15-418 Parallel Computer Architecture and Programming Final Project

Benjamin Chu - Jordan Widjaja
bcchu - jwidjaja
Friday, May 12th, 2017

1 Summary

In our project we attempt to parallelize boruvka's algorithm and manage some sort of meaningful speedup. We will analyze where the speedups come from and figure out why they arise. These speedups can arise from different properties of the graph such as the sparsity of the graph as well as the size and/or the representation of the graph. We might alter our data structures to fit the problem in more parallel friendly way. To do this we need some sort of graph generator to generate "fair" graphs that we can run boruvka's on.

2 Background

A **graph** is a data entity that has many useful real-world applications in mathematics and computer science. We define a graph $G = (V, E)$ by a set of *vertices* V and a set of *edges* E. Vertices are uniquely labeled as a numeral in the range $0, 1, \ldots, |V|-1$, and edges are labeled by its source vertex, destination vertex, and (unique and integer greater than 0) weight. One of the many problems involving graphs which are important to analyze is the **minimum spanning tree** problem. A minimum spanning tree is a connected subset of the edges in a graph which connects all vertices together without cycles and with the smallest weight possible. **Boruvka's Algorithm** is one of the popular efficient algorithms today which can solve such a problem of finding the minimum spanning tree of a graph. One of the data structures used in this algorithm is known as a **Union Find** data structure, which keeps track of the set of vertices, partitioned into a number of disjoint subsets. Two useful operations that are used in a union find data structure are the following:

- `find` - finds the particular subset that a vertex is in. Subsets are usually denoted by a representative vertex in that subset, and we can also use this function to see if two vertices are in the same subset.
- `union` - joins two subsets into a single subset

To that end, we can summarize Boruvka's Algorithm, as follows:

```
Give as input a connected, weighted graph
Initialize each vertex as an individual component (or set)
Initialize MST as empty
while there are more than 1 component, on each component:
   find closest weight edge that connects this and other
     component
   add edge to MST if not already added
return MST
```

The reason why union find is an effective data structure for Boruvka's Algorithm comes from the idea of the *light-edge property*, which states that for any cut in a graph, the minimum weight that crosses the cut is in the minimum spanning tree of the graph.

When representing graphs in code, there are many useful representations, but in particular we used:

- **adjacency list** - an array of size $|V|$ of C++ standard vector primitives, where each index in the array represents the source vertex and the vector in the array index represents a vector of edges. An edge is defined to be a struct consisting of the destination vertex and the edge weight.
- **edge list** - an array of edges, where an edge is a struct consisting of a source vertex, a destination vertex, and an edge weight
- **adjacency matrix** - a Boolean matrix where 0 in the (i, j)-th entry implies that i is not connected to j by an edge, and a 1 in the (i, j)-th entry implies that i is connected to j by an edge. Also, we maintain a separate C++ `int` vector primitive of size $|E|$ to store the weights of the edges.

2.1 Parallelization

Boruvka's is inherently parallel because we choose the minimum weight edge for each vertex and contract it into a vertex, there for reducing the number of components of the graph. This can be done in parallel as long as contracting the min edge for a component does not conflict with other contractions. Moreover, finding the min edge can be done in parallel as well. As long as we keep track the minimum for each vertex we can override which ever vertex's previous minimum edge atomically. Our issues with concurrency will come with the unionfind data structure our biggest challenge. We expect finding edges to be the most parallelizable since there are a lot more edges than vertices.

2.2 Graph Creation

Our initial approach to graph creation used the idea of creating arbitrary random edges of a graph when specified the number of vertices therefore creating an edge list. We then assigned random unique weights to these edges. If at any point we come up with an edge or weight we have seen before, we retry another random edge/weight. For an undirected graph some challenges we faced were attempting to have only one of the edges out of the (u,v) pair exist (I.E. if (u,v) exists in our edge list then (v,u) cannot for the sake of our implementation) We initially used this idea for small scale graphs ($\leq$ 20 vertices) for testing our algorithm. We can assert correctness on these small graphs since it is easy to see the MST. However, on larger graphs this method proved to be very slow as the random weight/edge assignment would have lots of collisions later. Also it proved difficult to demonstrate the fully connected graph property for larger graphs since it was difficult to fit into memory.

