

# Approximated Parallel Betweenness Centrality

Geoffrey Foster  
School of Computer Science  
Carleton University  
Ottawa, Canada K1S 5B6  
*gfoster@scs.carleton.ca*

December 17, 2007

## Abstract

Using a CGM algorithm the betweenness centrality of a very large scale network with almost 4 million vertices and almost 16 million edges was approximated to within a  $\pm 25\%$  rate of error using only 4 processors in under an hour of computational time.

## 1 Introduction

The concept of Betweenness Centrality was first introduced by Freeman[8] as a measure of the importance of a node (actor) in a network based upon how much traffic flows through them.

For a given graph  $G = (V, E)$  that has  $n$  vertices, the betweenness centrality ( $C_B(v)$ ) is measured for a vertex,  $v$  by the following:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

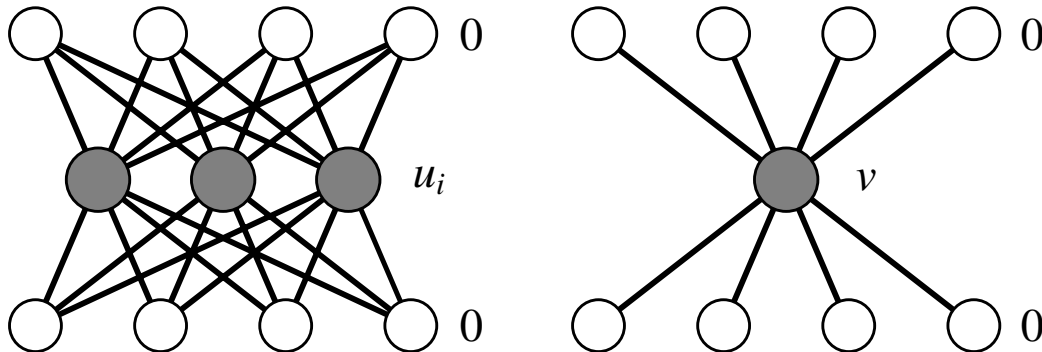
where  $\sigma_{st}$  is the number of shortest paths from  $s$  to  $t$  and  $\sigma_{st}(v)$  is the number of shortest paths from  $s$  to  $t$  that pass through the vertex  $v$ . Two simple examples of betweenness centrality are shown in Figure 1.

Typical sequential betweenness centrality algorithms use a breadth-first-search which has a runtime of  $\mathcal{O}(n+m)$ , where  $n = |V|$ ,  $m = |E|$ , and thus an overall runtime of  $\mathcal{O}(n(n+m))$  since a breadth-first-search must be done for each vertex. For large scale networks this can be quite costly. In order to reduce the amount of time for these computations approximations can be performed.

The aim of this project is to develop and test a parallel implementation for computing an approximated betweenness centrality on a real-world, large-scale social network data set.

In Section 2 a review of the relevant literature is presented along with some of the required mathematics and algorithms that are used. In Section 3 the algorithm used (Section 3.1), its performance (Section 3.2), its results (Section 3.3) and limitations (Section 3.4) for the parallel implementation are shown. Finally, in Section 4 the conclusions are presented.

Figure 1: Examples of Betweenness Centrality Measurements:  $C_B[u_i] = \frac{1}{3}$  and  $C_B[v] = 1$



## 2 Literature Review

In [4] Brandes provided a sequential algorithm (see Algorithm 1) for computing the betweenness centrality of a graph. His algorithm has a running time of  $\mathcal{O}(n(n+m))$  and uses  $\mathcal{O}(n+m)$  space for a graph,  $G(V, E)$  where  $n = |V|$  and  $m = |E|$ . His algorithm is very similar to the standard Breadth-First-Search except with a few modifications. The first is that a stack is maintained that contains the vertices in the order in which they are visited. This allows us to work backwards from all of the leaf vertices to the source vertex in order of distance from the source. The second is that while doing the BFS each vertex maintains a list of possible back-edges (edges that would lead back to the source) that are the minimal distance away from the source. Third, a  $\sigma$  value is maintained for each vertex that tracks the number of shortest paths that go through a vertex. Finally, after the completion of any BFS computation each element in the stack is popped and a  $\delta$  value is computed, and, so long as the vertex currently being examined is not the source vertex, then that  $\delta$  value is added to that vertex's centrality.

