow dive into concurrent implementation in c++ for 4 graph topics: All-Pairs-Shortest-Path, the Disjoint Set Union data structure, Minimum-Spanning-Tree, and Single-Source-Shortest-Path. Implementations take both correctness and running-time into account, and are benchmarked on a 12-core, 24-thread machine.**

***Index Terms*—graph, algorithm, dijkstra, floyd-warshall, shortest-path**

I. Introduction

In this paper, we will perform a wide analysis on several high-level graph topics: the All-Pairs-Shortest-Path problem, the Minimum Spanning Tree problem, the Shortest-Distance problem with negative edges, and the Disjoint Set Union data structure (which is often used in graph algorithms such as connectivity or Kruskal's MST algorithm). In particular, the implementations of these algorithms in parallel will be analyzed in detail regarding the best practices for different use-cases or input regimes.

II. All-Pairs Shortest-Path

A. *Problem Definition*

Given a weighted, undirected graph G, $\forall u, v \in G$, find $DIST_{u,v}$ equal to the shortest path in G from u to v. In general, the edge weights can be any real number, but for our purposes it will be convenient to restrict edge weights to non-negative integers which do not exceed 10^9. This will be done for two main reasons:

- To restrict the maximum possible answer into an integer which can be stored conveniently in most modern operating systems (a 64-bit signed integer will suffice)
- To restrict the graph such that path lengths are defined. With negative edge weights, a path could simply traverse this negative edge back and forth infinitely many times, so any fully relaxed path in a graph with at least one negative edge weight will have all paths of length $-\infty$

These two items let us prove a convenient upper-bound on the length of any one shortest path as follows: Let w_{max} be the largest possible edge weight. Any shortest path from u to v, $P = p_0, p_1, ...p_{n-1}, p_k \mid p_0 = u, p_k = v$ will be simple; that is, there will not exist any node p_i which appears in P twice. We can show this by contradiction: If there were some node that appeared twice in P, let the first and last occurrence of this duplicate node be p_i and p_j, then $P' = p_0, p_1...p_i, p_{j+1}, p_{j+2}...p_k$ would be shorter. That is, we could "cut out" the cycle between p_i and p_j. Thus, we know that for any shortest path, every node in G will appear at most once, so the maximum value for any shortest path in a graph of n nodes is $n * w_{max}$.

B. *Sequential Implementations*

We will examine two implementations for the All-Pairs-Shortest-Path problem: the Floyd-Warshall algorithm, and All-Sources Dijkstra's algorithm. While these algorithms are well-known, we will enumerate the specific implementations in c++ we'll be building off in the future.

```cpp
typedef pair<ll, int> st;
typedef priority_queue<st, vector<st>, greater<st>>
    dijPQ;

// Fills in every column of res[s]; that is,
// shortest distance from s to every node.
void dijkstra(int s) {
   for(int i = 0; i < n; i++) res[s][i] = oo;
   res[s][s] = 0;
   dijPQ pq;
   pq.emplace(0, s);

   while(pq.size()) {
      auto [d, u] = pq.top(); pq.pop();
      if(d > res[s][u]) continue;
      for(int v = 0; v < n; v++) {
         if(v != u && res[s][v] > d + adj[u][v]) {
            res[s][v] = d + adj[u][v];
            pq.emplace(res[s][v], v);
         }
      }
   }
}

for(int s = 0; s < n; s++)
   dijkstra(s);
```

Listing 1. Sequential Dijkstra's on Adjacency Matrix

```cpp
for(int k = 0; k < n; k++)
   for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++)
         // edge relaxation of i->k->j into i->j
         if(adj[i][k] + adj[k][j] < adj[i][j])
            adj[i][j] = adj[i][k] + adj[k][j];
```

Listing 2. Sequential Floyd-Warshall's on Adjacency Matrix

These are the most basic implementations of both Dijkstra's and Floyd-Warshall's. Note that the Dijkstra's implementation has been adapted to write its output into the pre-defined matrix *res* and called on every start point (Since Dijkstra's algorithm calculates Single-source Shortest-Path to all nodes). As a sanity-check, we run these implementations on the CSES (Code Submission Evaluation System) set "Shortest Routes II", and they of course pass. The Dijkstra's implementation is $\mathcal{O}(n^3 \log n)$, and the Floyd's implementation is $\mathcal{O}(n^3)$. Sequentially, Floyd's is faster in every case due to its superior time-complexity and extremely low overhead; however, when both algorithms for APSP are adapted to be run non-sequentially, we'll see that this is no longer *always* true.

C. Floyd-Warshall's Algorithm in Parallel

Perhaps the most straightforward attempt at parallelizing the Floyd-Warshall algorithm would be to split up the outermost for loop (the "K loop"), and dividing the execution of this outer loop (and of course, the corresponding execution of the inner loops) evenly among the threads. Such an implementation would look as follows:

```
auto threadEx = [&](int threadNum) -> void {
   for(int k = threadNum; k < n; k += NUM_THREADS)
      for(int i = 0; i < n; i++)
         for(int j = 0; j < n; j++)
            if(adj[i][k] + adj[k][j] < adj[i][j])
               adj[i][j] = adj[i][k] + adj[k][j];
};

vector<thread*> threads;
for(int i = 0; i < NUM_THREADS; i++)
   threads.push_back(new thread(threadEx, i));
```

Listing 3. Incorrect attempt at parallelizing Floyd's

It isn't too hard to see why this is wrong: semantically, the "K loop" is iterating over all paths of length 2 that pass through one node called the "pivot", and performing edge relaxations over that pivot. However, when multiple pivots are being simultaneously processed, a situation could arise where a node which is a pivot in one thread has an edge relaxation happen over it, which disturbs the value of any future relaxations with that pivot by introducing a circular dependency in the relaxations. Indeed, it is not hard to generate graphs for which this implementation is correct; suffice it to say, this approach is incorrect.

