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

15-418 Checkpoint 2

Jared Moore (jmoore1) and Josh Kalapos (jkalapos)

Progress

We wanted to accomplish the following for Checkpoint 2:

- Floyd-Warshall's Algorithm with OpenMP.
- Johnson's Algorithm with OpenMP.

All of these goals have been completed. We have both an implementation of Floyd-Warshall's Algorithm and Johnson's Algorithm with OpenMP. As we have met our checkpoint, we will continue with our originally planned schedule, which includes in the following week implementing Floyd-Warshall's Algorithm and Johnson's Algorithm with CUDA. However, this does preclude us from further improving our OpenMP algorithms.

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

Our remaining schedule is as follows, split into half-week segments:

May 2nd:

- 1st Half: Floyd-Warshall's Algorithm with CUDA.
- 2nd Half: Johnson's Algorithm with CUDA.

May 7th (Final Deadline):

- 1st Half: Leftover work.
- 1st Half: Parameter tweaking for each implementation.
- 2nd Half: More interesting graphs to benchmark and compare with.
- 2nd Half: Cohesive comparison of all benchmark results.
- 2nd Half: Final report ready, best benchmarks taken.

Completed Work

At this checkpoint, our most significant work was parallelizing our algorithms with OpenMP. Both implementations were done by our group though the implementation of Johnson's Algorithm uses the Boost Graph Library's Dijkstra function as a helper.

These two parallel implementations are for comparison against our sequential benchmarks to study the effectiveness of OpenMP in this graph problem scenario. Floyd-Warshall's Algorithm had parallelism achieved by making the blocked loops inside of the main k iterative loop run in parallel. The outermost loop for Floyd-Warshall unfortunately cannot take advantage of parallelism because of the dependencies between iterations. Blocked Floyd-Warshall performed well with added OMP pragmas as better cache locality is still beneficial with more threads.

Our Johnson's Algorithm took a different approach to parallelism because it is more than three for loops and has multiple stages. Different parts of the algorithm, the reweighting of the edges, and the Bellman-Ford algorithm were all parallelized. For the execution of Dijkstra on all sources in the graph, instead of parallelizing dijkstra itself, we chose to instead launch worker threads to dynamically work on building the final output. Dynamic work scheduling actually performed better than static scheduling in this case because Dijkstra the work for different threads is difficult to predict for random graphs. We have collected some preliminary data to support the advantage of dynamic scheduling over static in this case.

Additionally, we were able to improve the reach of our benchmarking with an improved benchmarking program. A lot of the data that gets reported remains the same, but it has made it easier for us to quickly test the impact of our optimizations.

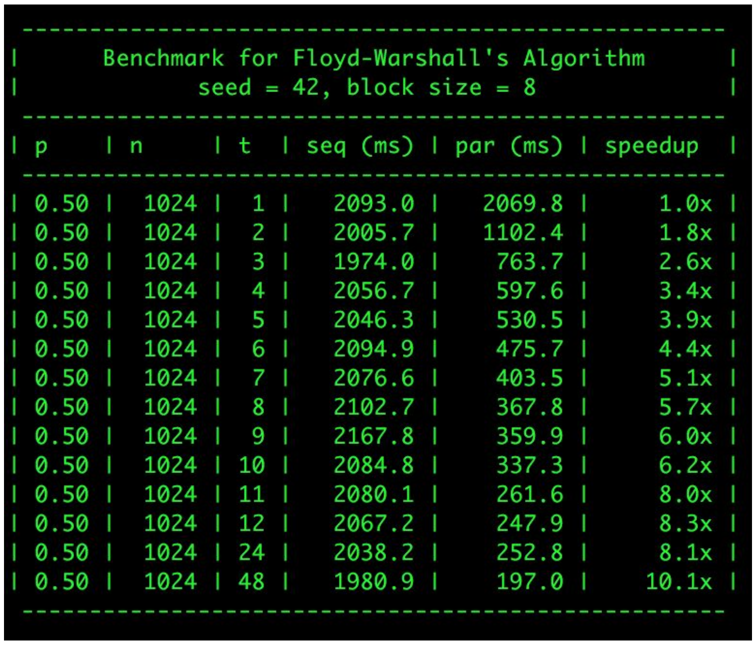
Challenges

The code has still not been compiled or run on the Latedays machines. Our Makefile should be flexible enough that it will not be hard to have it working on Latedays (and we have the assignment Makefiles to draw from), but I am unsure of how we will be building the CUDA code on Latedays. Nevertheless, our very next task is researching and working on our latedays.sh script that will be used for queuing up at Latedays.

