Floyd-Warshall vs Johnson: Solving All Pairs Shortest Paths in Parallel

15-418 Checkpoint 1
Jared Moore (jmoore1) and Josh Kalapos (jkalapos)

Progress

We wanted to accomplish the following for Checkpoint 1:

- Graph generation code (supporting density parameter) finished.
    - For Floyd-Warshall, this was pretty straightforward and is done in floyd_warshall.h
    - For Johnson's Algorithm, graph generation was slightly more complicated as we wanted to use a edge list representation of a graph while making the random graphs match the exact nature of the adjacency matrix graphs given to Floyd-Warshall. The creation of these graphs is done in johnson.h
- Benchmarking wrapper code finished.
    - We wrote our benchmarking code in C++ (main.cpp) and it produces the pretty graphs seen below. We are in the progress of migrating this code to a Python script to suit benchmarking multiple executables (sequential, OpenMP, CUDA).
- Debugging macros made and Makefile configured for C++ and OpenMP implementation.
    - Debugging marcos turned out not to be needed.
    - Makefile now produces two executables: one with OpenMP and one without.
- Regression and correctness checking code finished.
    - Correctness tests implemented in C++ and includes a solution cache for large graphs.
- Sequential implementations of Floyd-Warshall's Algorithm and Johnson's Algorithm finished.
    - Both have been implemented and are working.
    - The sequential version of Johnson's Algorithm uses Boost's implementation .

You can see our progress in at https://github.com/moorejs/418-final.

We believe we will be able to meet our next checkpoint goal, with finished parallel implementations of both graph algorithms in OpenMP.

Completed Work

So far, our most significant work is the completion of the sequential implementations of Floyd-Warshall's Algorithm and Johnson's Algorithm. The implementation of Floyd-Warshall's algorithm is our own because we wanted to optimize blocking of the adjacency matrix representation of the graph for better cache locality, while Johnson's sequential algorithm uses the Boost Graph Library.

These two sequential implementations are going to be the basis for our benchmarks against faster, parallel implementations. Though the parallel code has not been written yet, we have completed the benchmarking wrappers that will measure the speedup from parallelism. Benchmarking also takes in an additional flag that runs correctness checks after taking measurements, so that we know if any of our changes compromise the correctness. Additionally, regression testing can be done on a case basis through multiple flags that allow for changing the expected density of edges, and the number of vertices. Also in preparation for the parallel implementations, we have configured the makefile to support a separate OpenMP executable.

Challenges

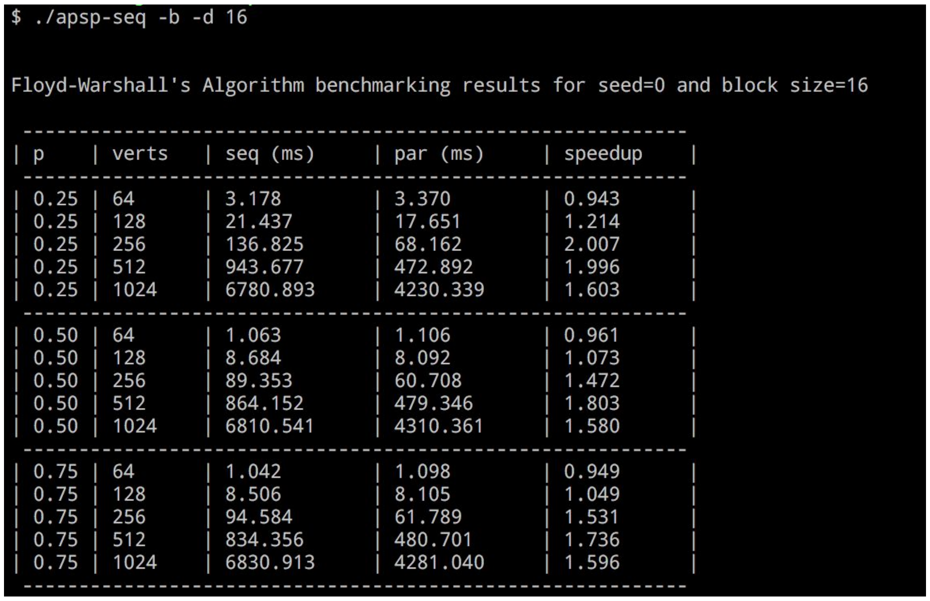
One problem we have not solved is how to most effectively generate random graphs with negative edge weights. We want to verify our solution works and can do this with only positive weights, but the algorithms are supposed to support negative weight edges as well. However, they do not support graphs with negative cycles. When randomly generating a graph, it is hard to randomly place negative edges without making a cycle, and trying repeatedly until you get a good result that has no negative cycles requires a lot of wasted time and a working negative cycle detection algorithm. One workaround could be to distribute negative edges very carefully in a method that guarantees there are no negative cycles, which I believe is possible but may produce less interesting or less unique graphs.

