Parallel Minimum Spanning Tree Data Structures and Algorithms

Connor Clayton, Jacqui Fashimpaur

SUMMARY

We explored several parallel variations on two Minimum Spanning Tree algorithms--Prim's and Boruvka's--designed to run with shared memory on an 8-core CPU. We achieved the best speedup using Boruvka's algorithm and a data structure called Flexible Adjacency List.

BACKGROUND

Minimum Spanning Tree Algorithms

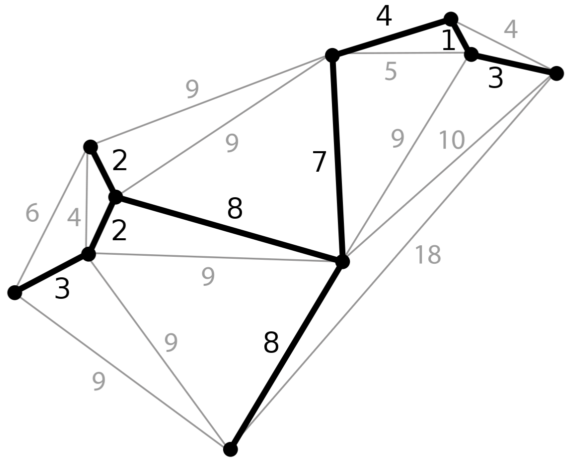


Figure 1: a minimum spanning tree
https://en.wikipedia.org/wiki/Minimum_spanning_tree#/media/File:Minimum_spanning_tree.svg

A Minimum Spanning Tree (MST) of a weighted, undirected, connected graph G = (V, E) is an acyclic subset of G such that all vertices are connected which has the minimum possible total edge weight. For a graph with all unique edge weights, this tree will be unique. Otherwise, there may be multiple minimum spanning trees, but all will have the same total weight. There are many algorithms for finding an MST, but all rely on the property that for any subset of the MST, the lowest-weight edge intersecting this component will be in the final MST.

One common algorithm is Prim's algorithm. Prim's algorithm builds up the MST starting with a single vertex. At each iteration, it adds the lowest weight edge in the neighborhood of the current tree until all vertices are included. As written, this algorithm is inherently sequential, since it relies on the tree calculated in the previous iteration in order to calculate the neighborhood of the next iteration. However, we explored a variation of Prim's algorithm that allows us to add parallelism.

Another common algorithm is Boruvka's algorithm. Boruvka's initially considers each vertex to be a component, and then proceeds in three stages until there is only one component remaining:

1. Find-min: for each component, find the minimum weight edge leaving that component. Add edges to the MST.
2. Connect-components: using the edges found in find-min, identify which components are now connected and group them together.
3. Compact-graph: condense each connected component into a single supervertex.

Once only one component remains, all vertices have been connected and the MST has been found.

Implementation Details

For most of our algorithms, we represent graphs as adjacency lists. We store a 2D array, where the outer array represents vertices and each inner array stores the incident edges to a vertex. The adjacency list representation was chosen because it is optimal for iterating over the neighbors of a vertex (a very common operation in both algorithms). A vertex's neighbors can be accessed at a single array index and are stored consecutively for good locality. The input to each algorithm is an adjacency list graph and the output is the total weight of the minimum spanning tree. We chose to return the MST weight and not the explicit MST for correctness checking purposes, to allow for implementation differences in the case of multiple valid MST's.

Another data structure we used for one algorithm is a flexible adjacency list. This data structure essentially stores multiple adjacency lists at each vertex as a linked list. To merge two vertices, one simply needs to point one vertex's flexible adjacency list to the other. This creates interesting changes in the algorithm that shift work away from compact-graph, allowing some optimizations.

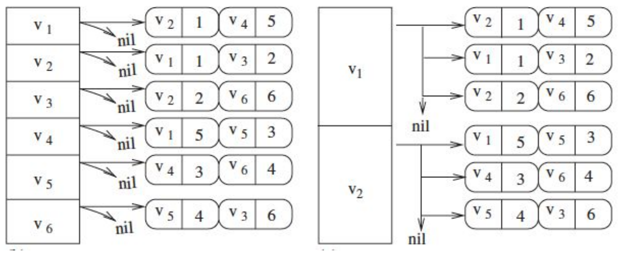


Figure 2: Flexible adjacency list before (left) and after(right) merging vertices 1-3 and 4-6

Prim's algorithm is inherently sequential: data dependencies exist within each iteration when computing the minimum-weight neighbor, as well as between iterations. Thus, our approach (described in detail below) was to run multiple instances of Prim's algorithm concurrently across different sections of the graph. This adds obvious parallelism during much of the algorithm, but creates contention for memory when combining sub-trees found by different instances.