The initial approach, while it was wrong, did give some hints as to a way to correctly parallelize Floyd-Warshall's algorithm. Instead of splitting up the outermost "K loop", we can split up the middle "I loop" instead for each k. This works correctly, since every relaxation over a single pivot operates independently. In fact, even without a proof of correctness for the Floyd-Warshall algorithm or knowing how it works, this can be observed simply from the code shown in Listing 2. All 5 of the array lookups and the array update are only operating on either row $adj[i]$ or $adj[k]$, so the only issue any thread would have performing an entire pass through the "I loop" would be if an update to $adj[k]$ was an update on $adj[i]$; that is, if for some relaxation inside the "K loop", node k had an edge relaxation. However, since k is the pivot, any relaxation would do nothing, since $adj[k][k] = 0$. Thus, every thread's work is independent of every other thread. So an algorithm to run Floyd's in parallel would be to iterate the outer loop sequentially, and then spawn new threads for every outer iteration to complete every middle and inner loop in parallel:

```
auto threadEx = [&](int tnum, int k) -> void {
   for(int i = tnum; i < n; i += NUM_THREADS) {
      for(int j = 0; j < n; j++)
         if(adj[i][k] + adj[k][j] < adj[i][j])
            adj[i][j] = adj[i][k] + adj[k][j];
   }
};

for(int k = 0; k < n; k++){
   vector<thread*> threads;
   for(int i = 0; i < NUM_THREADS; i++)
      threads.push_back(new thread(threadEx, i, k));
   for(auto cur : threads) cur->join();
}
```

Listing 4. Correct attempt at parallelizing Floyd's

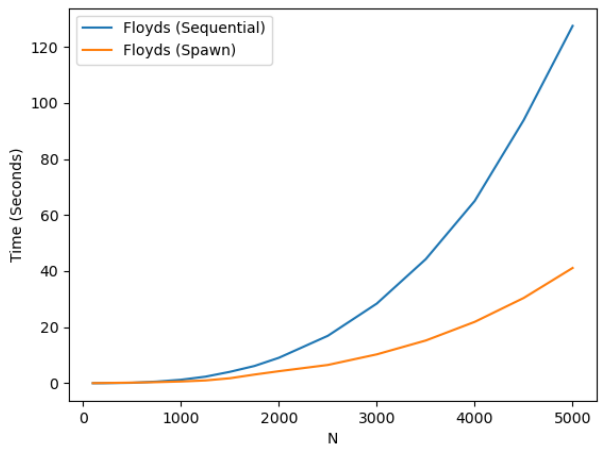

Fig. 1. Parallel Floyd's-Spawn with 20 threads versus sequential Floyd's

Indeed, generating and testing this implementation (Which we'll call "Floyd's Spawn" since it spawns new worker threads for each iteration) against random graphs, and comparing the output to the Dijkstra's and Floyd's implementations shown in Figures 1 and 2 shown to be correct, this works as intended.

We can observe in figure 1 a roughly 3x speedup with 20 threads with this method. We will go in greater detail with the effects of thread count on running time later.

The main potential problem with this solution is the overhead of repeatedly creating and destroying threads for every iteration of the "K loop". We can fix this problem by creating the worker threads once and re-using them, but we will have to make sure that one worker thread does not move ahead to the next value of k before every other thread has finished with the same thread. We can accomplish this by having each thread increment a counter when it finishes for some k, and then wait (by locking a mutex). Then, when this counter reaches the number of threads, every thread can be unlocked and continue on to the next value of k. This process will repeat for n iterations. We will call this approach "Floyd's-Sync", since all the threads are "syncing" before moving on together to the next value of k (after which they might fall out of sync until

```cpp
int k = 0, numFinished = 0;
mutex finished_lock;

// Create mutexes, and lock them all to start.
vector<mutex*> waits;
for(int i = 0; i < NUM_THREADS; i++)
   waits.push_back(new mutex());
for(auto wait : waits)
   wait->lock();

auto threadEx = [&](int tnum) -> void {
   while(k < n) {
       // Complete the "I loop"s for this value of k
      for(int i = tnum; i < n; i += num_threads) {
         for(int j = 0; j < n; j++)
            if(adj[i][k] + adj[k][j] < adj[i][j])
               adj[i][j] = adj[i][k] + adj[k][j];
      }

      // We only allow one thread to "finish" at the
     same time, enforced with finished_lock.
      finished_lock.lock();
      if(numFinished == NUM_THREADS - 1) {
          // This thread is the last one to finish,
    so it's responsible for setting up all the
    global variables for the next iteration

          // increment k and reset numFinished
         ++k, numFinished = 0;
         // unlock every thread's mutex
         for(auto wait : waits)
            wait->unlock();
         waits[tnum]->lock();
         finished_lock.unlock();
      } else {
         numFinished++;
         finished_lock.unlock();
         waits[tnum]->lock();
      }
   }
};

vector<thread*> threads;
for(int i = 0; i < NUM_THREADS; i++)
   threads.push_back(new thread(threadEx, i));
for(auto cur : threads) cur->join();
```

Listing 5. Floyd's-Sync implementation

finishing). The c++ implementation for this approach is shown in listing 5.

