CS 6230 Project Report

Parallel Computation of Betweenness Centrality

Rui Dai, Sam Olds

May 9, 2017

1 Introduction

In graph theory, centrality indicates the importance of a vertex in the network. This concept is naturally applied on social network analysis. Imagine you are producing a new product and want to find beta users. It's simple to let users with high centrality to use and spread the news to their reachable networks.

There are many different definitions of centrality, eg. degree centrality, closeness centrality and betweenness centrality. In our project, we will use the vertex betweenness centrality.

Betweenness centrality quantifies the number of times a node acts as a bridge along the shortest path between two other nodes. It's not hard to imagine that the computation of centrality is very expensive with all the operations with shortest paths. Our goal of this project is parallel such computation and leverage the technology of MPI/OPENMP we learned in class.

In this report, we will first introduce data preparation and graph representation. Then explain the algorithms we are using with performance experiments.

2 Related Work

Centrality was first introduced as a measure for quantifying the control of a human on the communication between other humans in a social network by Linton Freeman[4] In his conception. Ever since, the concept has drawn many interests in cross-indiscipline area like network analysis and social science.

Betweenness centralities in a graph involve calculating the shortest paths between all pairs of vertices on a graph, which requires $\Theta(V^3)$ time with the FloydWarshall[5] algorithm. On sparse graphs, Johnson's[6] algorithm may be more efficient, taking $O(V^2 \log V + VE)$ time. In the case of unweighted graphs the calculations can be done with Brandes' algorithm[3] which takes $\Theta(VE)$ time. Normally, these algorithms assume that graphs are undirected and connected with the allowance of loops and multiple edges. When specifically dealing with network graphs, often graphs are without loops or multiple edges to maintain simple relationships (where edges represent connections between two people or vertices). In this case, using Brandes' algorithm will divide final centrality scores by 2 to account for each shortest path being counted twice.

Breadth first search is also a great part in centrality computation as for the shortest path part. Parallel BFS algorithms attracts many researchers to explore, among those, [2] proposed a two-direction way to do BFS.

3 Implementation

We implemented our algorithms in C++ using OpenMP and MPI. Our final submission employs an adjacency list to represent the graph and uses a parallel version of the Brandes algorithm for calculating betweenness centrality. We will examine the results more closely later, but it appears that this implementation does not scale well which is most likely due to the fact that the graph is in shared memory. Distributing the graph was a large hurdle we tried to avoid.

3.1 Challenges

The largest challenge was handling a graph distributed across processors. Trying to partition the graph evenly among the workers could be a project on its own. We encountered this problem when breaking up the adjacency matrix among processors. We realized each processor might not have all of the edges attached to each vertex in memory, which would complicate things. This was solved simply by putting the whole graph in shared memory. However, it is believed that this was the cause of the poor scaling performance.

The next challenge that arose was getting the graph data. During initial development, a synthetic graph was used by looping through every vertex for each vertex and creating and edge between them with a 25% probability. Once we were comfortable with our implementation we found a graph of Facebook friend circles used in a Stanford paper [7] with 4039 vertices and 88234 edges. We then found an even larger dataset of Google+ users used in the same paper. This graph has 107614 vertices and 13673453 edges.

Validating the results was another challenge. We spent a fair number of hours finding a way to visually render the graphs. The Graphviz open source project was used, but was somewhat buggy. Ultimately, our validation of the metrics became a visual inspection of the rendered graphs, which is less than ideal. However, the nodes that were marked with higher centrality seemed to be the correct ones.

3.2 Graph Representation

At first, the underlying representation we used was a sparse adjacency matrix. We thought that this would allow for easier parallelization when performing the matrix multiplication. However, it presented a few challenges during implementation. First, an $n \times n$ matrix becomes memory demanding, even though most of the values are simply 0. Second, we used the vertex ids as the indices into the n^2 array, which didn't work for the Google+ dataset. This dataset used 21 digit long values, which means a separate data structure would have been needed to map the vertex ids back to the matrix indices.