Boruvka's algorithm, on the other hand, lends itself much more naturally to parallelism. In each of the three steps, components are independent of each other (in step 3, it is actually the newly connected components that are independent). However, operations within a component can be sequential and expensive (especially within section 3), and the benefit of inherent parallelism is mitigated when the number of components decreases very quickly. Compacting the graph is the most computationally intensive step and also the least naturally parallel, because if many components point to each other in a chain then they all need to be compacted, but this takes time proportional to the length of that chain.

APPROACH

All code was written in C++, with parallelism implemented using OpenMP. The code was run on the GHC machines. We didn't use any existing code, but referenced algorithms described in Bader et al [1].

Prim's Algorithm

We began by implementing a sequential version of Prim's. This was the same algorithm as described above: building the tree one vertex at a time by choosing the nearest vertex, and updating the distance from our tree to any given vertex at each step. We call this the "standard" implementation. One thing we experimented with that was unsurprisingly slow was a "simple parallel" implementation that matched the above algorithm except for parallelizing all of its loops when possible. More discussion of this can be found in "results," but in any case it was clear that we needed to modify the algorithm itself.

Then we moved on to an implementation inspired by Bader et al [1], called "parallel threads". First, we tried one using OpenMP parallel regions. Each thread was given its own array to keep track of its personal tree's distance to each vertex, and they were each assigned a vertex to start on. We experimented with how to assign these start vertices and found that random assignment or assignment to evenly spaced numbers (ie. `(num_nodes / num_threads) * thread_id`) worked better than assigning thread i to vertex i, because our graph representation tended to put closely-numbered vertices together, thus increasing collisions between subtress later. After this setup, each thread ran Prim's algorithm independently from each other, building a subtree until its next nearest neighbor was a vertex already part of another thread's tree. When this happened, the second arriving thread would merge its tree into the other thread's, and then start all over from scratch (we found that random assignment worked best for the re-spawning, because in expectation only one guess was needed--much faster than scanning to find an empty vertex).

However, detecting collisions between trees and then merging those trees without any race conditions proved to be a difficult task. If one thread looked at a vertex, saw it had no color (we used “color” to represent ownership by a specific thread), and then tried to set its color, another thread could have modified the color value in between those two steps. This would lead to both threads thinking they owned that vertex, and too many edges would end up being included in the MST. To resolve this problem, we created a single shared omp lock that protected all vertex’s colors at once (because collisions like this were rare enough not to warrant individual locks).

Another more complicated issue was that of actually merging two trees. The process of merging involved iterating over both threads’ distances list and replacing one thread’s distance for each vertex with the minimum of the two threads’ distances. After this was done for each vertex, the thread that did not receive the minimum values could clear out its distances (ie. reset them to INT_MAX) and start again with a new vertex. There are a lot of opportunities for race conditions in this process--if one thread starts merging its distances into another thread’s distance list while that thread is reading the distance list to find its next closest neighbor, that would interfere with correctness. And if a thread tries to merge its distances with another thread that is in the process of clearing its distances and resetting, this is also a serious problem. After experimenting with several arrangements of locks and critical regions, we came up with the following configuration:

We maintain a 2D array of booleans, where the [*i, j*] entry is 1 if and only if thread *j* is waiting to merge into thread *i*. The [*i, i*] entry is 1 is and only if thread *i* is waiting to merge into some other thread (this indicates that no one else is able to request to merge into thread *i*--it is going to be cleared soon). All threads check this array and perform necessary merges at the start of each iteration of their personal Prim’s algorithm. When they do, they set the appropriate flags back to 0. There is also an array of `num_threads` locks, with one corresponding to each thread, to protect against race conditions. Say thread *a* is running Prim’s and finds that its next nearest neighbor already belongs to thread *b*. It will try to grab lock *a* and lock *b* from the array, and once it has done this it will set [*b, a*] and [*a, a*] to be 1. Then it will release the locks and busy wait for [*b, a*] to be 0. Once this happens, it will clear its distances find a random new start node, and set [*a, a*] back to 0, allowing it to be merged with again. This way, threads don’t need to worry about locking their own distances list--they can trust that only they will ever edit it. This reduces contention.