After demonstrating our boruvka's implemntation was correct on smaller graphs, we turned to larger datasets such the Stanford Large Network Dataset Collection (SNAP). These graph's were representations of real life data such as Facebook's social circles. These graphs were also represented as an edge list, so it was simple to modify our graph generator to add random weight edges to an existing dataset. Unfortunately our algorithm would simply never converge on some inputs SNAP's graphs. Due to the massive size of these graphs, it was almost impossible to test for interconnectedness (even though SNAP's dataset specified they were connected).

Finally we decided to switch to C++ to generate graphs as it was much quicker while using an adjacency matrix to represent our graph. We initialized an array of 0 to n-1 vertices and then created a permutation of that array to have a guaranteed connected graph. (A[0] $\rightarrow$ A[1], A[1] $\rightarrow$ A[2] etc) We then would generate two random numbers between 0 to n-1 to and change the corresponding cell to 1 of the adjacency matrix. If that cell was already 1, we left it as is. Although we never got the exact number of edges that we wanted, it always close enough as the likelihood for collisions at a lower edge count would be close to 0. To have unique weights, we added all the weights to an vector. Everytime we chose an edge to add to our graph, we would choose a random index from that vector and choose that as the weight. Then we would delete that weight from the vector so we would get no repeats. We did run into some issues of generating graphs of size large 32mb. We noticed that these graphs would not be complete since no matter what these graphs would always be at most 32 mb. So we decided to just keep graphs in memory asserting the randomness would not vary the algorithm time due to the randomness of the creation.

3 Approach and Results

NOTE: Although it says to have a separate approach and results section, we decided to combine the two as we would state our approach then show results, then move onto our next iteration of our algorithm which conforms to logical flow.

4 Sequential Algorithm Implementation

Our algorithm runs until the number of connected components is equal to 1 while starting out with n components. We reduce the number of components on each iteration of the algorithm by doing two things. We first find the min edges for each component and store it for each component. We do this by finding edges between components and then record the min edge in an array we create to keep track of the min edge. If we find an edge of the same component we pass on that iteration. The next iteration we iterate through each of the components and contract the edge into the component, therefore reducing the number of components and adding it to our MST. This is where our union-find data structure will come in handy. Our unionfind will allow us to quickly determine which vertices are in which components. Although the algorithm is relatively straightforward, we had initial difficulty implementing a 210 style solution of deleting edges and vertices that had been contracted. Thus our turn to a more standard unionfind for our datastructure and our reasons why below.

4.1 Tradeoffs with the Algorithm

Here comes our first tradeoff in terms of performance. As opposed to our 210 style implementation which directly modified the graph so we would have fewer edges/vertices. We instead choose to modify a unionfind data structure for ease of use (In 210 directly modifying the graph was much easier) We also know that for each iteration modifying the graph would incur some overhead for contracting edges into the graph. This is also not the easiest thing to parallelize as we run into concurrency issues with vertices and edges dissappearing from our graph, adding unnecessary complexity to our algorithm. However, with a unionfind implementation we do not have these issues with removing edges/vertices. (However we might run into some concurrency issues with union). We iterate through the entire graph of edges and vertices every single time unlike edge contraction which decreases the size of the graph each time. So even for 2 components a lot of these iterations might not matter (even though we continue these iterations) We hope that the most of the time in these worst case scenarios do not happen too often as the expected number of vertices contracted is at least 1/4 every time [1]. We acknowledge these shortcomings with the unionfind datastructure and simply assert that the complexity of the 210 implementation would cause more harm than good.