Again, running this implementation against the others which are working as intended, this implementation also produces correct output. Figure 2 shows Floyd's-Sync and Floyd's-Spawn run on the same inputs, and while the improvement is only marginal, Floyd's-Sync is consistently slightly faster than Floyd's-Spawn.

D. Dijkstra's Algorithm in Parallel

In the previous section, we analyzed different practices for computing the All-Pairs-Shortest-Path matrix in parallel using Floyd-Warshall's algorithm. The other algorithm we will be considering for computing this matrix will be All-Sources Dijkstra's algorithm; that is, we will run Dijkstra's from every node as a source. If we observe the implementation shown in listing 1, we will notice that, for any function call of

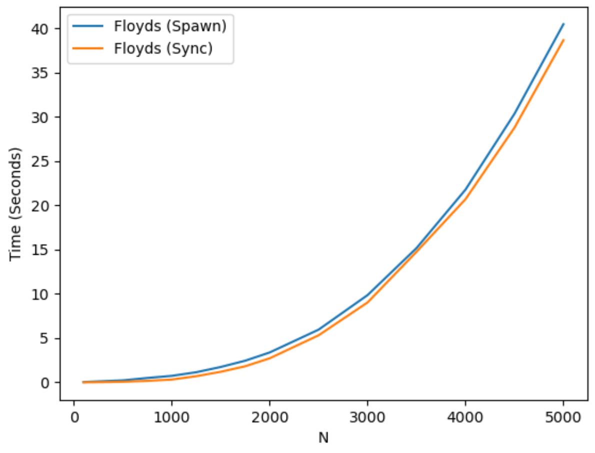

Fig. 2. Floyd's-Sync versus Floyd's-Spawn, both with 20 threads

```cpp
void dijkstra(int s) {
   for(int i = 0; i < n; i++) res[s][i] = oo;
   res[s][s] = 0;
   dijPQ pq;
   pq.emplace(0, s);

   while(pq.size()) {
      auto [d, u] = pq.top(); pq.pop();
      if(d > res[s][u]) continue;
      for(int v = 0; v < n; v++) {
         if(v != u && res[s][v] > d + adj[u][v]) {
            res[s][v] = d + adj[u][v];
            pq.emplace(res[s][v], v);
         }
      }
   }
}

auto threadEx = [&](int tnum) -> void {
   for(int i = tnum; i < n; i += num_threads)
    dijkstra(i);
};

vector<thread*> threads;
for(int i = 0; i < NUM_THREADS; i++)
   threads.push_back(new thread(threadEx, i));
for(auto th : threads) th->join();
```

Listing 6. Trivial Dijkstra's parallelization

$dijkstra$ on an arbitrary starting point s, the only changes that can be seen from outside the method are in the row $res[s]$. This suggests an implementation of All-Sources Dijkstra's in parallel: Call $dijkstra(i)$ on every node. As observed previously, we won't have any issues like we had in the previous section with Floyd's, as any updates are constrained to a single row in the result matrix, and function calls to $dijkstra$ are ambivalent to any changes made in any row except for the row of the starting node. Thus, we do not expect any issues, and can code a relatively straightforward parallelization shown in listing 6.