Though this system ultimately worked, it has a lot of moving parts and we faced some very subtle bugs as we were developing it. One of these was an instance of deadlock when two threads would try to merge with each other at the same time. Because the code had each thread grab their own lock and then the other lock, the two threads would do the first step at the same time and then both be stuck, unable to access the second lock they needed. We fixed this by putting both lock acquisitions in an OpenMP critical region. Another tricky issue was the busy wait that a thread does when it's trying to merge its tree into another. We used a simple while loop, spinning as long as the value at the corresponding boolean in the array was true. When we ran the code, threads would get stuck in this loop forever, despite numerous print statements confirming for us that the value in the array had indeed been set to 0. It turned out that the thread had been caching the array entry's value, causing it to read a 1 even when another thread had changed it. When we added the "volatile" keyword to the array, this forced threads to read its entries from memory, fixing our correctness issue but adding to our runtime quite a bit. We added a short `sleep()` in the middle of the busy wait to reduce the amount of times we were reading from memory. More analysis of this is below.

Boruvka's Algorithm

We began by implementing a sequential version of Boruvka's. We speculated that the algorithm as described above would not have great sequential performance since it was designed to be parallel, so we started with a different version. Instead of explicitly merging components together and updating the graph structure between rounds, we kept a lookup table of which component each original vertex now belonged to. We looped over all vertices to find the minimum edge for each component, then looped over each of these edges and added it to the MST if it did not create a cycle, updating the component lookup table after each addition. Whenever possible, we designed these loops such that they could be executed in any order, thus making it easier to parallelize them later. We call this the "standard" implementation.

The next step was to parallelize this implementation. There were several things that made this difficult. Since the vertices were not organized by their component, there was no way to assign a thread to each component in the find-min step. In addition, the addition of edges to the graph could not be directly parallelized because components depend on previous additions to determine validity and component labels. Thus, only the inner for-loop which updates component labels could be run in parallel. Timing tests showed that this loop contributed to most of the run time, so this seemed like a good improvement. We call this the "parallel standard" implementation.

Third, we implemented the algorithm as described by Bader et al [1], involving explicit component merging (the "merging" and "parallel merging" implementations). The three stages of the algorithm above, and the steps taken to implement parallelism, are described here:

1. Find-min: this step is simple--each component is a single vertex, so loop over all its neighbors and record the cheapest edge. This was easily parallelizable with a pragma omp parallel for loop. An insignificant portion of the total time was spent in this step.

2. Connect-components: There were two options for parallelism in this step: assigning one thread per vertex or one thread per component. Since nothing is known about the connected components before this step, we created a loop that worked on each vertex independently. Viewing the set of minimum edges calculated in find-min as a directed graph, each vertex traverses through the graph until it encounters a cycle. Note that each vertex has exactly one out-edge (its cheapest edge) and, therefore, any connected component has exactly one cycle (a connected component of n vertices has n edges) which is reachable from all vertices in the component. For consistency, the lowest-numbered vertex in this cycle is declared the root, and the entire connected component is given this root label.

   The complexity of connect-components was necessary to avoid merging components and to keep the independence of each vertex. Parallelism was achieved by giving a private array to each thread to each vertex to record visited vertices.

   Also in this step, we calculate the weight of the edges added to the MST in this round. The sum of the weights of all the edges found in find-min will overcount the weight because there is a cycle in each connected component. Therefore, we sum the weights of all these edges and subtract the weights of the edges coming from all the roots to get the proper value. Lastly, we re-enumerate the new vertices (so they have consecutive values starting at 0) and update the labels of all the edges accordingly.
3. Compact-graph: the final step is by far the most time-consuming. Compacting begins by sorting supervertices by their new labels, then sorting the neighbor list of each vertex by their endpoints. This second set of sorts can be done concurrently and efficiently with dynamic scheduling, since the length of each neighbor list is unknown. Now, all supervertices to be aggregated are contiguous in the array, and neighbor lists can be merged. The merging takes place similarly to that of standard mergesort, except edges are not included if they form a self-loop or are redundant (are multi-edges).

   Each merge is sequential, so parallelizing graph contraction involved assigning one thread to each component-to-be. This is done neatly inside a for loop by calculating the offset of each set of contiguous supervertices beforehand.

The above Boruvka implementation did not achieve the speedup we hoped for, so we implemented a different version described by Bader et al [1] that used an interesting graph data structure known as a flexible adjacency list (FAL), described above. This is the “FAL” implementation. Since the bottleneck in the explicit-merging method is the expensive sorts and merges inside compact-graph, FAL-Boruvka eliminates these operations. A FAL stores its neighbors as a linked list of adjacency arrays, and combines components by appending linked lists. It does not remove self-loops or multi-edges--instead, find-min filters through and does not select these edges. Parallelism was mapped to this algorithm in a similar way to the explicit component merging implementation.