[1]15-210 notes

4.2 Initial testing with sequential

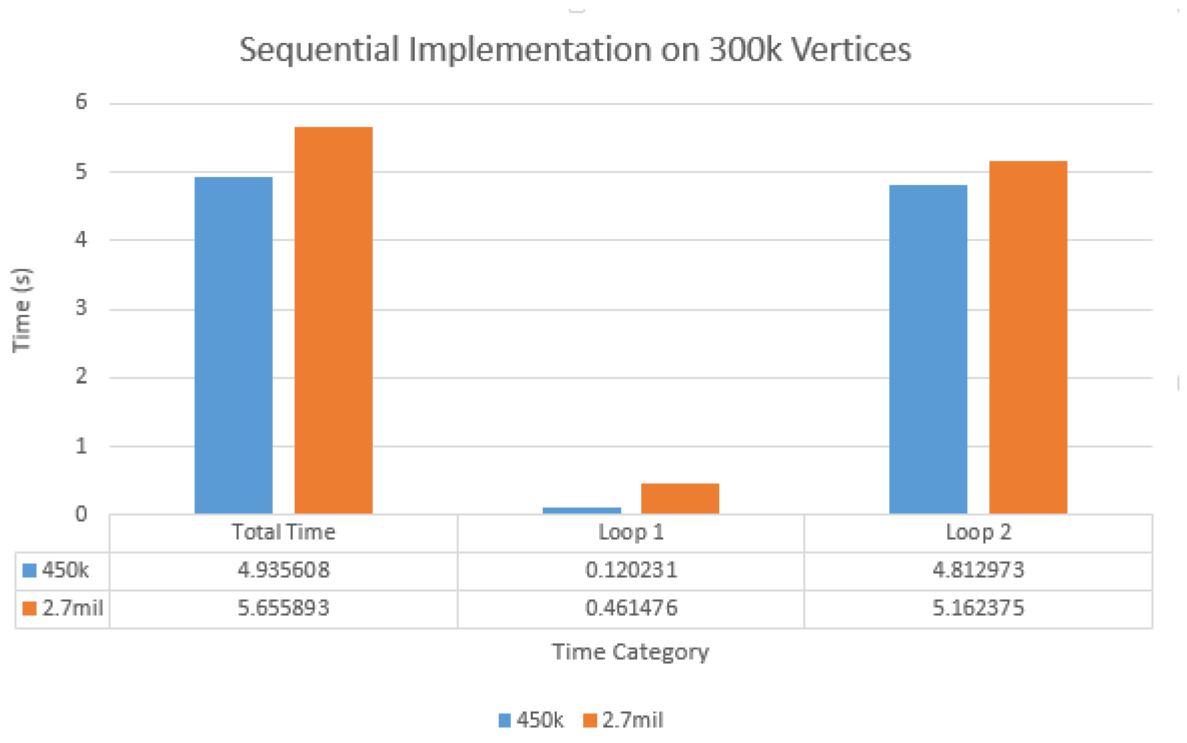
We do our testing on the gates machine servers. These servers use two threads per core and have 8 cores per machine for a total of up to 16 execution contexts

We then do some initial testing to get a sense of what can or should be parallelized and/or needs improvements on performance. The program has two main for loops, one that iterates over all edges and one that iterates over all vertices. Since supposedly there are at least as many nodes as edges in the graph (but in our case, we have much more edges than vertices), the 1st body loop that iterates over all edges should take up the most of the computation time. Both loop bodies do the same find operations and skip the iteration on the same condition. The 2nd loop that iterates over vertices also includes a union operation, however the union operation only incurs an two extra find operations for a total of 4 per loop iteration. Because of this, an initial cursory analysis leads us to believe that the 1st loop body would take (m/2n) more time than the other body (I.E. if we had 9x as many edges as nodes, the first loop body would take 4.5x time)

We ran our sequential tests using three different graphs: one with 300,000 nodes and 450,000 edges (sparse graph), one with 300,000 nodes and one with 300,000 nodes and 2,700,000 edges (dense graph). We expected to see at least 4.5x as much time on the 1st body loop than the 2nd body loop. Results from our initial tests are displayed below with all times given in seconds (note that as each graph has 300,000 nodes, we labeled the charts only with number of edges):

Number of edges	Total Time	Loop 1 Run	Loop 2 Run
450,000	5.00852	0.094201	4.91268
	5.12097	0.12392	4.99495
	4.48049	0.0968737	4.38179
	5.13245	0.165928	4.96247
2,700,000	5.24645	0.685731	4.55676
	5.01382	0.469891	4.54111
	6.7904	0.323842	6.46456
	5.5729	0.366441	5.08707

We provide a graph of average running times below:



	Total Time	Loop 1	Loop 2
🟦 450k	4.935608	0.120231	4.812973
🟧 2.7mil	5.655893	0.461476	5.162375