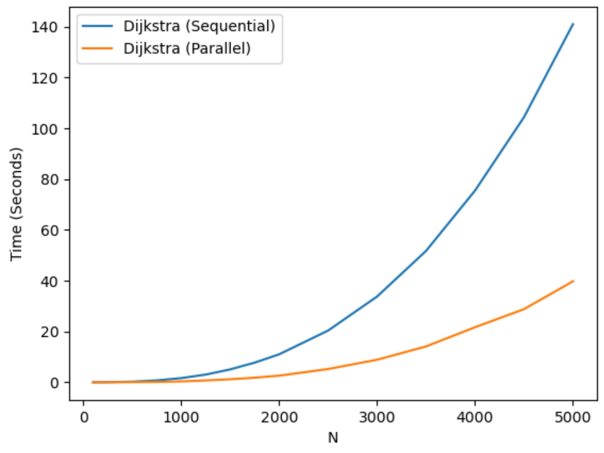

Fig. 3. Sequential vs Parallel Dijkstra's APSP with 20 threads

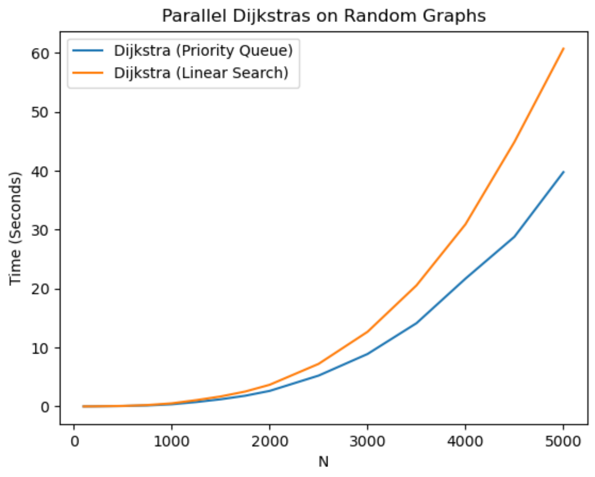

Fig. 4. Dijkstra's with Priority Queue vs Dijkstra's with Linear Search on randomly-generated graphs, both with 20 threads

```
void dijkstra(int s) {
   for(int i = 0; i < n; i++) res[s][i] = oo;
   res[s][s] = 0;

   for(int i = 0; i < n; i++){
      // Find lowest distance not-done node
      int u = -1;
      for(int v = 0; v < n; v++)
         if(!done[s][v])
            if(u == -1 || res[s][v] < res[s][u])
               u = v;
      done[s][u] = 1;
      // explore u's edges
      for(int v = 0; v < n; v++)
         if(res[s][v] > res[s][u] + adj[u][v])
            res[s][v] = res[s][u] + adj[u][v];
   }
}

auto threadEx = [&](int tnum) -> void {
   for(int i = tnum; i < n; i += num_threads)
    dijkstra(i);
};

vector<thread*> threads;
for(int i = 0; i < NUM_THREADS; i++)
   threads.push_back(new thread(threadEx, i));
for(auto th : threads) th->join();
```

Listing 7. Dijkstra's with Linear Search instead of Priority Queue label

E. Parallel Dijkstra's Algorithm Performance in the Worst Case

The above results, showing that parallel Dijkstra's with a priority queue for selecting states performs better than using a linear search when the input is a randomly-generated graph (Still within the restrictions outlined earlier). On randomly-generated graphs, we can expect the size of the priority queue to be quite small, since a large majority of the graphs would have edges which are far greater in weight than some shortest path which could circumvent that edge; thus, the number of new states which repeat on unexplored nodes would be low. In fact, the size of the priority queue can be observed to be $O(n)$ on random graphs. However, if we have an adversarial-generated graph, one where the particular implementation of priority-queue Dijkstra's is taken into account, we can get this priority queue to grow with the number of edges in the graph, which is $O(n^2)$.

Performing a rough analysis of the linear-search versus the priority-queue implementations will illuminate the running time in the average (random graph) versus worst(adversarial graph) case. The linear search implementation always takes exactly $2n^2$ per Dijkstra's call: For each of i iterations, do a single loop of size n to find the source, and then another loop of size n to perform relaxations. This is the same regardless of the graph itself because the edge weights do not affect the time it takes to update the res array or perform the first loop to find the source. However, for the priority-queue implementation, the edge weights strongly influence the running-time in that they influence the size of the priority queue used to find the source for each iteration. In a random graph, the size of the priority queue is expected to be roughly linear, since the expected number of edges in a path is very small. We perform n removals from the priority queue, each of which takes $O(log(n))$, and n insertions, each of which also takes $O(log(n))$. Thus, in the average case, the expected number of operations is roughly $n * log(n) + n * log(n) + n^2$ which is strongly dominated by n^2. However, in the worst case, the size of the priority queue is $O(n^2)$; thus, the number of insertions is n^2, each taking $log(n^2) = 2log(n)$, so the number of operations in the worst case for this implementation is roughly $2n^2 * log(n)$ which, compared to the worst-case for linear search Dijkstra's, is worse by an entire log factor, not to mention the overhead of actually maintaining the priority queue rather than the extremely low-overhead for loop.

We can improve upon this worst-case runtime in Dijkstra's by making an observation about the priority queue iteself:

```
void dijkstra(int s) {
   for(int i = 0; i < n; i++) res[s][i] = oo;
   set<pair<ll, int>> nxt;
   nxt.emplace(res[s][s] = 0, s);

   for(int i = 0; i < n; i++){
      auto [d, u] = *nxt.begin();
      nxt.erase(nxt.begin());
      // explore u's edges
      for(int v = 0; v < n; v++)
         if(res[s][v] > d + adj[u][v]) {
            nxt.erase(make_pair(res[s][v], v));
            res[s][v] =  d + adj[u][v];
            nxt.emplace(make_pair(res[s][v], v));
         }
   }
}
```

Listing 8. Dijkstra's with Ordered Set instead of Priority Queue label

There are many states inside the priority queue which are contributing to the runtime of adding/removing to the queue that aren't even useful. Consider a graph with 3 nodes u, v, w and a BFS-ordering u, v, w: node u will place nodes v and w into the priority queue, then node v might put node w into the queue again, so now node w is in the queue twice, even though only one of the states is actually the minimal distance. We can fix this by instead of maintaining a priority queue of states, we can maintain an ordered set of them. Then, when we want to update the distance of a state, we can remove the old entry in the set for that state, having found a better one, before inserting the new one. That way, even if every node in the graph can be found in the "queue" at one time, it will appear only once, so the size of the "queue" is bounded by n instead of n^2. We will still have a worst-case number of insertions being n^2, but at least we can speed up the insertions/removals by a factor of two. This can be observed in Fig. 5