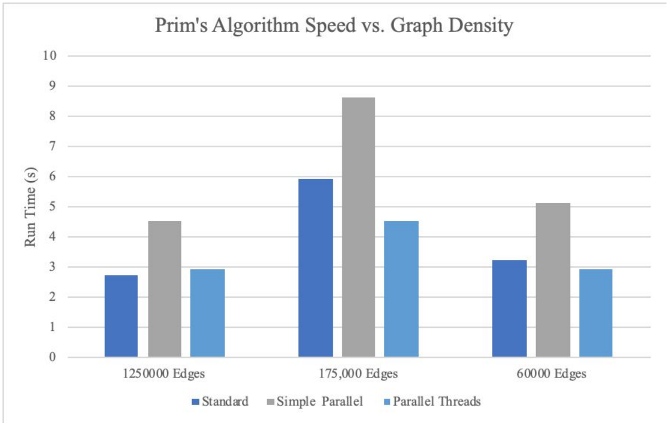
RESULTS

Speed tests were run on random graphs generated by a Python script, which included every possible edge with some probability. This probability was varied to create graphs of varying densities. Performance was measured by the total run time of each algorithm.

Prim's Algorithm

It was not surprising that sticking pragma omp parallel for on our Prim's algorithm loops failed to speed up the algorithm--the loops were almost always over a given node's neighbors rather than all of the nodes in the graph, so they were fairly short loops to begin with. The overhead of parallelizing these was rarely worth it, though perhaps it would be on very very dense graphs when the neighbor lists were long. In general, our "parallel threads" Prim performance had a modest improvement over the standard implementation, likely because the contention and communication required offset some of the benefit.

Another parameter with a more profound impact was our maximum_respawn_attempt variable, that would limit how many times a recently respawned vertex was allowed to randomly try to select an unclaimed vertex before being forced to give up. Our test found that decreasing this parameter only helped improve speeds. This is because in cases where many attempts were necessary, the tree was mostly completed already and the benefit of adding a new thread into the mix would be small.



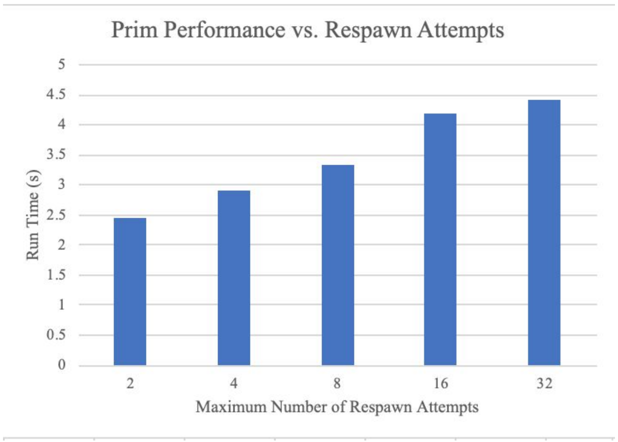


Figure 3: (top) performance of Prim implementations on graphs of varying densities; (bottom) performance of various sequential cutoff values

Boruvka's Algorithm

Each of the Boruvka implementations exhibited differing characteristics. The standard implementation was very consistent across the graphs in the figure, taking about 1 second each time despite widely varying numbers of edges. This consistency can be attributed to the inner for loop of the connect-components step that does $O(n^2)$ work (where n is the number of vertices), independent of the number of edges. As the charts show, find-min became faster as the graph became sparser, and connect-components dominates.

For the same reason, parallel standard was also consistent. The standard implementations actually surpassed the merging implementations for denser graphs. However, the merges are more efficient in the sparse case where merging is less costly.

The figures also show the limitations of the explicit merging Boruvka implementation. For higher edge counts, the expensive sorting and merging operations dominate in the compact-graph step. The merging implementations are significantly out-performed by the simpler standard implementations.