As the results above show, surprisingly the 2nd body loop takes a lot more time to complete than the 1st. On average over 95% of the computation time took place in the 2nd body loop. In fact, even if the edge count changed, the program would take around the same amount of time to complete. While a much larger increase in edge count would also increase in a larger increase in run time of loop 1, the 2nd body loop still overshadows the 1st body loop in terms of runtime. After some further inspection of the runtimes for each loop, the first run through of the 2nd body loop would call union on every vertex which caused a large jump in runtime for specifically that first iteration loop. The next subsequent finds for the rest of the loops would then cause a lot more loop iteration skips as the program would detect the edge would be of the same component. Because of this we will try focus on parallelizing the 2nd body loop even though the 1st body loop does include a lot of inherent parallelism. We believe that the overhead from parallelizing it will overshadow the benefits of parallel computation.

We use these results as a reasonable baseline for our parallelization. On a cursory test with 50k vertices and the same amount of edges, we confirm our theory that boruvka's at least sequentially is bounded by the number of vertices as the runtime is significantly faster. We only use the graph with 300,000 vertices and 450,000 edges as our test graph since the number of edges doesn't matter.

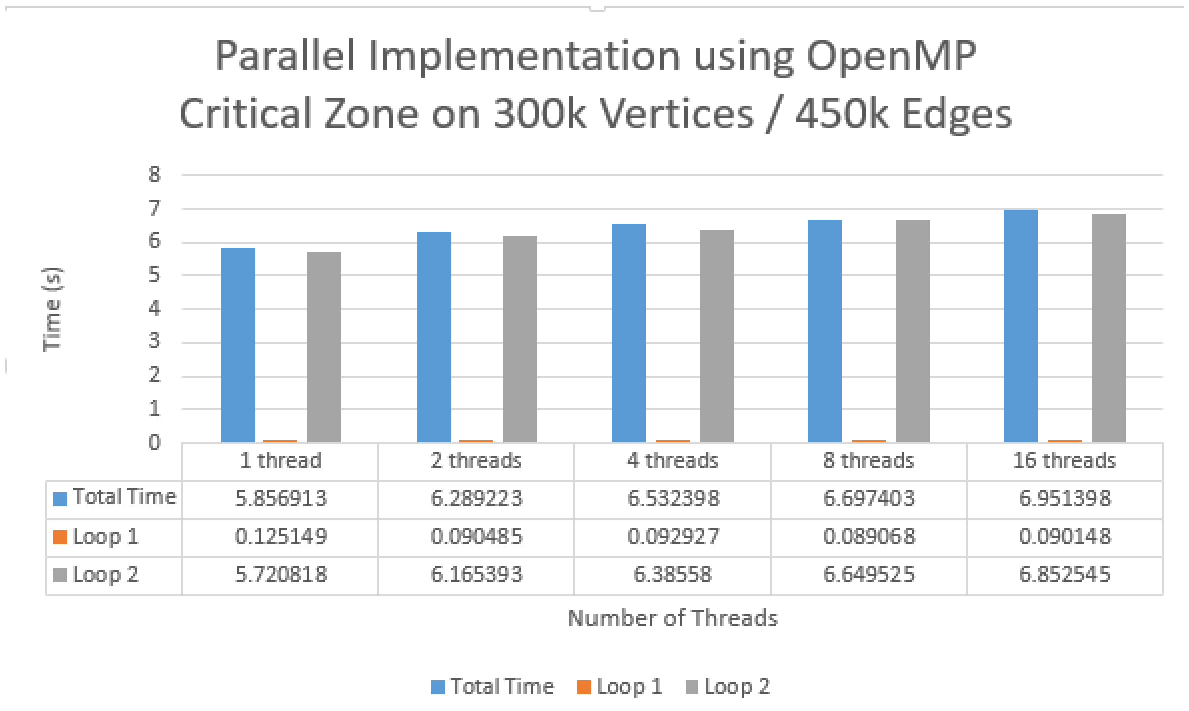
5 Parallelizing Borvuka's

From our results from our initial sequential testing, we think we should focus on parallelizing the 2nd body loop as the most amount of performance improvement could be gotten. Although it should be relatively easy to completely parallelize the first body loop, the amount of time it runs in comparison to the 2nd body loop is relatively miniscule. We note the overhead from parallelizing it and simply ignore it for now. An easy way to parallelize it would to be add a parallel for loop over the entire thing. This has no concurrency issues because we do not modify the union find structure.