Another problem is the uncertainty surrounding using CUDA on the Latedays machines. We know the machines have a NVIDIA Titan X GPU but I could not find a NVIDIA compiler on the machines. We may be able to compile for the Titan X architecture on the GHC machines and then transfer this executable to the Latedays machines, but I am uncertain about this.

Our third concern lies in Johnson's Algorithm's performance and implementation. From our preliminary results Floyd-Warshall beats Johnson's in many aspects, so we might have to investigate further about which metrics we should focus on comparing Johnson's Algorithm and Floyd Warshall's Algorithm. Additionally, implementation could be an issue as our sequential version of Johnson’s uses boost library. So then, in the coming parallel implementation of Johnson’s, we will have to decide how much of boost’s library we will want to use. Boost provides a Dijkstra function and a Bellman-Ford function, but utilizing them inside of our own Johnson’s might be difficult and it would be simpler to use more basic boost graph functions and fibonacci heaps to implement more parts ourselves. Time spent on these implementations might interfere with the spirit of focusing on parallelism over sequential tasks but might ultimately be necessary.

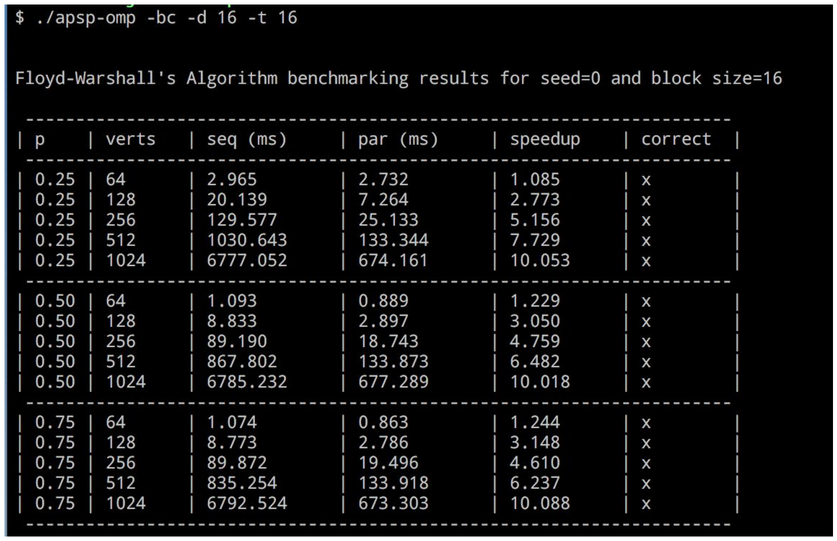
Floyd-Warshall Results

Thus far, we have implemented the most basic version of Floyd-Warshall for correctness checking, the blocked version of Floyd-Warshall, and a basic parallel implementation on top of the blocked Floyd-Warshall. Below, we compare two sequential versions (mislabelled parallel and sequential when it’s really sequential vs sequential) with a given block size, and the (slower) sequential version compared to the basic parallel implementation in OpenMP.



```
$ ./apsp-seq -b -d 16

Floyd-Warshall's Algorithm benchmarking results for seed=0 and block size=16

 ----------------------------------------------------------
| p    | verts   | seq (ms)     | par (ms)     | speedup    |
 ----------------------------------------------------------
| 0.25 | 64      | 3.178        | 3.370        | 0.943      |
| 0.25 | 128     | 21.437       | 17.651       | 1.214      |
| 0.25 | 256     | 136.825      | 68.162       | 2.007      |
| 0.25 | 512     | 943.677      | 472.892      | 1.996      |
| 0.25 | 1024    | 6780.893     | 4230.339     | 1.603      |
 ----------------------------------------------------------
| 0.50 | 64      | 1.063        | 1.106        | 0.961      |
