Milestone Report: Parallel MST Algorithm Project

Mengrou Shou (mshou)
Jacky He (junchenh)

Revised Schedule

Date	Deliverable
April 16	• Initial fast sequential MST complete • CUDA approach complete
April 19	• OpenMP approach complete • MPI approach complete
April 21	Benchmarking for OpenMP, MPI and CUDA approach
April 24	Improve speedup for the 3 approaches
April 27	Explore CuGraph + GraphLab
April 30	Heterogeneous parallelism approach tuning complete
May 3	Final Report complete
May 4	Poster complete

Work Completed

On the infrastructure side, so far, we've completed a fast sequential MST algorithm using Kruskal's with disjoint-set union. This algorithm would be what we benchmark all our parallel algorithms against. Along with the sequential algorithm, we've set up a modular benchmarking and testing suite that makes it easy to implement and benchmark different parallel algorithms without having to rewrite all the benchmarking code. The graphs we're testing our algorithms are currently randomly generated via a script and are divided into "basic", "sparse" and "dense" graphs. The "basic" test suites are small graphs for checking the correctness of the algorithm. The "sparse" test suite tests large sparse graphs, and the "dense" test suite tests large dense graphs.

In terms of parallel algorithms, we've completed our initial implementation using CUDA and MPI. Our CUDA approach happens to also make use of a lock-free disjoint set data structure and many atomic instructions to increase concurrency. The resulting speedup is better on dense graphs than on sparse graphs, and is, in general, better on graphs with larger size due to the overhead of starting CUDA kernels.

The MPI approach currently lets every worker node make a copy of the entire graph, partition the vertices among the worker nodes, let each node compute the minimum edge out of every disjoint set on that worker node, and perform an AllReduce to get the minimum edge out of every disjointed set for the entire graph. Each node then adds those edges to their local MST separately and repeats the process. This approach turned out to be not storage efficient at all (and we quickly run the computer out of memory on large graphs), so we'll have to explore alternative approaches.

Evaluation of Progress

There have been some deviations from the original schedule. In particular, the shared address space and MPI has been swapped in the schedule. Nonetheless, we believe that we will still be able to produce the deliverables up to exploring CuGraph and GraphLab. However, we thought that exploring heterogenous approaches should now be a stretch goal, so as to factor in time for improving the three approaches (CUDA, MPI, OpenMP). The following is a list of goals we plan to hit for the poster session:

Minimum goals

- CUDA approach with benchmarking and analysis
- MPI approach with benchmarking and analysis
- OpenMP approach with benchmarking and analysis

Target goals

- Applying graph DSLs (CUGraph, GraphLab) to the MST problem

Stretch goals / "Nice to have"

- Heterogenous approaches i.e. combining OpenMP and MPI

Preliminary Results

CUDA Results:

Randomly Generated Graph	**Computation Time (ms)**	**Computation Speedup (relative to fast sequential Kruskal's algorithm)**
dense-extreme (8,000 nodes and 28,793,554 edges)	108.60	21.28
dense-large (5,000 nodes and 11,245,624 edges	40.50	19.98

dense-medium (1,000 nodes and 449,576 edges)	2.33	10.68
dense-small (200 nodes and 17,847 edges)	0.29	2.65
sparse-extreme (20,000,000 nodes and 29,998,467 edges)	502.67	12.04
sparse-large (10,000,000 nodes and 15,000,571 edges)	228.47	11.10
sparse-medium (1,000,000 nodes and 1,501,216 edges)	13.15	9.40
sparse-small (100,000 nodes and 149,960 edges)	0.93	10.55

Here are the raw terminal outputs:

```
---------------------------
Testing Suite: dense
Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/extreme.txt"
Verification Succeeded!
Initialization (ms): 69.71918
Computation (ms): 108.601683
Total (ms): 178.320863
Computation Speedup: 21.28432169
Total Speedup: 13.93592144

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/large.txt"
Verification Succeeded!
Initialization (ms): 27.949016
Computation (ms): 40.501619
Total (ms): 68.450635
Computation Speedup: 19.9792953
Total Speedup: 12.06661617

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/medium.txt"
Verification Succeeded!
Initialization (ms): 1.659557
Computation (ms): 2.333943
Total (ms): 3.9935
Computation Speedup: 10.68034823
Total Speedup: 6.343671717

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/small.txt"
Verification Succeeded!
Initialization (ms): 0.187286
Computation (ms): 0.286728
Total (ms): 0.474014
Computation Speedup: 2.64793114
Total Speedup: 1.636932664
```

```
Testing Suite: sparse
Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/extreme.txt"
Verification Succeeded!
Initialization (ms): 167.973669
Computation (ms): 502.669035
Total (ms): 670.642704
Computation Speedup: 12.04424638
Total Speedup: 9.366112516

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/large.txt"
Verification Succeeded!
Initialization (ms): 54.110293
Computation (ms): 228.47394
Total (ms): 282.584233
Computation Speedup: 11.10528621
Total Speedup: 9.116259558

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/medium.txt"
Verification Succeeded!
Initialization (ms): 5.96319
Computation (ms): 13.153084
Total (ms): 19.116274
Computation Speedup: 9.403812292
Total Speedup: 6.667658457

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/small.txt"
Verification Succeeded!
Initialization (ms): 0.743391
Computation (ms): 0.926244
Total (ms): 1.669635
Computation Speedup: 10.54645968
Total Speedup: 5.938188287
```