The 2nd body is a little bit trickier to parallelize. The union operation for the data structure should be atomic in nature; if we try to union to things at once, it must decide which one becomes the "head" in an atomic manner. For sparse graphs with high connected component counts, it is generally easier to avoid contention for the same vertices to contract into since usually selecting two edges from two threads would have source and destination vertices of different components; this is where most of the parallelizability comes from. If they were the same source and destination they would contend to access the same part of the union find. Although it is unlikely for this to happen for sparse graphs, for denser graphs this is more likely. For this we utilize a critical zone to maintain atomicity, and the results for 300,000 vertices and 450,000 edges run on various threads are displayed below:

Number of Threads	Total Time	Loop 1 Run	Loop 2 Run
1 thread	5.9006	0.110089	5.78158
	5.83482	0.132516	5.69042
	5.85043	0.127378	5.71121
	5.8418	0.130612	5.70006
2 threads	6.30649	0.09374	6.20462
	6.28579	0.0907588	6.0858
	6.20054	0.099553	6.09229
	6.36407	0.0778897	6.27886
4 threads	6.49503	0.0867478	6.3997
	6.55655	0.0904758	6.25713
	6.59143	0.115285	6.46597
	6.48658	0.0792001	6.41952
8 threads	6.75922	0.0891706	6.86108
	6.65461	0.0806528	6.565
	6.65505	0.0955912	6.55047
	6.72073	0.0908571	6.62155
16 threads	7.01303	0.0936533	6.91042
	6.80756	0.0915825	6.70678
	6.92656	0.090138	6.82788
	7.05844	0.0852177	6.9651

The graph of the average run times for each of these threads is given below:



	1 thread	2 threads	4 threads	8 threads	16 threads
■ Total Time	5.856913	6.289223	6.532398	6.697403	6.951398
■ Loop 1	0.125149	0.090485	0.092927	0.089068	0.090148
■ Loop 2	5.720818	6.165393	6.38558	6.649525	6.852545

We would expect a 2× between 1 thread and 2 threads, but instead, there is a slowdown when utilizing more and more threads, where 1 thread has about a 6 second run time, followed by 6.3 seconds, 6.5 seconds, 6.7 seconds, and 7 seconds for 2, 4, 8, and 16 threads, respectively. Upon further analysis, it is important to notice, however, the amount of parallelizability in this particular portion of the code. While parallelizing over iterations of a for loop can be highly effective in improving running times, the contents of the loop body for our Boruvka's algorithm implementation involve the atomic union of sets, which take up most of the running time of the algorithm. We place this content in a `#pragma omp critical` code zone, but in doing so, we effectively place the bulk of the code run time into a single threaded operation, which effectively makes the code run effectively sequentially. To that end, it appears to make sense that with an increasing number of threads, we obtain a slowdown by utilizing more and more cores because of overhead of communication on inherently sequential code.

5.1 Issues with atomicity

We found that the critical zone needed to be placed around not only the union operation, but also the find operations. This is most likely because doing the find's out of order before the union's complete can affect the overall structure of the data structure. Since the data structure itself is not locked, threads can access all elements of the union find at any time with find operations. If we were to find to in this case find that two vertices are in different components at first but then the union operation puts them into the same component from another thread, we would erroneously contract an edge into another vertex of the same component giving us wrong answers. Thus we end up putting the entire block into critical zone. For some small optimizations we created two arrays of size num vertices to keep track of the additional MST weight and num components decreased in this iteration of the 2nd body loop so our critical section just composes of find and unions. We then add the MST weight and num components after the body loop finishes.

Since our speedup is at best .8x, along with the issues with threads accessing areas of the data structure, we attempt a locking solution.

5.2 Locks

We then try to use a lock to restrict usage of the data structure to specific threads. A lock over the entire union find data structure would probably as inefficient as the critical zone, so we resort to a fine grained lock. We use a hand over hand locking technique to ensure minimum contention for locks. This hand over hand locking is similar to the implementation of fine grained locks on linked lists. Instead of actual pointers to the next node we utilize a parent field to indicate the next index of the array. We have some added complexity when unioning the sets together which can be resolved with good scheduling of locking and unlocking. We use openMP's lock set to do fine grained locking; we create a mutex field for each element in the union find. In the find operation, we lock the initial node, then lock the next lock one if we have not found the correct node, while unlocking the next one after.