Instead, we switched to using an adjacency list to decrease the memory footprint and simplify the vertex id handling. This had a few additional advantages. With an adjacency list, we could easily figure out the total number of vertexes that exist in the graph by just getting the size of the list. This also made it easy to split work among processors by making each processor handle some chunk of the list. This made implementing the Brandes algorithm more straightforward as this was the underlying graph representation used in that paper.

3.3 Algorithms

Effectively, we implemented two different parallel algorithms for calculating the betweenness centrality of a graph. As we mentioned in the introduction section, vertex betweenness centrality is formally defined as:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

3.3.1 Matrix Multiplication

Calculating betweenness centrality can be naively accomplished by calculating the shortest path from every vertex to every other vertex. This was our initial approach as we demonstrated during the presentation. We used a breadth first search technique using matrix multiplication because it was believed that this would make it easy to parallelize. Using matrix multiplication to find the shortest paths behaves as follows:

For our $n \times n$ matrix, we initialize a vector of size n to all 0s. We set some arbitrary value to 1, to be our root node. Then, we simply perform a matrix multiplication with this vector. The resulting vector can be interpreted to mean that every n_i value that is non-zero is directly reachable from the root node. We keep track of this meta data at the end of this loop so we know the predecessors. We then take this resulting vector and use it as the vector we multiply the matrix with in the first step. The next resulting vector is all of the vertices reachable from the root node in two steps. We repeat this process until all vertices have been reached.

This simple technique seemed to be easy to parallelize because we could just send chunks of the matrix to each processor. However, once we discovered that not all of the edges connected to each vertex would be in memory for each processor, we decided to look into different methods.

3.3.2 Brandes' Algorithm

We came across a paper by Brandes [3], which described an algorithm for calculating betweenness centrality efficiently. Instead of running in $\Theta(V^3)$ time, like the Floyd-Warshall algorithm, this algorithm costs $\Theta(VE)$. We implemented this algorithm and tried to find a good way to parallelize it. Ultimately, we found a published parallel algorithm that was based off of Brandes' algorithm. We implemented the parallel version (presented as *Algorithm 1* in a paper by Bader and Madduri [1]). Fortunately, this algorithm requires very few modifications from the original. In addition to some very minor tweaks, the only additions were a number of directives, such as `#pragma omp parallel for`, and the associated declaration of critical sections and shared memory management.

All of our experimentation and results are from this implementation.

4 Experimentation

We began testing our implementation of the parallel Brandes algorithm using synthetic graphs. We randomly generated this data by doubly-looping through every vertex and adding an edge with a probability of 25%. This obviously did not create accurate "social" graphs due to the fact that there would be no clusters. However, we were able to try and visually validate our implementation using the Graphviz renderings:

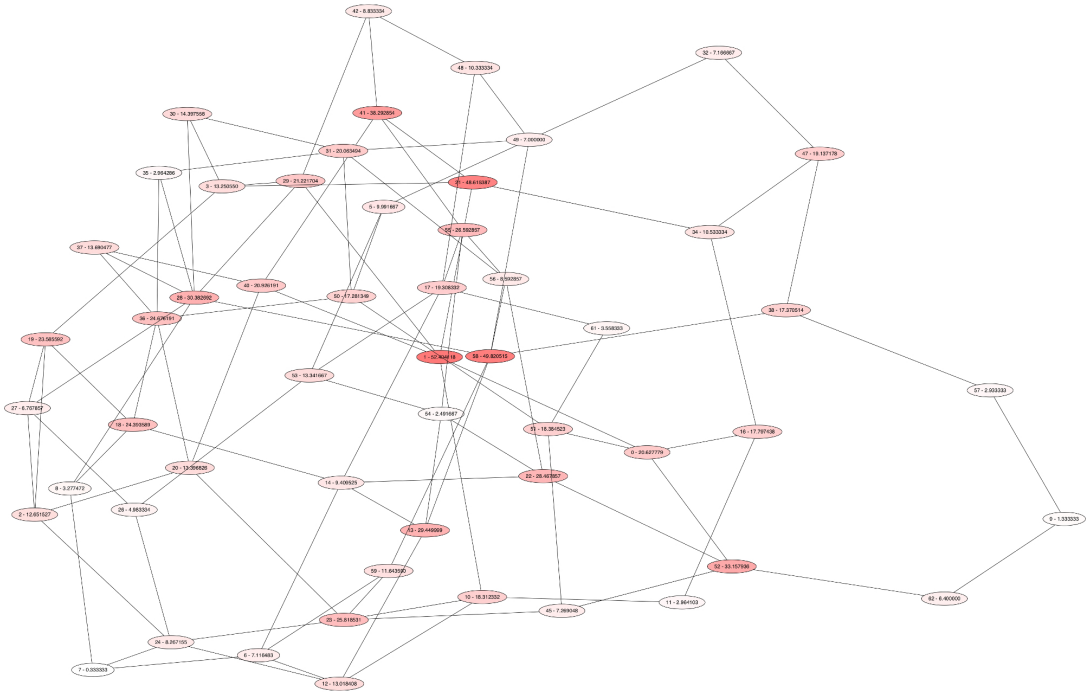

Figure 1: Synthetic with 64 edges

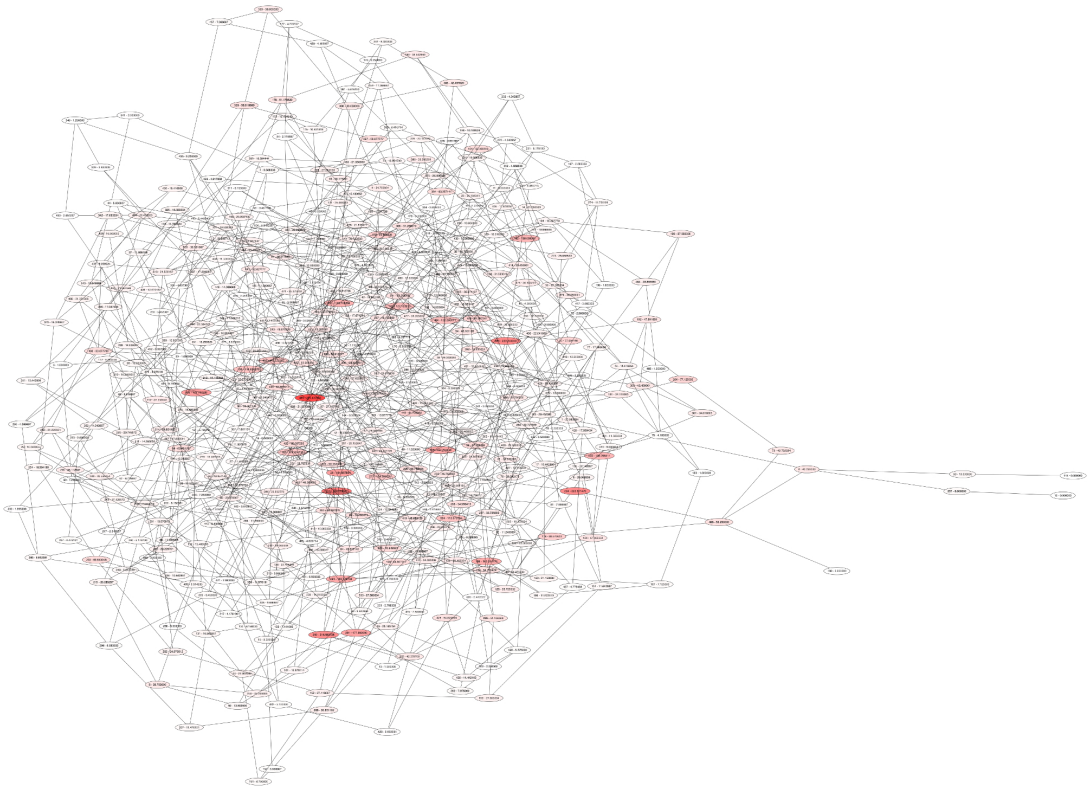

Figure 2: Synthetic with 512 edges

The vertices with deeper reds have larger betweenness centrality metric values.

