Minimum-weight Spanning Tree in Parallel

Ben Ledingham and Sunil Mann
CMPT 431, Simon Fraser University

1. Introduction

In graph theory, it is important to look at various Spanning Trees of a connected weighted graph. Spanning Trees are subsets of the graph that include every vertex and are acyclic. Special subset of these trees are Minimum-weight Spanning Trees (MST). These are tree that are built using a weighted graph that aims to find the Spanning Tree with the least weight. This is important in many such cases where someone wants to find the smallest spanning tree to connect every vertice. It is also important while solving this problem to recall that our Tree is acyclic. There are several famous serial algorithms for solving this problem, in this report, Prim's algorithm is the main focus.

2. Background

The goal of Prim's algorithm is to build this tree one vertex at a time. Starting from an arbitrary vertex, each outgoing edge is evaluated. The edge with the least weight and that does not form a cyclic is added. This process is repeated until there is a path to each edge. Currently, this is the serial implementation but there are parts of the algorithm that can be parallelized. The goal of the project is to find an efficient alternative to the serial Prim's algorithm that is sped up when running in parallel. When Prim's algorithm is run in serial it has a Big-O runtime of $O((V + E) log\, V)$. Thus the idea when designing the parallel implementation was to try to achieve a speed-up when going over the various vertices and edges. The main idea of this implementation is to break the graph into disjoint sets of vertices that are then merged into one final set that represents the MST.

3. Implementation Details

The implementations of MST we decided on are all based on the parallel prim's algorithm presented by Ramaswamy & Patki (1). In this paper, the two authors present a version of the prim's algorithm optimized for a distributed runtime environment. We modified our implementations using this algorithm as a starting point to simplify comparisons between the three implementations.

3.1. Serial Prim's

The serial implementation was created to help benchmark the runtime of the other two. Given an undirected connected graph G with V vertices and E edges, this implementation breaks G into V disjoint sets of vertices. The algorithm then repeatedly performs two logical steps on these disjoint sets to construct an MST. Step one finds a single minimum weighted outgoing edge from every disjoint set. Step two iterates over these edges and adds them to the final MST while merging the disjoint sets the given edge connects. These two steps are repeated until the disjoint set contains only a single set, therefore no more outgoing edges can be taken.

```
SerialPrims(Graph g) {
   DisjointSet ds(g.num_vertices);
   vector<Edge> mst;
   vector<Edge> min_edges;
   for(every vertex id) {
      // add the minimum weighted edge leaving the set the vertex is a part of
      min_edges.push_back(min_leaving_edge);
   }
   sort(min_edges); // sort smallest to largest
while(min_edges contains edges) {
      for(every edge in min_edges) {
         if(edge span across two disjoint subsets in ds) {
            ds.merge(vertex1.id, vertex2.id); // merge the sets spanned by the edge
            mst.push_back(edge); // add the edge to the mst
         }
      }
      min_edges.clear(); // empty the min_edges vector
      for(every vertex id) {
         // add the minimum weighted edge leaving the set the vertex is a part of
         min_edges.push_back(min_leaving_edge);
      }
   }
   return mst;
}
```

3.2. Threaded Prim's

The threaded implementation starts with the same disjoint set created from the undirected connected graph G with V vertices and E edges. It then shares the same core steps as the serial version. Each iteration of the algorithm will find the minimum weighted outgoing edges and merge them with the final MST. The differences arise in the need to lock access to the global collection of minimum weighted edges and the division of the initial disjoint set among the available threads. You will see in the pseudo-code below, that the access to the collection of minimum edges is controlled through the locking of a mutex and the division of the disjoint set among the threads.

```
mutex minEdgesMutex;


SelectMinEdgesThread(vector<Edge>* min_edges, int start, int end) {
   vector<Edge> local_min_edges;
   for(every vertex id) {
      // add a min weighted leaving edge
      local_min_edges.push_back(min_leaving_edge)
   }

   lock(minEdgesMutex);
   // add local minimum edges to a global collection
   min_edges->push(local_min_edges);
   unlock(minEdgesMutex);
```

```
}

ParallelPrims(Graph g) {

   DisjointSet ds(num_vertices);
   vector<Edge> mst;
   vector<Edge> min_edges;
   vector<vector<int>> thread_ranges;

   for(each thread) {
      // add a range of indices into the disjoint set for each thread.
      thread_ranges.push_back({start,end});
   }

   for (each range in thread_ranges) {
      thread.start(SelectMinEdges, g, start, end);
   }

   sort(min_edges); // sort smallest to largest

   while(min_edges contains edges) {

      for(every edge in min_edges) {
         if(edge span across two disjoint subsets in ds) {
            ds.merge(vertex1.id, vertex2.id); // merge the sets spanned by the edge
            mst.push_back(edge); // add the edge to the mst
         }
      }

      min_edges.clear(); // empty the min_edges vector

      for (each range in thread_ranges) {
         thread.start(SelectMinEdges, g, start, end);
      }

      sort(min_edges); // sort smallest to largest
   }
   return mst;
}
```

In this implementation of Prim's the algorithm runs in parallel when searching for minimum outgoing edges, yet switches back to serial when merging those edges into the MST. This avoids the high contention of merging the disjoint sets and could be improved with a parallel implementation of the disjoint set data structure.