5.3 Issues with locks

Unfortunately with this section we had issues with getting a working implementation of fine grained locking up and running. We attempted to try and implement this fine grained locking. For smaller test cases ($\leq$ 1000 vertices) it worked fine. However, for larger graphs we would to run into issues with dead lock. It seems that in later iterations of this algorithm two find operations would collide with each other on the tree, with each trying to acquire a lock on each other's currently locked vertex. Our locking solution can be found in our files. We believe that this implementation would perform much better than the critical zone implementation as multiple threads would be able to access the union find data structure at the same time as long as multiple vertices aren't being unioned together at once. In later iterations of the algorithm, where contention is higher the algorithm would perform around the same.

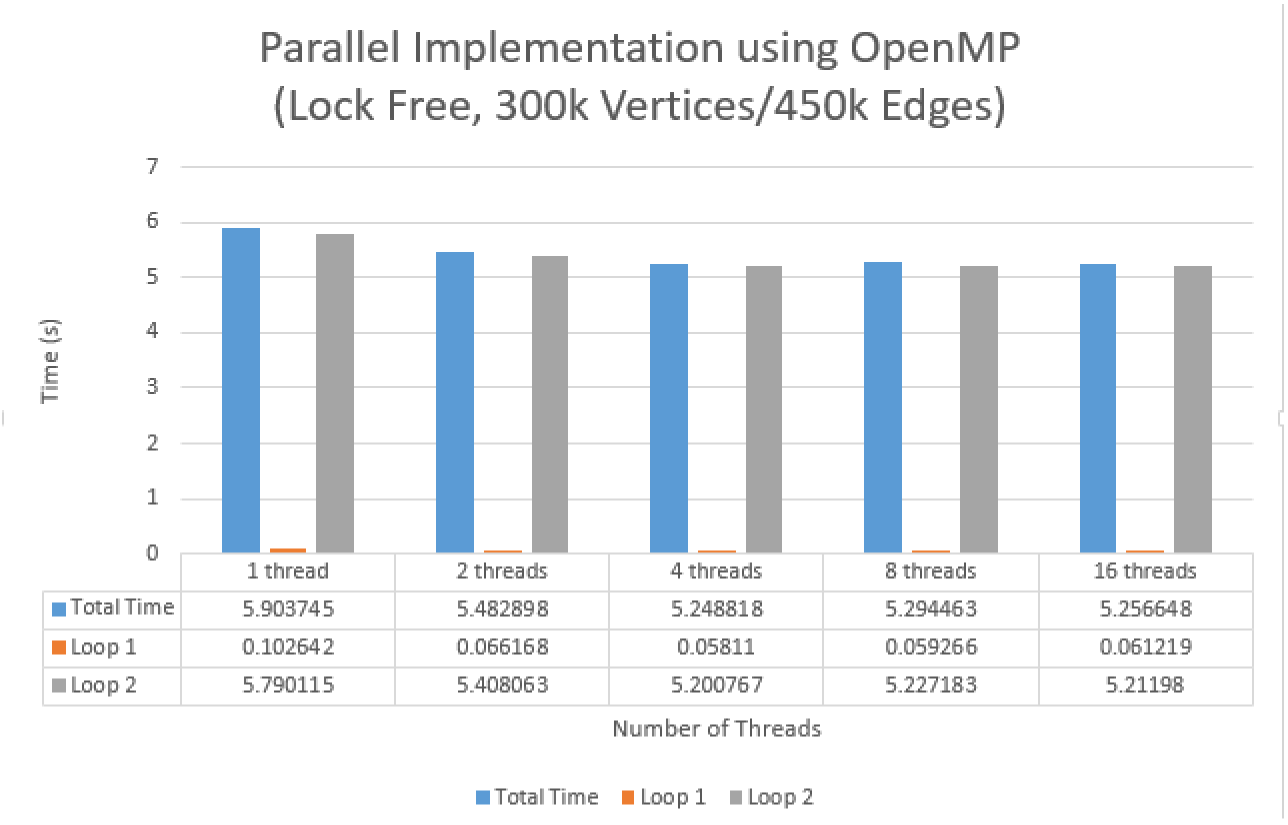
5.4 Lock Free Implementation

Due to the problems with contention, we try a lock free solution which doesn't have the overhead of acquiring locks. We created a lock free solution at first based off this paper [2] that we found. We used the However, we found another implementation that used bitwise operations instead of parent pointers to represent the unionfind operations which proved to be much more efficient and based off the same paper [3]. In essence, both used compare and swap operations to test if the parent values had changed at anytime during the find and union operations. If it had, the operation would be aborted and the process would restart. It also included an atomic same operation, which atomically checked if two vertices were of the same component. This is important because if we checked for the same component iteratively, whether or not two vertices were of the same component could be modified some other thread.