| 0.50 | 128     | 8.684        | 8.092        | 1.073      |
| 0.50 | 256     | 89.353       | 60.708       | 1.472      |
| 0.50 | 512     | 864.152      | 479.346      | 1.803      |
| 0.50 | 1024    | 6810.541     | 4310.361     | 1.580      |
 ----------------------------------------------------------
| 0.75 | 64      | 1.042        | 1.098        | 0.949      |
| 0.75 | 128     | 8.506        | 8.105        | 1.049      |
| 0.75 | 256     | 94.584       | 61.789       | 1.531      |
| 0.75 | 512     | 834.356      | 480.701      | 1.736      |
| 0.75 | 1024    | 6830.913     | 4281.040     | 1.596      |
 ----------------------------------------------------------
```

The non-blocked sequential code is compared to the blocked sequential code, receiving **~1.4x** speedup

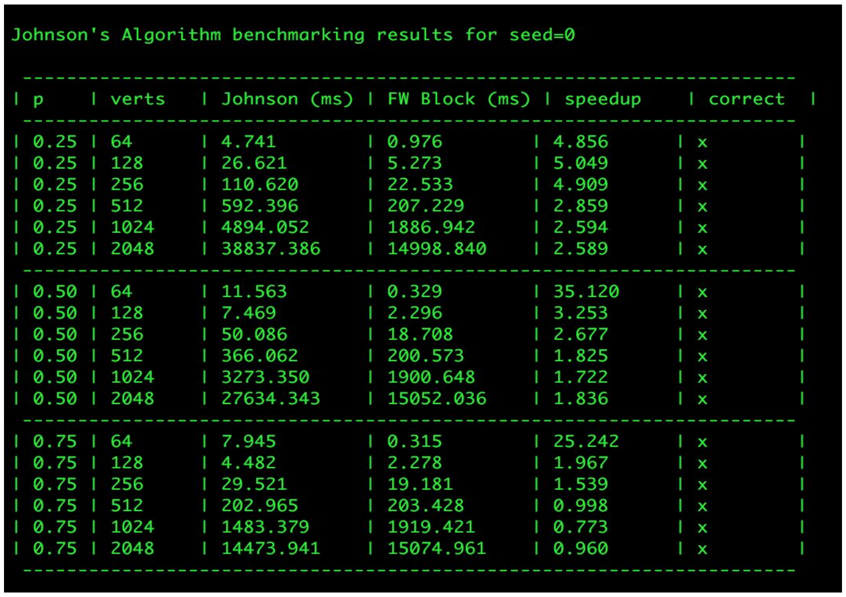


```
$ ./apsp-omp -bc -d 16 -t 16

Floyd-Warshall's Algorithm benchmarking results for seed=0 and block size=16

 -------------------------------------------------------------------
| p    | verts    | seq (ms)     | par (ms)     | speedup     | correct  |
 -------------------------------------------------------------------
| 0.25 | 64       | 2.965        | 2.732        | 1.085       | x        |
| 0.25 | 128      | 20.139       | 7.264        | 2.773       | x        |
| 0.25 | 256      | 129.577      | 25.133       | 5.156       | x        |
| 0.25 | 512      | 1030.643     | 133.344      | 7.729       | x        |
| 0.25 | 1024     | 6777.052     | 674.161      | 10.053      | x        |
 -------------------------------------------------------------------
| 0.50 | 64       | 1.093        | 0.889        | 1.229       | x        |
| 0.50 | 128      | 8.833        | 2.897        | 3.050       | x        |
| 0.50 | 256      | 89.190       | 18.743       | 4.759       | x        |
| 0.50 | 512      | 867.802      | 133.873      | 6.482       | x        |
| 0.50 | 1024     | 6785.232     | 677.289      | 10.018      | x        |
 -------------------------------------------------------------------
| 0.75 | 64       | 1.074        | 0.863        | 1.244       | x        |
| 0.75 | 128      | 8.773        | 2.786        | 3.148       | x        |
| 0.75 | 256      | 89.872       | 19.496       | 4.610       | x        |
| 0.75 | 512      | 835.254      | 133.918      | 6.237       | x        |
| 0.75 | 1024     | 6792.524     | 673.303      | 10.088      | x        |
 -------------------------------------------------------------------
```