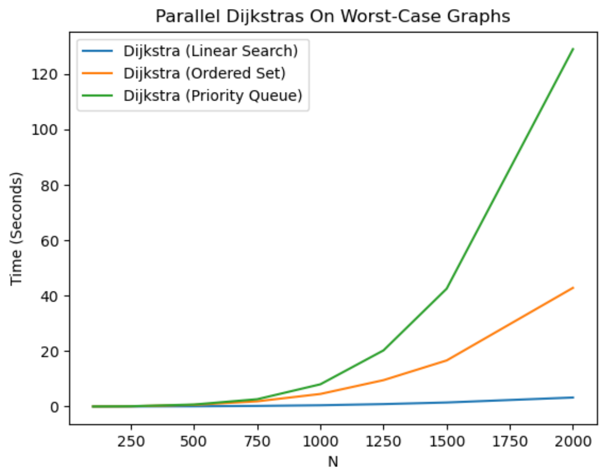

Fig. 5. Various Parallel Dijkstra's implementations on worst-case graphs

III. Minimum Spanning Tree

The Minimum Spanning Tree (MST) problem is a common architecture building problem that finds the minimum-weighted connected graph of a certain set of connected vertices [2]. The graph will contain V-1 edges where V is the total number of vertices in the graph. This problem can be found in many real life problems such as road construction, circuit design & development, and phone line planning. In the realm of computer science research the MST problem can also be used to approximate NP-Hard problems such as the traveling salesperson [1]. We will be looking at how to properly implement a solution concurrently and compare the runtimes to the sequential implementation.

A. *Prim's MST*

Prim's MST is a greedy approach to the MST problem where the concept is to maintain two sets of vertices: the set of vertices included in the tree, and the vertices yet to be added. As the tree is built, each edge between the two sets are considered and the smallest weighted edge is chosen to be a part of the tree. The minKey function takes in the two sets and returns the index of the minimum weighted edge that connects the two. The sequential implementation can be seen below:

```
void primMST(int** graph)
{
  for (int i = 0; i < num_nodes; i++)
    key[i] = INT_MAX, mstSet[i] = false;
  key[0] = 0;
  parent[0] = -1;

  for (int c = 0; c < num_nodes - 1; c++)
  {
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    for (int v = 0; v < num_nodes; v++)
      // Update the key only if graph[u][v] is
    smaller than key[v]
      if (graph[u][v] && mstSet[v] == false && graph
    [u][v] < key[v])
        parent[v] = u, key[v] = graph[u][v];
  }
}
```

Since the edges of the spanning tree need to be added sequentially, the algorithm contains only two highly parallelizable sections of code: the minKey function and the comparison of edge weights in the inner loop of the primMST function.

```
int minKey(int key[], bool mstSet[])
{
  int min = INT_MAX, min_index;
  for (int v = 0; v < num_nodes; v++)
    if (mstSet[v] == false && key[v] < min)
      min = key[v], min_index = v;
  return min_index;
}
```

B. Parallelization

The minKey function as shown above can be parallelized by dividing the for loop into subsections and storing the minimum key for a thread locally and comparing these minimums when the threads close.

```
int minKey(int key[], bool mstSet[])
{
   int min = INT_MAX, index, i;
#pragma omp parallel
    {
        int index_local = index, min_local = min;
#pragma omp for nowait
        for (i = 0; i < num_nodes; i++)
        {
            if (mstSet[i] == false && key[i] <
    min_local)
            {
                min_local = key[i];
                index_local = i;
            }
        }
        // Check if the local min is less than the
    shared min
#pragma omp critical
        {
            if (min_local < min)
            {
                min = min_local;
                index = index_local;
            }
        }
    }
    return index;
}
```

This implementation sustains the correctness of the minKey function in that it will always select the minimum key within the proper set of vertices, but because this is a parallel implementation it is possible that this function will produce a different minKey index than it's sequential counterpart. Therefore, the correctness of the sequential and parallel programs was checked by comparing the total weights of each spanning tree rather than checking which edges were included.

The second point of parallelization comes in the primMST function after the minKey function:

```
#pragma omp parallel for schedule(static)
    for (int v = 0; v < num_nodes; v++)
        // Update the key only if graph[u][v] is
    smaller than key[v]
        if (graph[u][v] && mstSet[v] == false &&
    graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
```

C. Test Data

Run-time testing was done on a 12-core CPU, and each test iteration was done with 10 total tests with incremental input sizes.

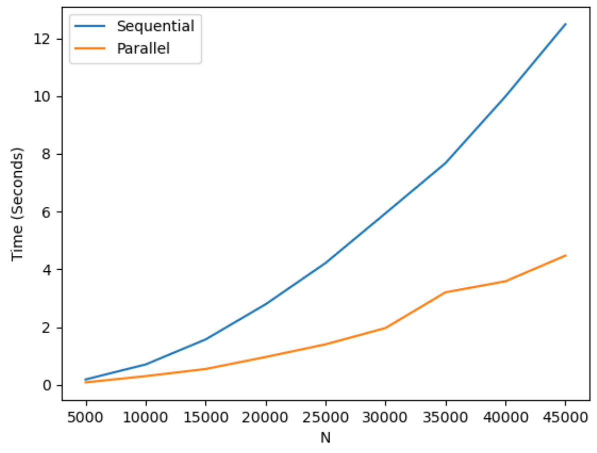

Fig. 6. Sequential Prim's vs. Parallel Prim's with 8 threads

Running the concurrent solution on a 12-core CPU with 8-threads can see that as input sizes grow, the parallel implementation can produce run-times more than 3 times faster than the sequential algorithm. As the data above indicates, as the number of threads reaches the number of cores the runtime improvements become inconsistent, as each shared-memory operation becomes more computationally expensive. Since testing data was limited by CPU cores, it is possible that even further runtime improvements can be observed on CPUs with 16 or more cores and utilizing more active threads.