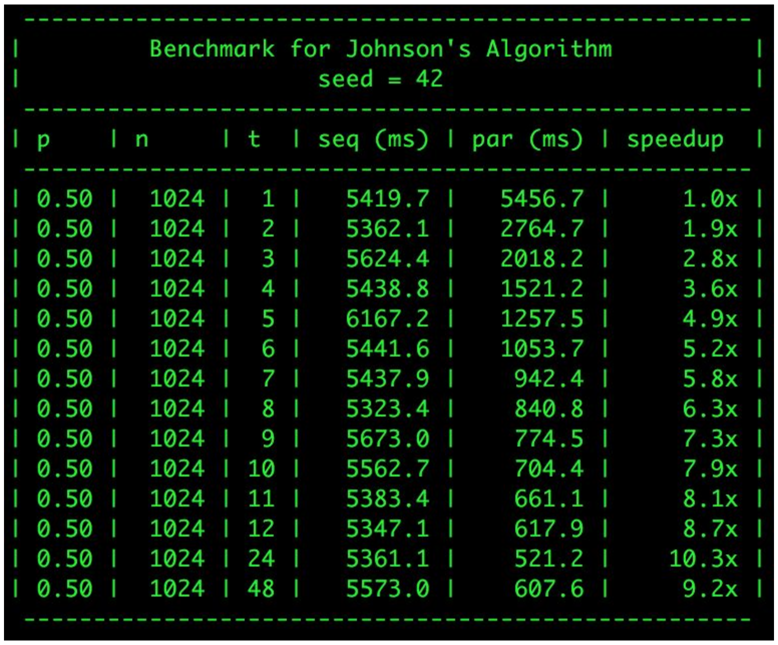
OMP Performance Results

After adding OMP to our implementation of original sequential algorithms, we gathered some preliminary results about the scaling of speedup as we start from running on 1 thread and move up to 48 threads. Although in our original proposal stated that testing was to be done on GHC machines, we decided that it would be beneficial to test on the Unix Andrew machines, as they have 40 cores and can run up to 80 threads. Previously on GHC machines we could only go up to 32 threads with 16 cores. As a result we can potentially run tests on larger graphs. In our preliminary data we kept the graph size constant and simply compared the increase in speedup as we increased cores.

From our two tables, we observe speedup from 1 to 48 threads. Both algorithms seem to max out at around 7-8x speedup with 12+ threads. This result makes sense because of the unavoidable serial portions of any optimized implementation.



```
 -----------------------------------------------------
|      Benchmark for Floyd-Warshall's Algorithm       |
|             seed = 42, block size = 8               |
 -----------------------------------------------------
| p    | n     | t  | seq (ms) | par (ms) | speedup  |
 -----------------------------------------------------
| 0.50 |  1024 |  1 |   2093.0 |   2069.8 |     1.0x |
| 0.50 |  1024 |  2 |   2005.7 |   1102.4 |     1.8x |
| 0.50 |  1024 |  3 |   1974.0 |    763.7 |     2.6x |
| 0.50 |  1024 |  4 |   2056.7 |    597.6 |     3.4x |
| 0.50 |  1024 |  5 |   2046.3 |    530.5 |     3.9x |
| 0.50 |  1024 |  6 |   2094.9 |    475.7 |     4.4x |
| 0.50 |  1024 |  7 |   2076.6 |    403.5 |     5.1x |
| 0.50 |  1024 |  8 |   2102.7 |    367.8 |     5.7x |
| 0.50 |  1024 |  9 |   2167.8 |    359.9 |     6.0x |
| 0.50 |  1024 | 10 |   2084.8 |    337.3 |     6.2x |
| 0.50 |  1024 | 11 |   2080.1 |    261.6 |     8.0x |
| 0.50 |  1024 | 12 |   2067.2 |    247.9 |     8.3x |
| 0.50 |  1024 | 24 |   2038.2 |    252.8 |     8.1x |
| 0.50 |  1024 | 48 |   1980.9 |    197.0 |    10.1x |
 -----------------------------------------------------
```