A graph of the average running times is provided below:

[2]http://dl.acm.org/citation.cfm?id=103458
[3]https://github.com/wjakob/dset

Number of Threads	Total Time	Loop 1 Run	Loop 2 Run
1 thread	5.90921	0.0949665	5.80388
	5.67295	0.138355	5.52037
	5.95861	0.0829766	5.866
	6.07421	0.0942703	5.97021
2 threads	5.59377	0.0673201	5.5177
	5.72448	0.0673311	5.64863
	5.46747	0.0641325	5.39486
	5.14587	0.0658899	5.07106
4 threads	5.2791	0.0596276	5.21129
	5.25133	0.0578424	5.198477
	5.24992	0.0566674	5.198506
	5.21492	0.0583029	5.194796
8 threads	5.29141	0.0600561	5.22334
	5.27293	0.0589659	5.20598
	5.33466	0.0588289	5.26783
	5.27885	0.0592113	5.21158
16 threads	5.27764	0.0618223	5.20717
	5.32692	0.0630646	5.25495
	5.23192	0.0577104	5.26624
	5.19011	0.0622781	5.11956



	1 thread	2 threads	4 threads	8 threads	16 threads
Total Time	5.903745	5.482898	5.248818	5.294463	5.256648
Loop 1	0.102642	0.066168	0.05811	0.059266	0.061219
Loop 2	5.790115	5.408063	5.200767	5.227183	5.21198

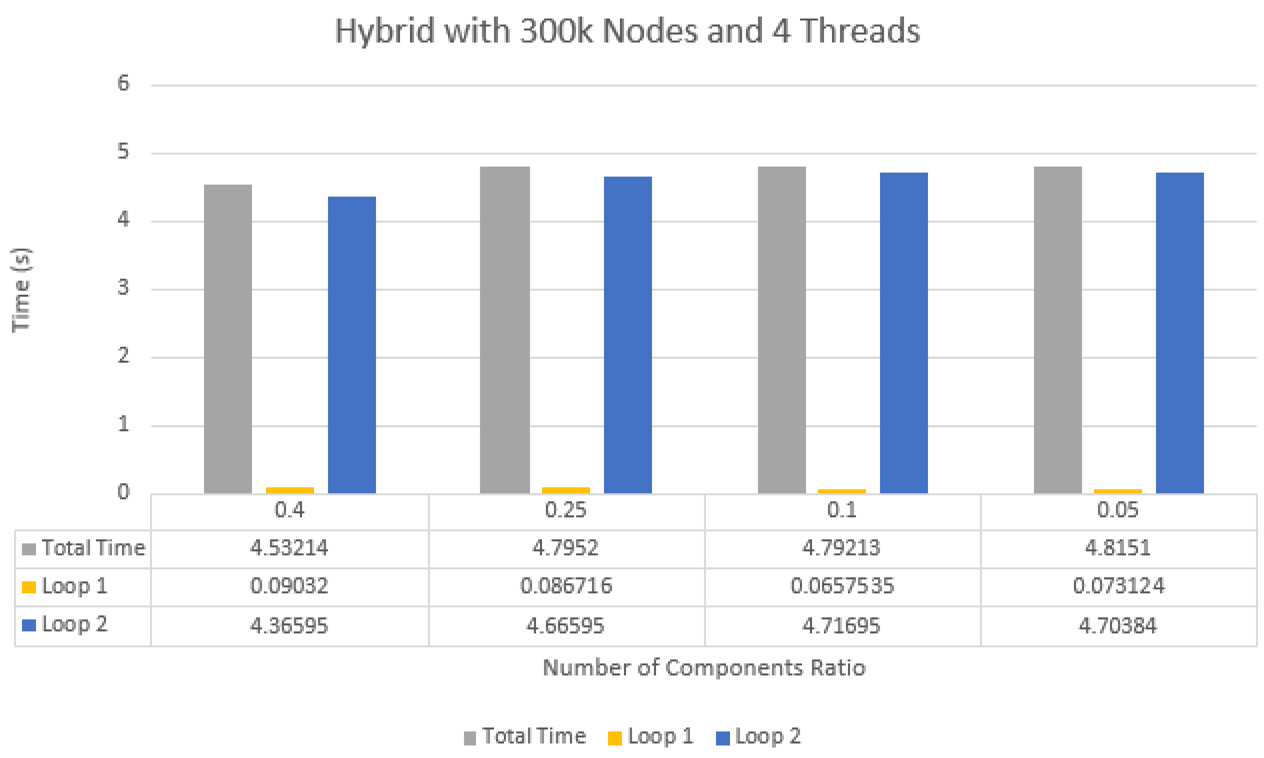
For our lock free implementation, for all threads we do have a slow down from our sequential version. The time taken to run also seems to plateau around 4 threads, while 1 and 2 threads are slower than the rest. However, it does appear that we have a speedup of about 1.1× between 1 thread and 2 threads. Looking more closely at individual loops the first couple of iterations do run more quickly than before, this is probably due to the sparsity of the graph initially allowing for less contention when contracting the graph. However, with later iterations, the implementation seems to slow down, probably from the extra overhead and high contention of accessing the unionfind data structure from multiple threads. With this in mind we move on to our next approach.