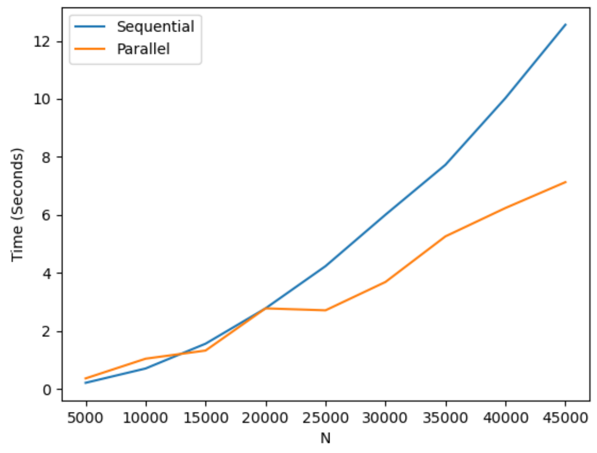

Fig. 7. Sequential Prim's vs. Parallel Prim's with 12 threads

IV. Single-Source Shortest Path with Negative Edge Weights

What this bit of the research hopes to address is the problem of finding shortest path distances in a directed weighted graph from a single source with the additional complication of allowing negative edge weights. We will also address the problem of finding negative sum cycles in such graphs. The two most known algorithms that are applicable to this problem are the Bellman-Ford algorithm and the Floyd-Warshall algorithm. We will explore the parallelization of these algorithms in relation to this problem and their performance.

A. The Bellman-Ford Algorithm

The Bellman-Ford algorithm is designed to find the shortest path from a single source in a general weighted graph G consisting of a vertex set V and an edge set E. The algorithm is also useful for finding negative cycles that are reachable from the source provided.

It relies on the principle that if all of the edges of of non-negative weight, then a shortest path from the source to some vertex will travel through at most V distinct vertices, and therefore at most V - 1 distinct edges. The algorithm works as follows: initially, a distance array will be set up to store the distance from the source to every other vertex. This distance array will be initially filled with an arbitrarily large value, signifying that we have not found a path from the source to this vertex. The source's distance will then be set to 0. Then, a certain number of iterations will be performed. In a single iteration, every edge will be considered in an arbitrary order, and an update for each edge will take place. For some edge from vertex u to vertex v with weight w, if there exists a path from the source to u that has been found (the distance to u is currently less than our arbitrarily large value), and if the distance from u plus the weight w is less than the currently stored distance to v (or if no path to v from the source has been discovered), then the distance to v will be updated. Then, since for any vertex some shortest path will consist of at most V - 1 edges, then the most number of iterations needed in the worst case is exactly V - 1. Some basic C++ code for this step of the algorithm is provided below:

```cpp
dist[sourceNode] = 0;
for(int i = 0; i < numNodes - 1; ++i){
   for(auto [u, v, w] : edges){
       if(dist[u] < INF && dist[u] + w < dist[v]){
           dist[v] = dist[u] + w;
       }
   }
}
```

The next step of the algorithm deals with handling negative sum cycles in the graph. If there is a negative sum cycle in the graph that is reachable from the source, then every node that exists in the cycle will have an indefinitely small shortest path to it. This is because if the cycle has negative sum to traverse the whole cycle, then for any shortest path of sum X, the cycle could be traversed exactly one more time to achieve a shortest path of sum less than X. This is true for any X, therefore the shortest path sum is undefined for these nodes. This proof expands to all nodes that are even reachable by some node on the negative cycle, even if the path from the cycle to the desired node is positive as an infinitely small sum is achievable in the cycle.

To detect this, only a single more iteration of the edge updating process needs to be done. If it is found that a vertex has a shorter distance now after this iteration, then it must be a part of a negative cycle reachable from the source. These nodes will be marked during this process. Then, a multi-source breadth first search algorithm will be performed with the set of sources as the initially marked nodes. The search will mark all nodes that are reachable from the initially marked nodes. At the end of this process, a node being marked signifies that its minimum sum path from the source to itself is infinitely small. An arbitrarily small value can be set in the distance array to show this. Note that some nodes can have negative shortest path distances without being infinitely negative. Some simple code to find these nodes is shown below:

```cpp
queue<int> q;
vector<bool> marked(numNodes);
for(auto [u, v, w] : edges){
    if(!marked[v] && dist[u] < INF && dist[u] + w <
    dist[v]){
        marked[v] = true;
        dist[v] = -INFINITY;
        q.push(v);
    }
}

while(!q.empty()){
    int u = q.front();
    q.pop();

    for(auto [v, w] : adjEdges[u]){
        if(!marked[v]){
            marked[v] = true;
            dist[v] = -INFINITY;
            q.push(v);
        }
    }
}
```

