Minimum Weight Spanning Tree

Application Description

The Spanning Tree of an undirected graph is an acyclic subset of the graph's edges that connects all of the vertices belonging to the graph. In the graph's edges are labeled with weights, the the Minimum Weight Spanning Tree (MST) is a spanning tree that has the minimum sum of edge weights. In the following, we describe briefly, two algorithms for computing the MST of a weighted undirected graph:

1. Kruskal's Algorithm
2. Boruvka's Algorithm

Boruvka's Algorithm

Boruvka's algorithm [1] computes the minimal spanning tree through successive applications of *edge-contraction* on an input graph (without self-loops). In edge-contraction, an edge is chosen from the graph and a new node is formed with the union of the connectivity of the incident nodes of the chosen edge. In the case that there are duplicate edges, only the one with least weight is carried through in the union. Figure 1 demonstrates this process. Boruvka's algorithm proceeds in an unordered fashion. Each node performs edge contraction with its lightest neighbor. This is in contrast with Kruskal's algorithm where, conceptually, edge-contractions are performed in increasing weight order.

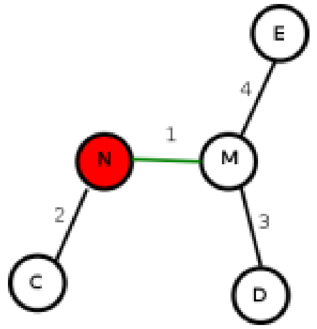


Figure 1a: Before edge-contract.

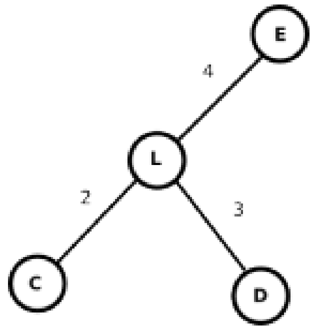


Figure 1b: After edge-contract.

While in principle, the edge-contraction operation can be performed by modifying the graph, it is costly to implement. Most implementations use a Union-Find (aka Disjoint-Set [2]) data structure, instead, to perform edge-contractions. Union-Find data structure assigns a unique component to each node; a component is identified by its representative, which initially is the node itself. Every node in the component has a (direct or indirect) reference to its representative. The algorithm uses a Find operation to find the representative of the component the node belongs to. If the two nodes of the lightest edge point to different representatives, then they must belong to different components and adding the edge to the MST does not introduce a cycle. Adding an edge to the MST involves combining the two components (to which the two nodes of the edge belong to) into one component. This is done by performing a Union operation on the representatives of the two components, which essentially makes one representative point to the other. Figure 2 gives the pseudocode for Boruvka's algorithm:

```
Graph g = /* read input */; UnionFind uf (g.getNodes ()); Workset ws = g.getNodes(); foreach (Node n : ws) { Edge e = minWeight
(g.getOutEdges (n)); Node rep1 = uf.find (e.src); Node rep2 = uf.find (e.dst); if (rep1 != rep2) { e.markInMST(); Node rep = uf.union
(rep1, rep2); ws.add (rep) } }
```

Figure 2: Pseudocode for Boruvka's algorithm.

Data Structures

There are three key data structures used in Boruvka's algorithm:

Graph	The input, which is a weightedundirected graph.
Unordered Set	The workset containing representative nodes
Union-Find	A disjoint-set data structure used to maintain components of graph.

Parallelism

Initially, there is a lot of parallelism in Boruvka's algorithm as about half the nodes can perform edge-contraction independently. After edge-contraction, the graph becomes more dense and there are fewer nodes, so the available parallelism decreases quickly. Figure 3 shows the parallelism profile for a sample input. Most parallel implementations of minimal spanning tree algorithms begin with Boruvka's algorithm but switch to another algorithm as the graph becomes more dense.