5.5 A hybrid approach

We noticed in later iterations of the algorithm, if multiple threads required to union at the same time with fewer effective vertices (components), they are more likely contend for the same element with in the unionfind where one thread stalls for the other. We see this in the run time for specific threads being much larger than they need to be (hence the stalling). Because of this, multiple threads seemed to be unnecessary, creating overhead for the program. Thus we elect to dynamically determine how many threads we use. This idea is similar from assignment 3's hybrid BFS where we would choose either top down or bottom up based off the size of the frontier.

By our results from our lock-free algorithm, we think 4 threads is as much speed up as we can get. So we choose a threshold to utilize change the number of threads running to 1. We base our threshold as a ratio between the number of components and the total number of vertices. We use this is as a metric because on a smaller number of components relative to the size of the graph, there is more likely to have contention. Here are our results for running on a graph on 300k nodes with 4 threads using a lock free algorithm.

Number of Components Ratio	Total Time	Loop 1 Run	Loop 2 Run
0.4	4.53214	0.09032	4.36595
0.25	4.7952	0.086716	4.66595
0.1	4.79213	0.0657535	4.71695
0.05	4.8151	0.073124	4.70384



	0.4	0.25	0.1	0.05
Total Time	4.53214	4.7952	4.79213	4.8151
Loop 1	0.09032	0.086716	0.0657535	0.073124
Loop 2	4.36595	4.66595	4.71695	4.70384

We see that a threshold ratio of around .4 provides the greatest speedup of about 1.15x. The rest of the speed up ratios are similar. At this point the sequential algorithm is faster than the parallel implementation. This is an unexpectedly high for a threshold ratio; only slightly more than half the vertices being contracted on provided a maximum speed up. This is probably due to the parallel algorithm being the most efficient when there are numerous components. The numerous components leads to less contention between finding and unioning vertices while the sequential algorithm is effecient at lower component counts.

6 Conclusion

In the end we were unable to make any meaningful speed up. Our best was with a hybrid approach with 1.15x speed up. Parallelizing this proved to be more difficult than expected since parallelizing over edges is generally easier than vertices. We did not expect the runtime of the edge loop to be less than the vertices. Because of this all of our efforts focused in on parallelizing the vertex body loop. Although there was initially alot of parallelism (no contention between threads), the graph getting smaller created more contention. We also note the variance in testing, as we did not get as consistent numbers as we liked. We simply took the median results which were more clustered in a certain area. The resources used are cited in footnotes through out this document.

Each person did an equal amount of work

git link: https://bitbucket.org/jwidjaja1/15418finalproject