The overall run time for this complete algorithm is O(V * E). It can be seen that the major bottleneck of the algorithm is looping through each edge V - 1 times. The additional Breadth-First-Search at the end to find the special marked nodes is a run time of O(V + E), so the focus of parallelization on the algorithm will be directed towards this O(V * E) double for loop. It is imperative that all edges perform an update on our distance array for one iteration before any edges of the next iteration perform an update. A naive approach to a parallelization of these updates can therefore be as follows: a single parent thread will loop for each iteration of edge updates. Then, the edge updates will be divided evenly among the worker threads. each thread will loop over its selected section of edges, and prepare to make an update for each edge. For some edge from vertex u to vertex v with weight w, the thread will need to make sure while reading the distance table at u and writing to the distance table at vertex v that no other thread modifies these values. A simple fix to this problem is to create a lock for each vertex. When a thread goes to read/modify these values, it locks the two locks for these vertices, then performs the check and update, and then unlocks these locks. Some example C++ code is shown below:

```cpp
dist[sourceNode] = 0;

int numHandledByEachThread = numEdges / numThreads;

```

```
for(int iteration = 0; iteration < numNodes - 1; ++iteration){
    vector<thread*> threads(numThreads);

    for(int threadIdx = 0; threadIdx < numThreads; ++threadIdx){
        int startEdge = numHandledByEachThread * threadIdx;
        int endEdge = numHandledByEachThread * (threadIdx + 1);

        endEdge = min(endEdge, numEdges);

        threads[threadIdx] = new thread(updateEdges, startEdge, endEdge);
    }

    for(int threadIdx = 0; threadIdx < numThreads; ++threadIdx){
        threads[threadIdx]->join();
    }
}
```

```
void updateEdges(int startEdge, int endEdge){
    for(int edgeID = startEdge; edgeID < endEdge; ++edgeID){
        auto [u, v, w] = edges[edgeID];

        locks[u]->lock();
        locks[v]->lock();

        if(dist[u] < INF && dist[u] + w < dist[v]){
            dist[v] = dist[u] + w;
        }

        locks[u]->unlock();
        locks[v]->unlock();
    }
}
```

The first thing that can be noted about this implementation is that the locks will block the threads quite frequently. A good portion of the time, a thread will be waiting for a lock to unlock before it can make a check/update. This causes a significant loss of efficiency. Not only this, but this implementation also opens up the possibility of deadlock. Since each thread will attempt to lock two different threads sequentially, a situation could arise in which one thread takes a resource needed by another at the same time the other thread takes the resource now needed by the original thread. Then both threads would hang, waiting on each other to give up access to the needed resources.

Both of these issues can be avoided by making the distance array an array of atomic integers. This allows threads to atomically read the distances to the vertices to check if an update is needed, and also atomically write new distances for needed updates. The updated code is shown below:

```
atomic<long long> dist[MAX_NODES];

void updateEdges(int startEdge, int endEdge){
    for(int edgeID = startEdge; edgeID < endEdge; ++edgeID){
        auto [u, v, w] = edges[edgeID];
        if(dist[u] < INF && dist[u] + w < dist[v]){
            dist[v] = dist[u] + w;
        }
    }
}
```

This removes the need for locks offering better performance. One final issue that can be noted is the recreation and destruction of all threads at the end of each iteration. These operations are expensive. Rather than recreating all threads and joining them at the end of every iteration, they can be created all at once and communicate with each other. What needs to be maintained is that all edge relaxations from one iteration be conducted before any of the next. The threads can therefore signal if they are working or waiting for all other threads to wrap up their computations for this current iteration. This can be done by counting the number of currently working threads via an atomic integer. When a thread is entering its workload, it adds one to the counter. Then, when it has finished, it decrements this counter and waits for all other threads to be finished. It will know everyone is done when the counter reaches 0. A simple implementation is shown below:

```
vector<thread*> threads(numThreads);

for(int threadIdx = 0; threadIdx < numThreads; ++threadIdx){
    int startEdge = numHandledByEachThread * threadIdx;
    int endEdge = numHandledByEachThread * (threadIdx + 1);

    endEdge = min(endEdge, numEdges);

    threads[threadIdx] = new thread(updateEdges, startEdge, endEdge);
}

for(int threadIdx = 0; threadIdx < numThreads; ++threadIdx){
    threads[threadIdx]->join();
}
```

```
atomic<int> numThreadsWorking(0);

void updateEdges(int startEdge, int endEdge){
    for(int iteration = 0; iteration < numNodes - 1; ++iteration){
        ++numThreadsWorking;

        for(int edgeID = startEdge; edgeID < endEdge; ++edgeID){
            auto [u, v, w] = edges[edgeID];

            if(dist[u] < INF && dist[u] + w < dist[v]){
                dist[v] = dist[u] + w;
                ++numUpdates;
            }
        }

        --numThreadsWorking;

        while(numThreadsWorking > 0);
    }
}
```

After the running of benchmarks, The only comparable parallelization implementation is the final implementation. The first two implementations struggled to show any efficiency improvements on specific graphs, and ended up being dramatically slower on large graphs due to the locking and thread creation issues mentioned earlier. The final and best implementation with 10 concurrent threads performed well. On average in very large random graphs, the algorithm is about two to three times faster in practice than the standard single-threaded algorithm.