A randomized approximation approach was presented in [7] by Eppstein and Wang. It is a sequential algorithm for computing the closeness centrality of the vertices in a graph. The closeness centrality metric is similar to betweenness centrality in that it requires the computation of single source shortest path. The presented algorithm performs a set number of iterations and for each generation uniformly at random selects a vertex from the graph and solves the single source shortest path problem for it. After completing the set number of iterations an estimate is obtained where each estimated value should probabilistically fall within an error range.

Building upon this in [9], Jacob, Peeters et al. used a similar technique in order to approximate the betweenness centrality for each vertex of a graph. Instead of performing  $n$  iterations of single source shortest path computations (one for each vertex) like Brandes does they chose  $K$  sample vertices and perform the SSSP computations for just those  $K$  vertices. This reduces the running time cost of their algorithm to  $\mathcal{O}(K \cdot (n+m))$ . The  $K$  value they chose to use is  $\Theta(\log n / \epsilon^2)$ , where  $\epsilon$  is a very small constant. This choice for  $K$  means

that  $|\hat{c}_B(v) - c_B(v)| \leq \epsilon n(n-1)$  with probability  $1/n$  where  $\hat{c}_B(v)$  is the approximated betweenness centrality for vertex  $v$  and  $c_B(v)$  is the actual betweenness centrality for vertex  $v$ .

In [12], Yang and Lonardi present an MPI implementation of a clustering algorithm that utilizes betweenness centrality for detecting clusters in protein-protein interaction networks. The betweenness clustering algorithm requires multiple iterations, each of which computes the betweenness centrality for the entire graph. The running time for this is stated as  $\mathcal{O}(nm^2)$ . Once again, the algorithm presented by Brandes[4] is used because of its  $\mathcal{O}(nm)$  runtime. Each of the centrality computations (breadth first search, summing of pair dependencies) for each vertex is done in parallel. In order to accomplish this each of the processors receives their own copy of the graph. During each iteration of the clustering algorithm, after the betweenness centrality is computed, each processor removes the same set of edges in order to stay in synch.

---

**Algorithm 1:** Betweenness centrality in unweighted graphs

---

```

 $C_B[v] \leftarrow 0, v \in V;$ 
for  $s \in V$  do
     $S \leftarrow$  empty stack;
     $P[w] \leftarrow$  empty list,  $\forall w \in V;$ 
     $\sigma[t] \leftarrow 0, t \in V; \sigma[s] \leftarrow 1;$ 
     $d[t] \leftarrow -1, t \in V; d[s] \leftarrow 0;$ 
     $Q \leftarrow$  empty queue;
    enqueue  $s \rightarrow Q;$ 
    while  $Q$  not empty do
        dequeue  $v \leftarrow Q;$ 
        push  $v \rightarrow S;$ 
        foreach neighbour  $w$  of  $v$  do
            //  $w$  found for the first time?
            if  $d[w] < 0$  then
                enqueue  $w \rightarrow Q;$ 
                 $d[w] \leftarrow d[v] + 1;$ 
            // shortest path to  $w$  via  $v$ ?
            if  $d[w] = d[v] + 1$  then
                 $\sigma[w] \leftarrow \sigma[w] + \sigma[v];$ 
                append  $v \rightarrow P[w];$ 
         $\delta[v] \leftarrow 0, v \in V;$ 
        while  $S$  not empty do
            pop  $w \leftarrow S;$ 
            for  $v \in P[w]$  do
                 $\delta[v] \leftarrow \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w]);$ 
            if  $w \neq s$  then
                 $C_B[w] \leftarrow C_B[w] + \delta[w];$ 

```

---

### 3 Approximated Parallel Betweenness Centrality

#### 3.1 Algorithm

In order to parallelize the computation of the approximation of betweenness centralities we use the Coarse-Grained Multi-computer model[6] and use a similar approach as mentioned in [12]. The major difference to our algorithm is that we are going for an approximation and thus we will not need to do all  $n$  SSSP computations. Instead, we borrow the approach for sequential approximated betweenness centrality[9] and implement it to work in parallel. The following is a high-level description of the process:

- Each processor loads the graph into local main memory
- Given the sample size,  $K$ , have  $p_0$  do the following:
  - Adjust  $K$  to distribute evenly among the processors and to fully utilize resources (within limits)
  - Take a random unique subset of size  $K$  from the vertices of the graph  $G$
  - Send  $K/p$  sample vertices to each processor
- Each processor receives *local\_sample\_subset*, a unique set of  $K/p$  sample vertices, from  $p_0$
- Each processor runs one iteration of the modified Brandes betweenness centrality algorithm (see Algorithm 2) for each  $v \in local\_sample\_subset$
- Each processor distributes a subset of the centrality values it has computed to the other processors
  - Each processor sends to  $p_i$  the subset indexed by  $[i \cdot \frac{n}{p}, (i + 1) \cdot \frac{n}{p})$
- Each processor receives  $p \cdot \frac{n}{p}$  centrality values, *incoming\_centralities*, where each  $\frac{n}{p}$  is for the same set of  $\frac{n}{p}$  vertices
  - Set the centrality of some pre-determined vertex (at index  $i$ ), to be

$$\frac{n}{K} \cdot \sum_{j=0}^{p-1} incoming\_centralities[i + j \cdot p]$$

In the presented algorithm we can see that there are only two h-relations. The first occurs when  $p_0$  divides the  $K$  sample vertices among all processes and the second occurs when each processor sends a subset of its results to all other processors. Also, note that the modified betweenness centrality (Algorithm 2) is merely a single iteration of the normal sequential algorithm as shown in Algorithm 1.

#### 3.2 Analysis

From [4] we know that we can compute the betweenness centralities for all vertices of an unweighted, undirected graph,  $G$ , in  $\mathcal{O}(n(n + m))$  time. Instead of computing the betweenness using all  $n$  vertices we can compute an approximation using only  $K = \Theta(\log n / \epsilon^2)$  sample vertices as seen in [9]. This leaves us with an overall running time of  $\mathcal{O}(\frac{\log n}{\epsilon^2}(n + m))$

---

**Algorithm 2:** Modified Betweenness Centrality - Runs the betweenness algorithm for only the given source vertex

---

**Input:**  $s$ : The source vertex  
 $S \leftarrow$  empty stack;  
 $P[w] \leftarrow$  empty list,  $\forall w \in V$ ;  
 $\sigma[t] \leftarrow 0, t \in V; \sigma[s] \leftarrow 1$ ;  
 $d[t] \leftarrow -1, t \in V; d[s] \leftarrow 0$ ;  
 $Q \leftarrow$  empty queue;  
enqueue  $s \rightarrow Q$ ;  
**while**  $Q$  not empty **do**  
    dequeue  $v \leftarrow Q$ ;  
    push  $v \rightarrow S$ ;  
    **foreach** neighbour  $w$  of  $v$  **do**  
        //  $w$  found for the first time?  
        **if**  $d[w] < 0$  **then**  
            enqueue  $w \rightarrow Q$ ;  
             $d[w] \leftarrow d[v] + 1$ ;  
        // shortest path to  $w$  via  $v$ ?  
        **if**  $d[w] = d[v] + 1$  **then**  
             $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$ ;  
            append  $v \rightarrow P[w]$ ;  
 $\delta[v] \leftarrow 0, v \in V$ ;  
**while**  $S$  not empty **do**  
    pop  $w \leftarrow S$ ;  
    **for**  $v \in P[w]$  **do**  
         $\delta[v] \leftarrow \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$ ;  
    **if**  $w \neq s$  **then**  
         $C_B[w] \leftarrow C_B[w] + \delta[w]$ ;

---

for the sequential algorithm. Since each single source shortest path computation performed is independent from the next the  $\Theta(\frac{\log n}{\epsilon^2})$  sample vertices can be computed independently from each other, and thus in parallel. As a result our total running time ends up being  $\mathcal{O}(\frac{K}{p}(n+m))$  where  $K = \Theta(\frac{\log n}{\epsilon^2})$

The memory usage requirements for Brandes' original algorithm are  $\mathcal{O}(n+m)$  and in fact, whether doing just one iteration or all  $n$ , the worst case memory usage requirements remain the same. Thus, each processor will use at least the same amount of memory as the sequential version. Examining our algorithm we can see that we add an additional two data structures. The first is a list of sample vertices that the processor is to use as the source vertex. Since we will never exceed the number of vertices we know that this list adds at most an additional  $\mathcal{O}(n)$ . Furthermore, when we receive the subset of vertices from all other processors in the second h-relation, we will receive at most  $p \cdot \lceil \frac{n}{p} \rceil$  values, which is also  $\mathcal{O}(n)$ . Therefore we still maintain a  $\mathcal{O}(n+m)$  memory usage.