The performance of the merging implementations was the incentive for the flexible adjacency list (FAL) implementations. The figure shows the superiority of the FAL data structure in facilitating MST calculations across both

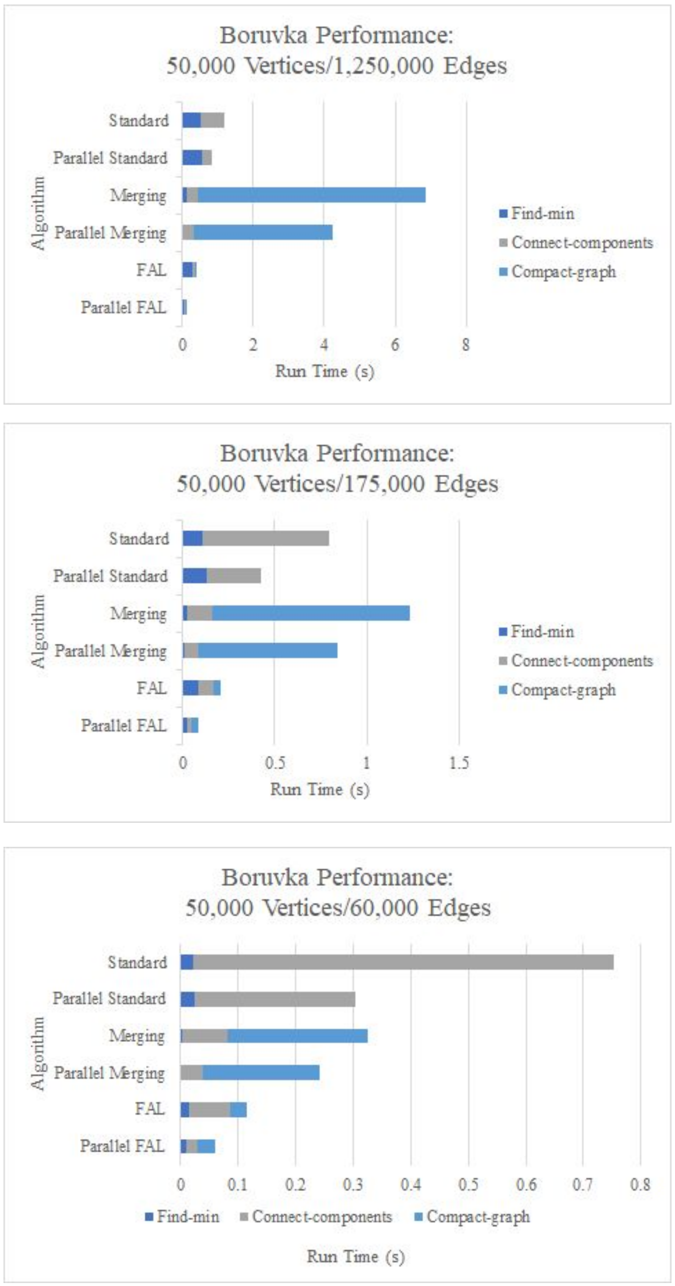


Figure 4: Performance of Boruvka implementations on graphs of varying densities

sparse and more dense graphs, with even the sequential FAL implementation out-performing all merging and standard implementations on all graphs. While the FAL greatly increased the amount of time spent in find-min, this was outweighed by the huge speedup in compact-graph.

An interesting finding across all Boruvka implementations was the generally modest speedup. Even using 16 OpenMP threads, the largest sequential-to-parallel speedup seen was 3.125 for FAL on the 125,000 edge graph (0.4s/0.128s). This makes sense because even though the algorithm is designed to be parallel, each of the three sections is dependent on the ones before it and therefore barriers are required between them. Furthermore, load balancing is difficult within each section because vertices (the basic unit which is parallelized over) tend to have very uneven neighborhoods and the number of vertices decreases exponentially in every round.

Overall

In general, we don’t expect graph algorithms to be highly parallelizable because of all of their dependencies and their limited arithmetic intensity. Our findings demonstrate just how important choosing the right data structure is. We believe that using a multi-core CPU with shared memory was the right approach, as graphs are just not organized in a way to make them very well-suited to SIMD instructions, or GPU parallelism. Because of how much communication was needed, message passing would likely not have worked better either.

REFERENCES

[1] Bader, David A., and Guojing Cong. “Fast Shared-Memory Algorithms for Computing the Minimum Spanning Forest of Sparse Graphs.” Journal of Parallel and Distributed Computing, vol. 66, no. 11, 2006, pp. 1366–1378., doi:10.1016/j.jpdc.2006.06.001.

DISTRIBUTION OF WORK AND CREDIT

Connor wrote the code to build graphs from text files, wrote the code for timing the algorithms, and focused on optimizing Boruvka’s Algorithm. Jacqui wrote initial sequential versions of both algorithms, including a sequential implementation of Boruvka’s using flexible adjacency lists, then focused on optimizing Prim’s Algorithm. Both wrote equal portions of the report and poster. We would say the total work was split 50%-50%.