V. The Disjoint Set Union Problem

A. Problem Definition

The Disjoint Set Union, alternatively called the Union Find Problem, is to maintain several disjoint sets with the ability to merge these disjoint sets together and determine if different items are in the same set. This can be achieved by maintaining a unique representative for each disjoint set. When we union two sets, we make the representative of one set the representative of the other set, thus uniting the sets into one.

Such a data structure that can do this should support the following operations:

$find(x)$: Find and return the representative of the set that contains x.

$unite(x, y)$: Get the representative of the set containing x and the representative of the set containing y. If the representatives are the same, they are in the same set, otherwise, combine the sets by setting the representative of one of the two representatives to the other.

All sets begin only containing it's index, with it's representative being the index. We will also include the following operations:

$sameSet(x, y)$: Determine if x and y are in the same set by getting their respective representatives and comparing if they are the same or not.

B. Sequential Implementations

We will examine naïve implementations of $find(x)$ and $unite(x, y)$, and go over the different optimizations that can be applied to them to improve overall time complexity. As we go along the path of representatives, we will call the next representative the parent, and the representative of the set the root.

```
int find(int x) {
  while(parents[x] != x) {
    x = parents[x];
  }
  return x;
}
```

Listing 9. Naïve implementation of $find(x)$.

```
void unite(int x, int y) {
  x = find(x);
  y = find(y);
  if(x != y) {
    parents[x] = y;
  }
}
```

Listing 10. Naïve implementation of $unite(x, y)$.

These implementations have problems because we have no rules on how we designate who becomes the root of who. We can see here that in the worst case, we can create a long chain of parents, which would make the worst case of $find(x)$ be $O(n)$, with no improvements to happen over time. There are several optimizations that we can do to improve the time complexity. We will first look at linking by size, linking by rank, and linking by random index.

When we link by size, the idea is to maintain the size of each disjoint set, and use the sizes to determine what the new root will be in $unite(x, y)$. When we go to try and unite the two sets, we will make the root of the set that has a larger size be the root of the new combined set. Ties are broken arbitrarily.

```
void unite(int x, int y) {
  x = find(x);
  y = find(y);
  if(x != y) {
    if(sizes[x] < sizes[y]) {
      parents[x] = y;
      sizes[y] += sizes[x];
    } else {
      parents[y] = x;
      sizes[x] += sizes[y];
    }
  }
}
```

Listing 11. Implementation of $unite(x, y)$ with linking by size.

When we link by rank, the idea is to maintain the length of the longest path from the root to any other value in the set. When we $unite(x, y)$, we make the root with the higher rank the root of the new combined set. If there is a tie, we arbitrarily choose the root of the new set, and increase the rank by 1.

```
void unite(int x, int y) {
  x = find(x);
  y = find(y);
  if(x != y) {
    if(ranks[x] < ranks[y]) {
      parents[x] = y;
    } else if(ranks[x] > ranks[y]) {
      parents[y] = x;
    } else {
      parents[x] = y;
      ranks[y]++;
    }
  }
}
```

Listing 12. Implementation of $unite(x, y)$ with linking by rank.

When we link by random index, the idea is to create a random ordering of the values, and use the position in the ordering to determine who becomes the root. The root with the later position in the ordering becomes the root of the combined set. Since each position is unique, there will never be ties.

```
void unite(int x, int y) {
  x = find(x);
  y = find(y);
  if(x != y) {
    if(index[x] < index[y]) {
      parents[x] = y;
    } else {
      parents[y] = x;
    }
  }
}
```

Listing 13. Implementation of $unite(x, y)$ with linking by random index.

Another point of improvement that can be made is that if the parent of x is y and the parent of y is z, since we only care about the root of the whole set, we can get rid of this chain by simply making it so that parent of x is z. There are several optimizations for achieving this, being compression, halving, and splitting.

When we do compression, we simply make all the parents on the path are set to the root.

```
int find(int x) {
  if(x == parents[x]) {
    return x;
  }
  parents[x] = find(parents[x]);
  return parents[x];
}
```

Listing 14. Implementation of $find(x, y)$ with compression.

When we do halving, we replace the parent of every node on the path with it's grandparent.

```
int find(int x) {
  while(x != parents[x]) {
    int y = parents[x];
    int z = parents[y];
    parents[x] = z;
    x = y;
  }
  return x;
}
```

Listing 15. Implementation of $find(x, y)$ with halving.

When we do splitting, we replace the parent of every other node on the path with it's grandparent.

```
int find(int x) {
  while(x != parents[x]) {
    int y = parents[x];
    int z = parents[y];
    parents[x] = z;
    x = z;
  }
  return x;
}
```

Listing 16. Implementation of $find(x, y)$ with splitting.

References

[1] Seo, Jarim. "Parallel Implementation of a Minimum Spanning Tree Algorithm." SEO-THESIS-2020, Texas State University, Dec. 2020, https://digital.library.txstate.edu/bitstream/handle/10877/12921/SEO-THESIS-2020.pdf?sequence=1&amp;isAllowed=y.

[2] "Prim's Minimum Spanning Tree (MST): Greedy Algo-5." GeeksforGeeks, 18 Jan. 2022, https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/.

[3] Graves, Jeffrey A. Parallelizing Single Source Shortest Path with OpenSHMEM. United States: N. p., 2017. Web.