### 3.3 Results

Unfortunately few results were able to be obtained. Using a graph with 108388 vertices and 203785 edges with a  $K \approx 100$  and the number of processors varying from 1 to 50 resulted in a performance and speedup as shown in Figure 2 and Figure 3. At roughly the 15 processors mark the performance and speedup becomes very disappointing. This is likely because at the 15 processor mark the communication cost starts to overtake the benefit of the additional processors. If we look only at the plot where  $p \leq 15$  in Figures 4 and 5 the performance and speedup begin to look better.

Using the full data-set of 3986232 vertices and 15726860 edges and  $K = 100$  (an approximate 25% error rate) on four processors showed promising results for running with a large set of data in a parallel environment. The total time required for this was 3190.81 seconds. From this it was determined that an approximate 115 seconds (or about 2 minutes) was spent by a single processor doing a single modified breadth first search (as seen in Algorithm 2).

### 3.4 Limitations

One of the major constraints of this approach is physical memory limitations. Upon testing our implementation it was found that, on average, loading a graph with 3986232 vertices that contained 15726860 edges would consume approximately 1.2 Gigabytes of memory on a 32-bit machine. The time, on average, associated with loading this graph was 145 seconds. If we were to use an even larger and/or denser graph both the memory usage and loading time would degrade even further.

## 4 Conclusion

Theoretically approximated betweenness centrality is well suited to being computed in a parallel environment. Unfortunately problems encountered while trying to test various graph and processor count combinations have resulted in a lack of abundant quality results. What we were able to see on the smaller subset of the graph is that the algorithm does scale up to a point. Being able to try again at using the cluster to test the full data-set, and having it give results, would be interesting to see since on a very large graph we were able

Figure 2: Runtime Performance of a subgraph from data-set with  $|V| = 108388, |E| = 203785, K \cong 100$