Once we were comfortable with our implementation, we found datasets used in a Stanford paper by Leskovec and Mcauley [7]. We downloaded this dataset and read it into memory. We ran this through our system and generated the following Graphviz renderings:

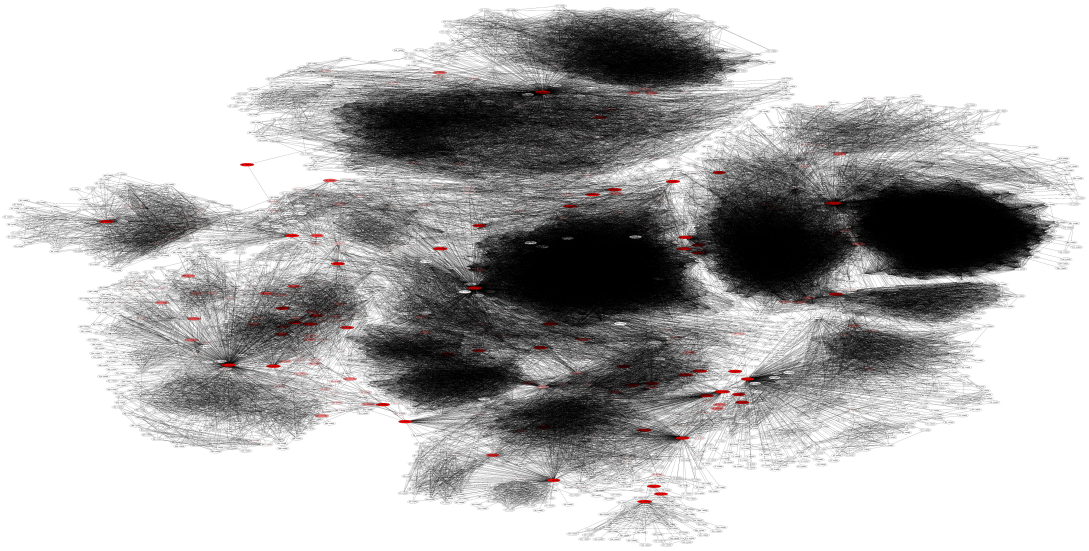

Figure 3: Facebook betweenness centrality calculation

5 Results

While the graphs we were able to generate were quite exciting, unfortunately our implementation appears to have poor performance when scaling. This is most likely due to memory contention from having the whole graph in shared memory.

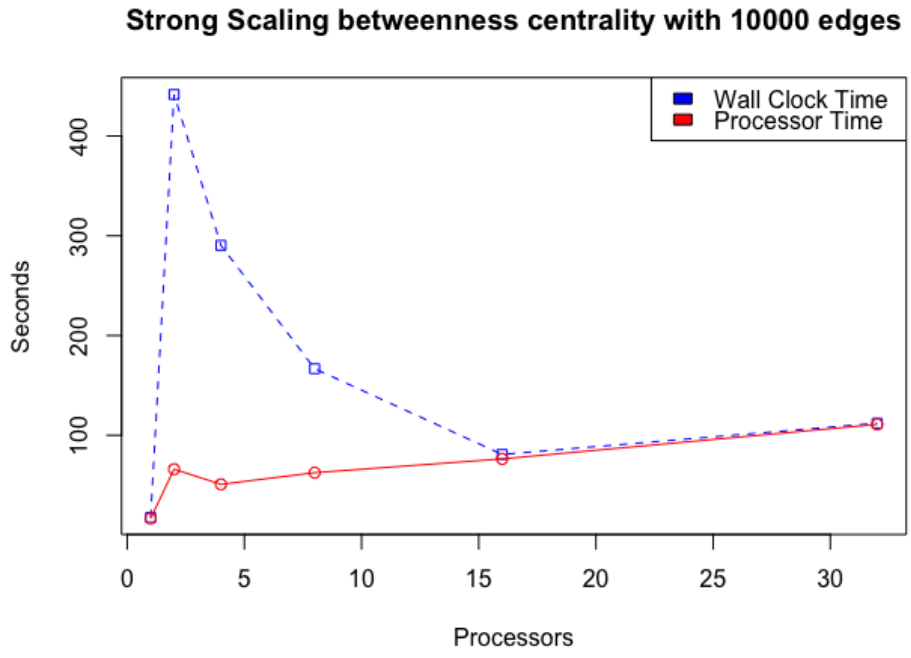

Figure 4: Strong scaling for betweenness centrality on a synthetic graph with 10000 edges