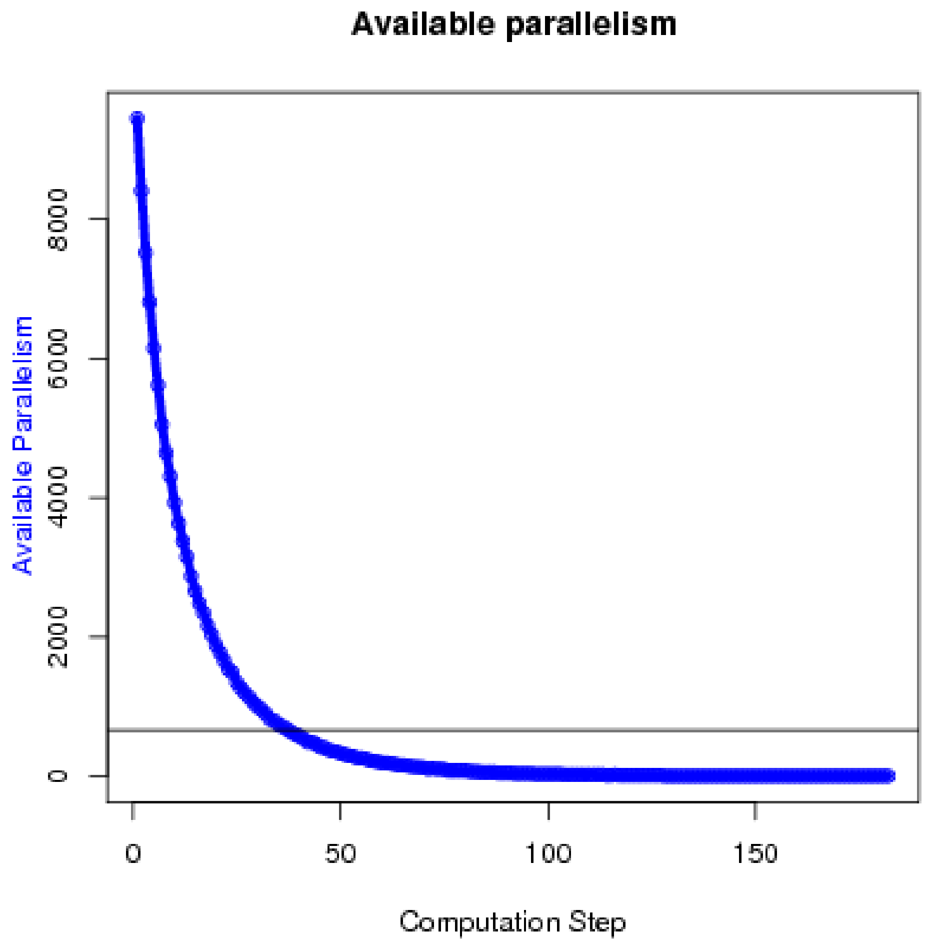
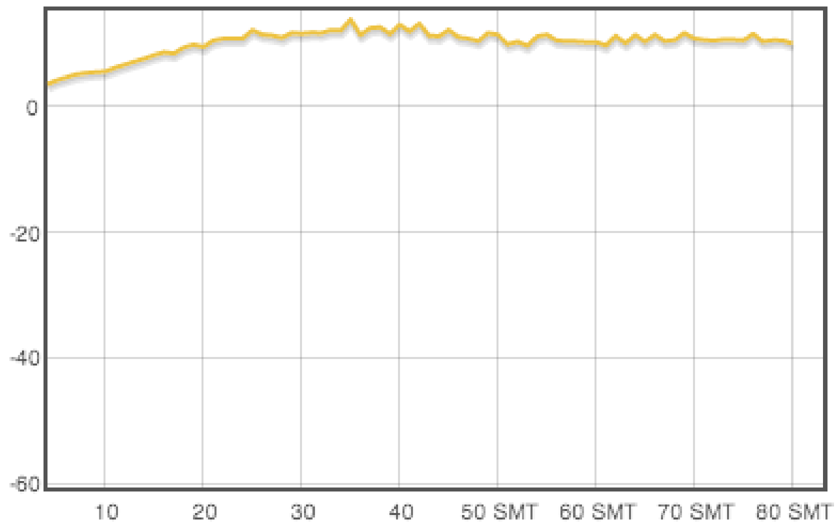