MPI Results:

Here, we report the preliminary results using only 4 compute nodes because we run out of memory if we go to 8 compute nodes

Randomly Generated Graph	**Computation Time (ms)**	**Computation Speedup (relative to fast sequential Kruskal's algorithm) with 4 compute nodes**
dense-extreme (8,000 nodes and 28,793,554 edges)	1809.92	1.33
dense-large (5,000 nodes and 11,245,624 edges	605.80	3.61
dense-medium (1,000 nodes and 449,576 edges)	29.69	0.89
dense-small (200 nodes and 17,847 edges)	7.41	0.16

Here's the raw terminal output

```
---------------------------
Testing Suite: dense
Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/extreme.txt"
Verification Succeeded!
Initialization (ms): 12437.37394
Computation (ms): 1809.923603
Total (ms): 14247.29754
Computation Speedup: 1.333353128
Total Speedup: 0.1822967851

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/large.txt"
Verification Succeeded!
Initialization (ms): 6861.859465
Computation (ms): 605.801431
Total (ms): 7467.660896
Computation Speedup: 3.611376053
Total Speedup: 0.4751761144

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/medium.txt"
Verification Succeeded!
Initialization (ms): 54.502446
Computation (ms): 29.685235
Total (ms): 84.187681
Computation Speedup: 0.8852517085
Total Speedup: 0.3170953242

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/dense/small.txt"
Verification Succeeded!
Initialization (ms): 1.732616
Computation (ms): 7.412523
Total (ms): 9.145139
Computation Speedup: 0.159293671
Total Speedup: 0.1719245601
```

And here, we report the preliminary results using only 2 compute nodes for the sparse graphs because even 4 compute nodes will run out of memory…

Randomly Generated Graph	**Computation Time (ms)**	**Computation Speedup (relative to fast sequential Kruskal's algorithm) with 4 compute nodes**
sparse-extreme (20,000,000 nodes and 29,998,467 edges)	22695.16	0.27
sparse-large (10,000,000 nodes and 15,000,571 edges)	8527.615283	0.29
sparse-medium (1,000,000 nodes and 1,501,216 edges)	546.765582	0.22
sparse-small (100,000 nodes and 149,960 edges)	95.444112	0.10

Here are the raw terminal outputs:

```
Testing Suite: sparse
Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/extreme.txt"
Verification Succeeded!
Initialization (ms): 17881.26728
Computation (ms): 22695.16462
Total (ms): 40576.4319
Computation Speedup: 0.2661015028
Total Speedup: 0.1541509709

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/large.txt"
Verification Succeeded!
Initialization (ms): 8883.700869
Computation (ms): 8527.615283
Total (ms): 17411.31615
Computation Speedup: 0.2952009834
Total Speedup: 0.1462355493

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/medium.txt"
Verification Succeeded!
Initialization (ms): 717.949676
Computation (ms): 546.765582
Total (ms): 1264.715258
Computation Speedup: 0.2255555014
Total Speedup: 0.09967004763

Running test:"/afs/andrew.cmu.edu/usr15/junchenh/private/15418/par-mst/tests/sparse/small.txt"
Verification Succeeded!
Initialization (ms): 31.78904
Computation (ms): 95.444112
Total (ms): 127.233152
Computation Speedup: 0.104226356
Total Speedup: 0.07937396694
```

Most Concerning Issues

MPI seems very tricky to parallelize because of how large our graphs are. To improve the performance from its current state, we'll need to first resolve the storage issue, and also significantly cut down communication and processing. Many approaches we thought of have storage as a bottleneck (either broadcasting an entire disjoint set or the entire graph). The only thing we can think of right now is using the Gallager, Humblet, and Spira (GHS) algorithm. This is an entirely distributed MST algorithm that works by treating each node in the graph as a compute node. For our purposes, we'll have to make one compute node handle multiple graph nodes. Even though we know of the existence of such an algorithm, it'll be tricky to code up.

Another open question is how much should we really care about initialization cost as a performance metric considering we're measuring against a fast sequential baseline (which has minimal initialization overhead). MPI will pretty much always have a higher initialization cost than the sequential baseline, but that's just due to the nature of that programming paradigm. Perhaps it makes sense to only worry about initialization cost if we're measuring speedup relative to the same paradigm, but not worry about it when comparing across different paradigms.

As another side note, we'll also need to change the graph representation of our test cases to some more compressed representation, most likely the Compressed Sparse Row(CSR) format, because our AFS space is limited.