The non-blocked sequential code is compared to the blocked parallel code, receiving **10x** speedup

Johnson's Algorithm Results

So far, our completed sequential Johnson's Algorithm can be compared to Floyd-Warshall's Algorithm with blocking optimization. As our table shows, Floyd-Warshall currently outperforms Johnson's in nearly all cases, though with more vertices Johnson appears to inch toward being competitive with Floyd-Warshall's. What is also interesting to note so far is that Johnson seems to do relatively better with a lower density of edges (reflected by 1-p). This agrees with our earlier prediction in our proposal that Johnson's would perform better with sparse graphs. In the coming weeks we would like to find more results of Johnson's performance vs Floyd-Warshall and its parallel version in even sparser graphs.



```
Johnson's Algorithm benchmarking results for seed=0

 ----------------------------------------------------------------------
| p    | verts    | Johnson (ms) | FW Block (ms) | speedup     | correct  |
 ----------------------------------------------------------------------
| 0.25 | 64       | 4.741        | 0.976         | 4.856       | x        |
| 0.25 | 128      | 26.621       | 5.273         | 5.049       | x        |
| 0.25 | 256      | 110.620      | 22.533        | 4.909       | x        |
| 0.25 | 512      | 592.396      | 207.229       | 2.859       | x        |
| 0.25 | 1024     | 4894.052     | 1886.942      | 2.594       | x        |
| 0.25 | 2048     | 38837.386    | 14998.840     | 2.589       | x        |
 ----------------------------------------------------------------------
| 0.50 | 64       | 11.563       | 0.329         | 35.120      | x        |
| 0.50 | 128      | 7.469        | 2.296         | 3.253       | x        |
| 0.50 | 256      | 50.086       | 18.708        | 2.677       | x        |
| 0.50 | 512      | 366.062      | 200.573       | 1.825       | x        |
| 0.50 | 1024     | 3273.350     | 1900.648      | 1.722       | x        |
| 0.50 | 2048     | 27634.343    | 15052.036     | 1.836       | x        |
 ----------------------------------------------------------------------
| 0.75 | 64       | 7.945        | 0.315         | 25.242      | x        |
| 0.75 | 128      | 4.482        | 2.278         | 1.967       | x        |
| 0.75 | 256      | 29.521       | 19.181        | 1.539       | x        |
| 0.75 | 512      | 202.965      | 203.428       | 0.998       | x        |
| 0.75 | 1024     | 1483.379     | 1919.421      | 0.773       | x        |
| 0.75 | 2048     | 14473.941    | 15074.961     | 0.960       | x        |
 ----------------------------------------------------------------------
```

Johnson's Algorithm Performance vs Floyd-Warshall Blocked

State of Deliverables

From our project proposal, we stated that we wanted to investigate the effects of OpenMP and CUDA on the Floyd-Warshall and Johnson algorithms. Though we have yet to implement either algorithm fully in both CUDA and OpenMP, these objectives were not part of our checkpoint 1 goals, so we still believe that we will be able to fully complete our project by the deadline. At the end of the course, we still, as our proposal states, hope to present graphs of the speedup of our different parallel implementations against different varieties of problem sizes and types. Additionally, we hope to see the effective differences of Floyd-Warshall and Johnson's performances under the same constraints.