3.3. Distributed Prim's

The distributed Prim's algorithm follows the conventions of the serial and parallel versions. The algorithm assumes we have a disjoint set constructed from the graph G with V vertices and E edges. With G, the algorithm repeats the same steps by finding the same minimum weighted outgoing edges and then merging them one by one into the MST. Like the threaded version, the

distributed algorithm parallelizes the detection of the minimum outgoing edges from each disjoint set by assigning different portions of the set to different processes. However, this introduces complexity when each worker process must serialize and send their collections of minimum weighted edges to the root process for the merging step. Additionally, the root process must update the disjoint set of every worker after it merges the sets based on the minimum weighted edges. All this communication is done using MPI.

```
DistributedPrims(Graph g, int world_size, int world_rank) {

   DisjointSet ds(num_vertices);
   vector<Edge> mst;
   vector<Edge> local_min_edges;
   vector<Edge> global_min_edges;
   bool edges_remaining = true;

   vector<Edge> local_min_edges;
   for(every vertex id in my section of ds) {
      // add the minimum weighted edge leaving the set the vertex is a part of
      local_min_edges.push_back(min_leaving_edge);
   }

   // serialize and gather the min_edges found to the root process global_min_edges
   serialize(local_min_edges);
   gather(global_min_edges, local_min_edges, process0);

   sort(global_min_edges); // sort smallest to largest

   while(edges_remaining==true) {
      for(every edge in min_edges) {
         if(edge span across two disjoint subsets in ds) {
            ds.merge(vertex1.id, vertex2.id); // merge the sets that are spanned by the edge
            mst.push_back(edge); // add the edge to the mst
         }
      }

      global_min_edges.clear(); // empty the min_edges vector

      broadcast(ds); // send the updated disjoint set to workers

      for(every vertex id in my section of ds) {
         // add the minimum weighted edge leaving the set the vertex is a part of
         local_min_edges.push_back(min_leaving_edge);
      }

      // serialize and gather the min_edges found to the root process global_min_edges
      serialize(local_min_edges);
      gather(global_min_edges, local_min_edges, process0);

      // if worker processes find no outgoing edges alert other processes to end the loop
      if (global_min_edges.empty() && world_rank==0) {
```

```
            edges_remaining = false;
        }

        broadcast(edges_remaining);
        sort(global_min_edges); // sort smallest to largest
    }
    return mst;
}
```

4. Evaluation

The implementations were all tested against eachother on the “cs-cloud.cs.surrey.sfu.ca” server. This way the experiments were run on a level playing field as the MPI implementation needs Slurm to be evaluated properly. The following shows the settings used for each implementation.

Serial:

```
#!/bin/bash
#
#SBATCH --cpus-per-task=8
#SBATCH --time=02:00
#SBATCH --mem=1G
#SBATCH --partition=slow
#SBATCH --output=test_serial.txt

srun ./mst_serial inputGraphs/50k-60k.csv true
```

Threaded:

```
#!/bin/bash
#
#SBATCH --cpus-per-task=8
#SBATCH --time=02:00
#SBATCH --mem=1G
#SBATCH --partition=slow
#SBATCH --output=test_thread.txt

srun ./mst_thread inputGraphs/50k-60k.csv 4 true
```

MPI:

```
#!/bin/bash
#
#SBATCH --cpus-per-task=1
#SBATCH --nodes=1
#SBATCH --ntasks=4
#SBATCH --partition=slow
```

```
#SBATCH --mem=1G
#SBATCH --output=test_mpi.txt

srun ./mst_mpi inputGraphs/50k-60k.csv true
```

The process for evaluating the scripts was to run each one, and then to compare the overall time taken to run the different algorithms. For the graph used it was randomly generated and included 60,000 edges. Each algorithm was run multiple times on this graph. Both the MPI and threaded implementations were run with the same number of processes to keep the division of labor fair for them. When run with n = 2 tasks, the times averaged out to 96 seconds for the serial implementation, 11 seconds for the threaded implementation and 54.5 seconds for the MPI implementation. When run with n = 4 tasks, the times averaged out to 96 seconds for the serial implementation, 5.6 seconds for the threaded implementation and 25.5 seconds for the MPI implementation. From this we achieved a speed up for the parallel algorithms, with the threaded implementation vastly out performing the other implementations.

5. Conclusions

There are many state of the art algorithm implementations that solve the minimum spanning tree problem. In this report, we covered one specific implementation of Prim's algorithm and measured the gains in performance that were seen when parallelizing and distributing the execution of that implementation. What we found was that the serial algorithm benchmarked a speed of 95.6 seconds on a graph with 50,000 vertices and 60,000 edges. On this same graph, we saw that the parallel version took 5.6 seconds and the distributed completed its MST in 27.6 seconds. Though the computations from the parallel and distributed versions were similar the communication overhead proved to be a big difference as it added up in the distributed's overall time.

These speedup's were seen as a result of parallelizing the process of choosing the minimum weighted outgoing edges from each subset of vertices. This process is computationally expensive and is easily compostable into multiple parts. However, the slowdown from the parallel to distributed version does tell us that the communication overhead of sending these minimum edges back and forth does not out-weigh the performance gain.

Works Cited

(1) Ramaswamy, S. I., & Patki, R. (2015). distributed minimum spanning trees.