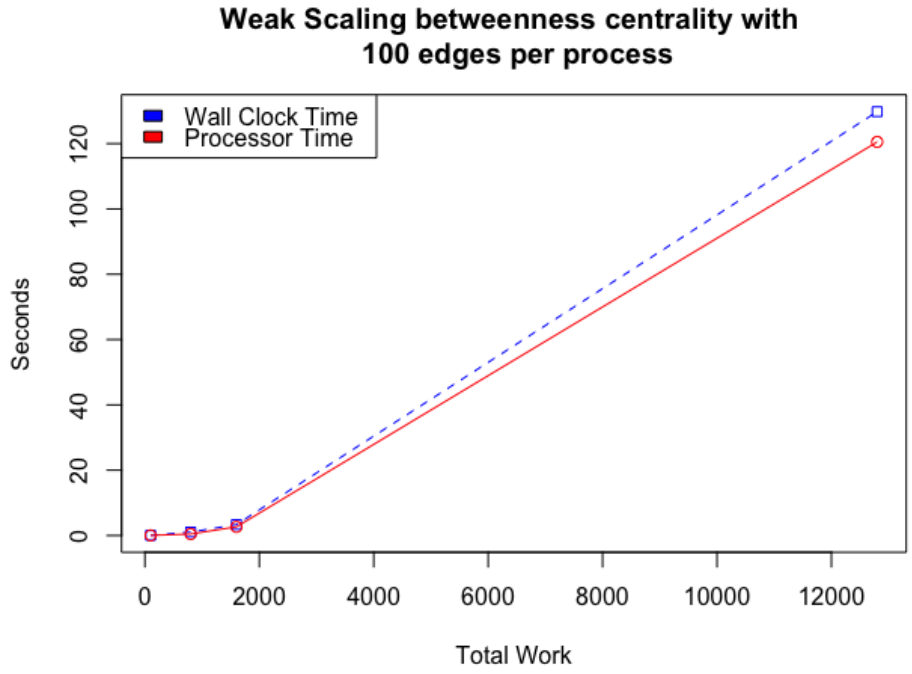

Figure 5: Weak scaling for betweenness centrality on a synthetic graph with 100 edges per process

This figure indicates that our implementation has very poor weak scaling. We hypothesize that this stems from the use of the graph being used in shared memory.

6 Conclusion

Our implementation of the Brandes' algorithm could be improved by distributing the memory across the processors instead of using shared memory. With that said, we achieved some interesting results in finding the vertices in a graph that have high betweenness centrality.

7 Future Work

Looking forward, there are a number of improvements we could make. Our scalability results showed poor performance, which is likely due to all of the contention over the shared memory. If we had more time, we would like to explore different options for distributing the graph across processors.

In addition we would like to try and find existing datasets that have already performed betweenness centrality calculations so that we might validate our metrics. Also it would be helpful to try with different graph density and see how our algorithms scales in different situations.

References

[1] Bader, D. A., and Madduri, K. Parallel algorithms for evaluating centrality indices in real-world networks. In *Parallel Processing, 2006. ICPP 2006. International Conference on* (2006), IEEE, pp. 539–550.

[2] Beamer, S., Asanović, K., and Patterson, D. Direction-optimizing breadth-first search. *Scientific Programming 21*, 3-4 (2013), 137–148.

[3] Brandes, U. A faster algorithm for betweenness centrality. *Journal of mathematical sociology 25*, 2 (2001), 163–177.

[4] Burt, R. S. *Structural holes: The social structure of competition*. Harvard university press, 2009.

[5] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[6] Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM) 24*, 1 (1977), 1–13.

[7] Leskovec, J., and Mcauley, J. J. Learning to discover social circles in ego networks. In *Advances in neural information processing systems* (2012), pp. 539–547.