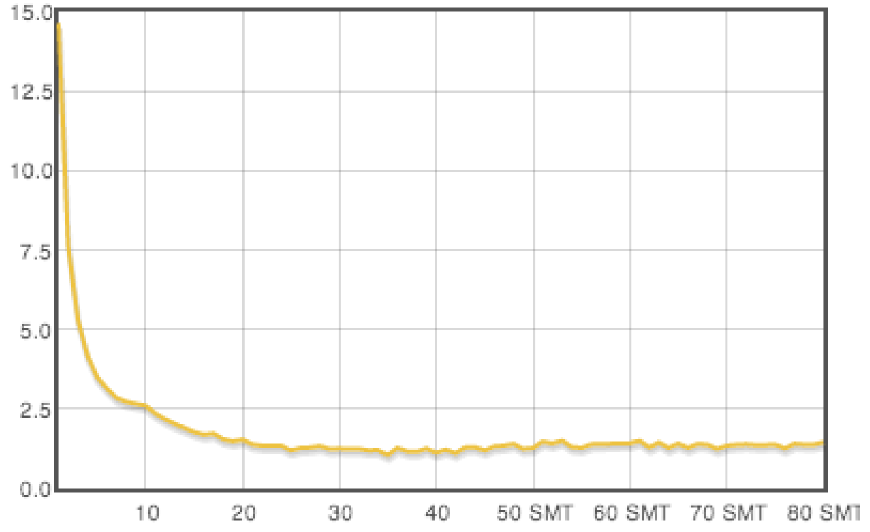
Figure 3: Available parallelism in Boruvka's algorithm.

Performance

In our implementation of Boruvka, we split the loop body into two parallel phases, which have a barrier in between them. The first phase invokes Find on the lightest edge of each representative, while the second phase performs Union operation. The tresulting set of representatives is then used for the next round in a similar manner. This makes the concurrency control of UnionFind data structure simpler, where one of the costly and complex mechanism is finding conflicts between concurrent Union and Find operations. With splitting into phases, Find operations can proceed in parallel with Find operations (conceptually being readonly). Similarly Union operations on separate compononets can proceed in parallel without any concurrency control.



Scalability (self-relative) versus number of threads



Time (seconds) versus number of threads

apps/boruvka/boruvka inputs/road/USA-road-d.USA.gr

Command line

Figure 4: Performance results on a random graph of 2^24 nodes and 4 times that many edges with uniformly random edge weights.

Kruskal's Algorithm

Kruskal's algorithm processes all the edges in the graph, one by one, in increasing order of weight. In each step the algorithm picks the next minimum weight edge and tries to add it to the MST if adding the edge does not introduce a cycle in the MST. Just like Boruvka's algorithm, the cycles are detected by maintaining component information, using a Union-Find data structures. The algorithm uses a Find operation to find the representative of a component that a particular node belongs to. If the edge belongs to different components, it is added to the MST and the components are merged using Union operation. When the MST has been computed, all the nodes belong to one component. The algorithm terminates when all the edges have been processed. An implementation can terminate earlier if it has performed *V-1*, Union operations, where *V* is the number of nodes in the graph. Figure 4 below shows the pseudocode for Kruskal's algorithm:

```
Graph g = /* input graph */;
Vector edges = g.allEdges ();
sort (edges) /* by weight */
UnionFind uf (g.getNodes ()) ;
foreach (Edge e : edges) {
  Node rep1 = uf.find (e.src);
  Node rep2 = uf.find (e.dst);
  if (rep1 != rep2) {
    e.markInMST();
    uf.union (rep1, rep2);
  }
}
```

Figure 4: Pseudocode for Kruskal's MST algorithm.

