Steven Cabral: 6801170
Bronwyn Perry-Huston: 5537360

Final Project Proposal: Parallel Breadth-First Search

For our final project, we plan on implementing a Breadth-First Search(BFS) algorithm that can be done in parallel. Ideally, we want to make an implementation that minimizes bottlenecks in parallelism and minimizes communication. To achieve the parallelism, we plan on implementing the bag structure illustrated in the MIT handout linked in the description. Once the basic algorithm and structure are implemented, we will attempt to do smaller optimizations to the algorithm to improve its efficiency on Triton. The optimizations will aim to maximize the parallel efficiency on Triton by utilizing Triton's specific cache size to minimize memory transfer between caches and cores. We will gather results from increasingly large graph sizes on varying numbers of nodes/processors on Triton, and we will calculate speedup and parallel efficiency to gauge the success of our optimizations.

We plan on using cilk as a language of choice, since it is easy to implement and test between varying numbers of processors and will likely handle the variable number of nodes given by BFS much better than MPI. Of course, as it is the main tool available to us in this course, we will also be doing the implementation on Triton.

Our implementation will include the bag structures illustrated in the paper, since they are a very easy structure to implement in parallel, but are still able to represent the data needed for BFS. The structures are simple to union and split, which will be utilized by parallel processors for easy division of labor and recompilation of results. It also allows for a type of 'work-stealing', similar to what is done in the cilk scheduler itself, which may be one of the methods we try to optimize the algorithm.

After the structures are implemented, the algorithm itself is fairly straightforward - make two separate bags, one for level i and one for level i + 1. Then, we can go across level i distributed across all processors before going onward to level i + i. However, the problem we will face here is having all processors know which data has been explored past this level in an efficient way. Here is where we will want to optimize the program according to the question "where's the data?" A potential optimization is to have some processors 'run ahead' in a smart way to work on the next level of the graph rather than waiting for others to finish the level, but this may also require recomputation. We will have to be careful in determining where the bottlenecks are, and if they are in communication, we will want to utilize other ways to reduce the flow of data. One possibility involves finding a way to "balance" the bags so that work stealing is not done as often, if this ends up being an issue. Another possibility is to locally

explore more nodes than necessary to trade recomputation for communication, though the extra nodes would have to be selected very carefully. We will also try to alter balance the bag sizes with the cache sizes on Triton to try to maximize the efficiency of the algorithm. Ideally by altering the algorithm to chunk the data in a manner that fits with the cache sizes on Triton we will be able to improve the scalability of the parallelized algorithm, especially for large graphs.

One advantage to doing a BFS algorithm is that testing is quite simple and straightforward. Breadth-first search data is easily available in the form of social media data, but it is also easy to generate. The tool linked to on the writeup for Graph 500 Benchmark tests will be very useful in generating large test data to stress-test the program itself. Small tests generated from this tool will likely be useful as well in verifying the correctness of the algorithm as well. If such data proves to be less effective, it is fairly straightforward to generate a graph by adding edges to a randomly-generated BFS tree to create easily-verified test data.Tools such as Cilkscreen and Cilkview will also be invaluable in catching errors involving the scalability of the program, as well as verify no errors are introduced due to parallelization.

In order to quantitatively measure the performance of the parallel algorithm, we will gather data from experiments on graphs of varying sizes and on varying numbers of processors. First, we will construct small graphs by hand to test the initial correctness of our parallelized algorithm. Next, we will use our social media networks to create slightly larger graphs to measure the performance of the algorithm on a reasonably small graph. However, these smaller graphs will not provide enough information on how scalable and efficient our implementation/optimizations are. To measure the scalability and performance of our implementation we will use very large RMAT graphs. We will generate these graphs from the Matlab code provided by the Graph500 website.

The timing information gathered from each graph size will then be used to calculate metrics that will allow us to judge the performance of our parallelized implementation. Each graph will be run on the parallelized, optimized parallelized, and serial algorithms on a set number of processors, which will allow us to calculate the speedup and parallel efficiency for each algorithm. By observing how the speedup and parallel efficiency values change, we will be able to conclude whether parallelization improves the performance of the BFS algorithm and whether or not our optimizations for Triton's cache sizes improved our parallel implementation. We will also use all of the algorithms with a large RMAT graph on a varying number of processors to observe if a plateau occurs in the amount of parallelization.