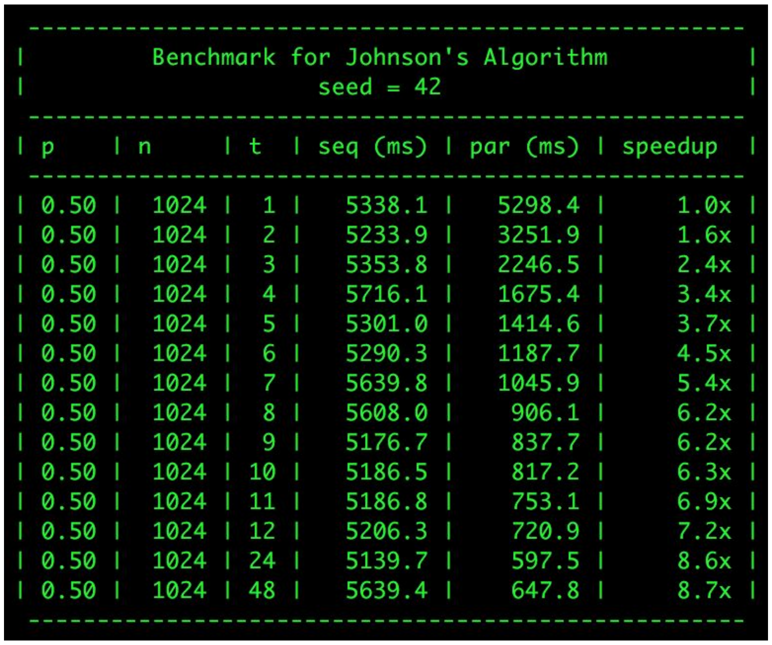
```
 -----------------------------------------------------
|         Benchmark for Johnson's Algorithm           |
|                     seed = 42                       |
 -----------------------------------------------------
| p    | n     | t  | seq (ms) | par (ms) | speedup  |
 -----------------------------------------------------
| 0.50 |  1024 |  1 |   5419.7 |   5456.7 |     1.0x |
| 0.50 |  1024 |  2 |   5362.1 |   2764.7 |     1.9x |
| 0.50 |  1024 |  3 |   5624.4 |   2018.2 |     2.8x |
| 0.50 |  1024 |  4 |   5438.8 |   1521.2 |     3.6x |
| 0.50 |  1024 |  5 |   6167.2 |   1257.5 |     4.9x |
| 0.50 |  1024 |  6 |   5441.6 |   1053.7 |     5.2x |
| 0.50 |  1024 |  7 |   5437.9 |    942.4 |     5.8x |
| 0.50 |  1024 |  8 |   5323.4 |    840.8 |     6.3x |
| 0.50 |  1024 |  9 |   5673.0 |    774.5 |     7.3x |
| 0.50 |  1024 | 10 |   5562.7 |    704.4 |     7.9x |
| 0.50 |  1024 | 11 |   5383.4 |    661.1 |     8.1x |
| 0.50 |  1024 | 12 |   5347.1 |    617.9 |     8.7x |
| 0.50 |  1024 | 24 |   5361.1 |    521.2 |    10.3x |
| 0.50 |  1024 | 48 |   5573.0 |    607.6 |     9.2x |
 -----------------------------------------------------
```

Floyd-Warshall’s Algorithm Thread Performance (Left) and Johnson’s Algorithm Thread Performance (Right)

‘

Johnson's Algorithm (Without dynamic OMP scheduling)

We compared the performance of dynamic vs static scheduling of one of the for loops of Johnson's algorithm. After the algorithm uses Bellman-Ford to reweight all of the edges, it runs Dijkstra on all of the vertices to create an array of distances. With dynamic scheduling we can achieve better work distribution. Below's table shows the lesser performing Johnson's Algorithm without dynamic scheduling.



```
 -------------------------------------------------
|         Benchmark for Johnson's Algorithm         |
|                    seed = 42                      |
 -------------------------------------------------
| p    | n      | t  | seq (ms) | par (ms) | speedup  |
 -------------------------------------------------
| 0.50 |   1024 |  1 |   5338.1 |   5298.4 |     1.0x |
| 0.50 |   1024 |  2 |   5233.9 |   3251.9 |     1.6x |
| 0.50 |   1024 |  3 |   5353.8 |   2246.5 |     2.4x |
| 0.50 |   1024 |  4 |   5716.1 |   1675.4 |     3.4x |
| 0.50 |   1024 |  5 |   5301.0 |   1414.6 |     3.7x |
| 0.50 |   1024 |  6 |   5290.3 |   1187.7 |     4.5x |
| 0.50 |   1024 |  7 |   5639.8 |   1045.9 |     5.4x |
| 0.50 |   1024 |  8 |   5608.0 |    906.1 |     6.2x |
| 0.50 |   1024 |  9 |   5176.7 |    837.7 |     6.2x |
| 0.50 |   1024 | 10 |   5186.5 |    817.2 |     6.3x |
| 0.50 |   1024 | 11 |   5186.8 |    753.1 |     6.9x |
| 0.50 |   1024 | 12 |   5206.3 |    720.9 |     7.2x |
| 0.50 |   1024 | 24 |   5139.7 |    597.5 |     8.6x |
| 0.50 |   1024 | 48 |   5639.4 |    647.8 |     8.7x |
 -------------------------------------------------
```

Johnson's Algorithm without dynamic OMP scheduling

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. So far, we have begun investigating the effectiveness of OpenMP on these two algorithms, and will soon begin implementing and studying the results of a CUDA implementation. So far we have been well on track with our deliverables, and we do not have reason to worry that our future progress will be hindered before the project deadline. At the end of our project, we still hope to present graphs of our accomplished performance gains from OpenMP and CUDA, along with comparisons between different solutions for APSP, and comparisons between different levels of sparse graphs.