Data Structures

There are three key data structures used in Kruskal's algorithm:

Graph
    The input undirected graph.
Ordered Set
    Because the algorithm is based on examining the edges in order of their weight, the workset can be represented as an ordered set.
Union-Find
    A disjoint-set data structure used to maintain components of graph.

Parallelism

The active nodes in Kruskal's algorithm are the edges of the graph. When executing the Find operation on a node, an iteration follows the representative pointers until it reaches the representative node of the component. Therefore, the neighborhood consists of the nodes visited when finding representatives for each of the nodes of the edge. The Union operation does not access any more nodes, and performs modifications to the state of representatives. Figure 5 shows the available parallelism of Kruskal's algorithm for a graph that represents the road network of Florida. The edge weights are distances between locations. The union operations performed by the algorithm combine the components into bigger ones, essentially coarsening each component. Therefore, initially many edges can be processed in parallel, but as the computation proceeds, the components become big and the available parallelism drops quickly, approaching one towards the end.

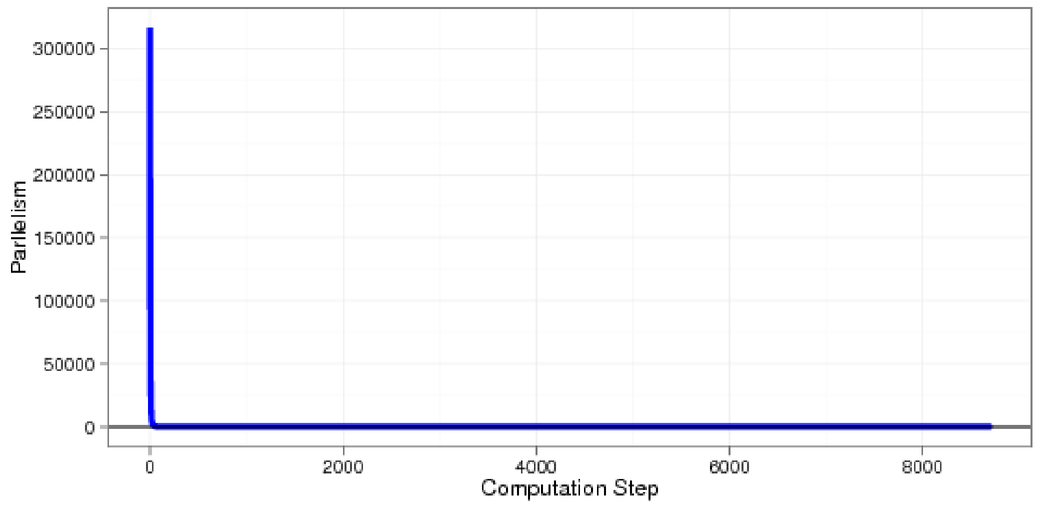


Figure 5: Available parallelism in Kruskal's Algorithm; input = Florida road network.

Implementation and Important Optimizations

In a manner similar to Boruvka, we split the loop body of Kruskal into two parallel phases, where first phase invokes Find on the two nodes of an edge, and, the second phase performs Union between the representatives of non-self (non-cycle creating) edges. In order to ensure that edges are added to the MST in priority order, we enforce priority locking on the neighborhood. The neighborhood of an edge is its two component. Since each component has a representative, it suffices to acquire a lock on the representative instead of all the members of the component. Due to priority locking, an edge with lighter weight steals a lock from a heavier edge. In the first phase, all the non-self edges acquire priority locks on their representative. In the second phase, only the edges that own locks on their representatives proceed with the union operation. The union operation on such edges can be performed without any further locking. The edges that have failed to acquire their locks are relegated to the next round, and the process repeats until there are no non-self edges left. As the parallelism profile suggests, typically a small fraction of edges is able to acquire the priority locks, while a large number is put off to the next round. Therefore, we use a window based scheduler, where a subset of highest priority edges is scheduled for processing. This improves the success ratio of priority locking and hence avoids wasting resources. This optimization was borrowed from PBBS implementation by Blelloch et al [3].