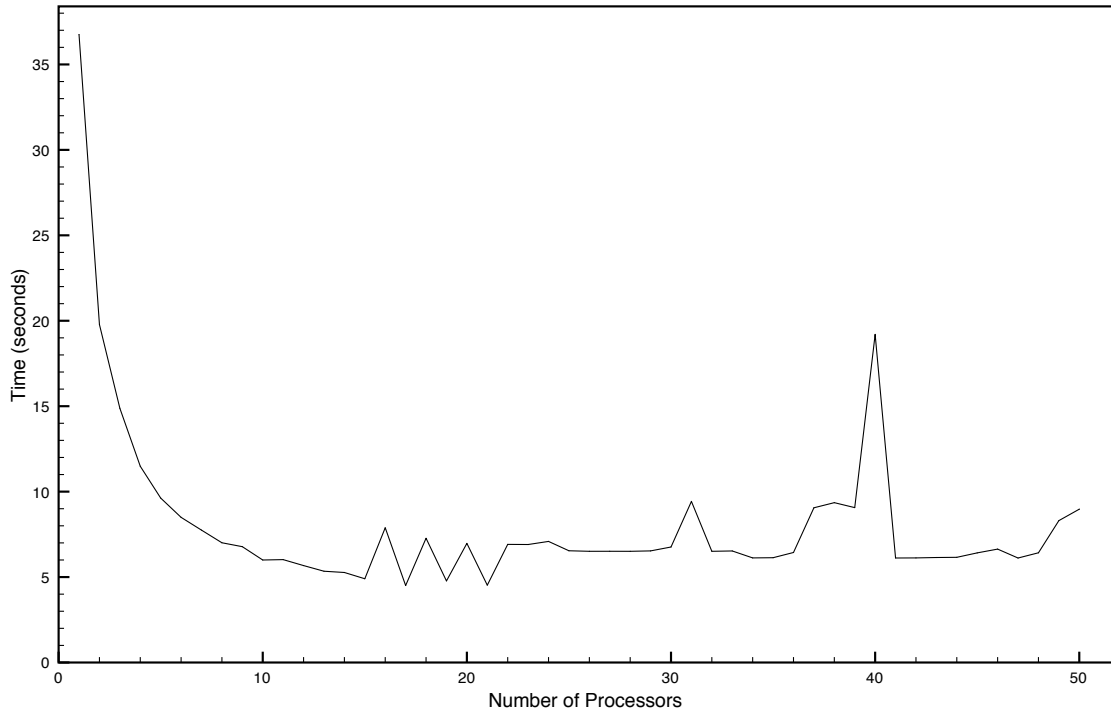


Figure 3: Speedup of data shown in Figure 2

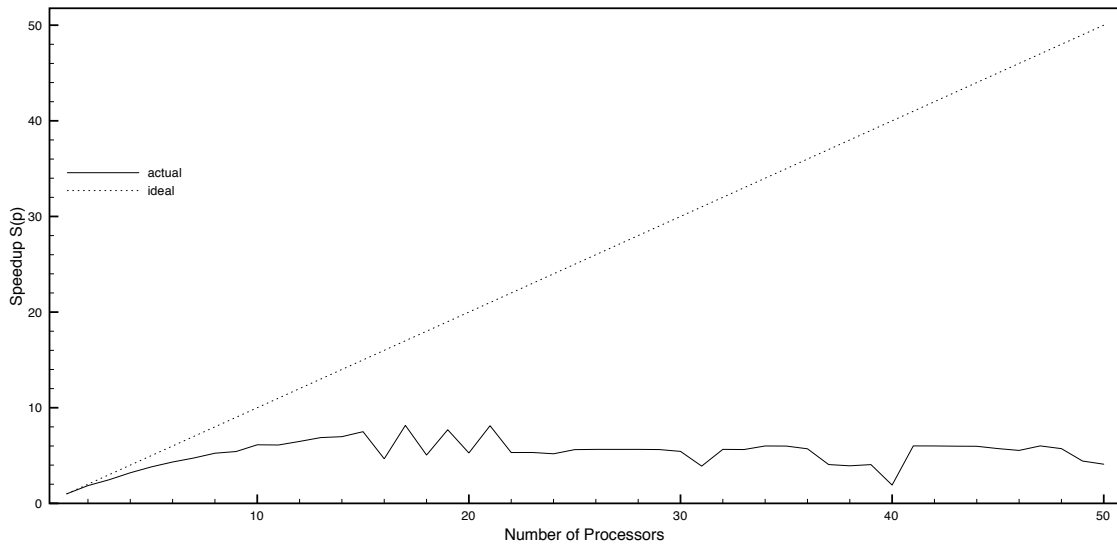


Figure 4: Subset of Figure 2

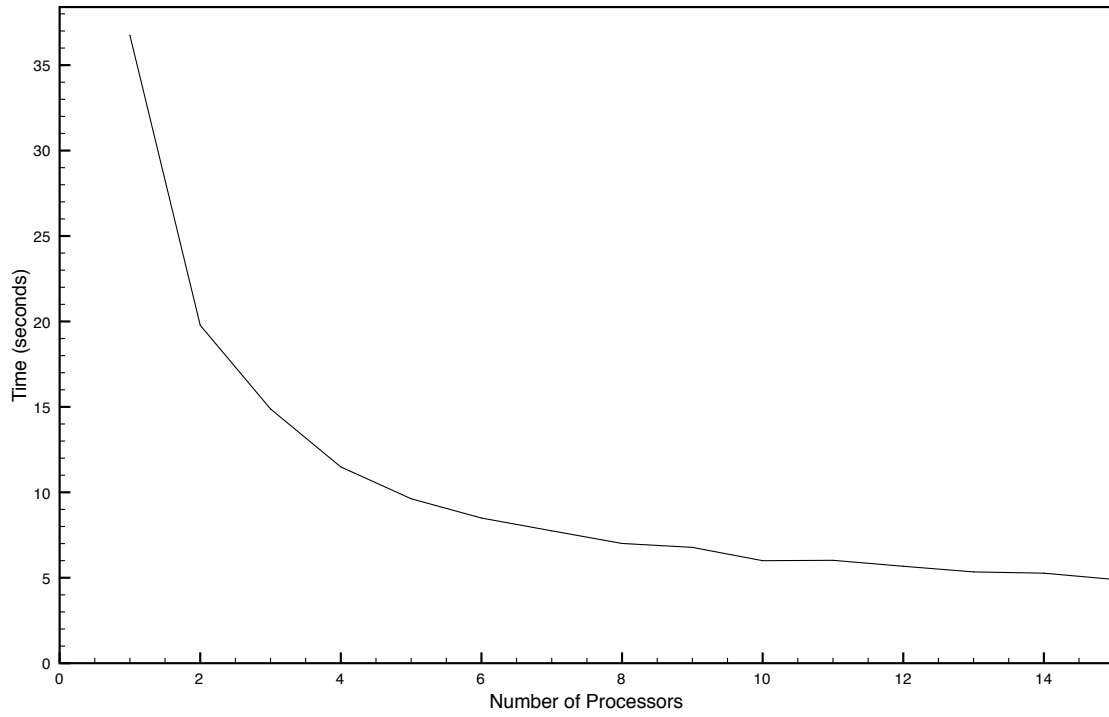
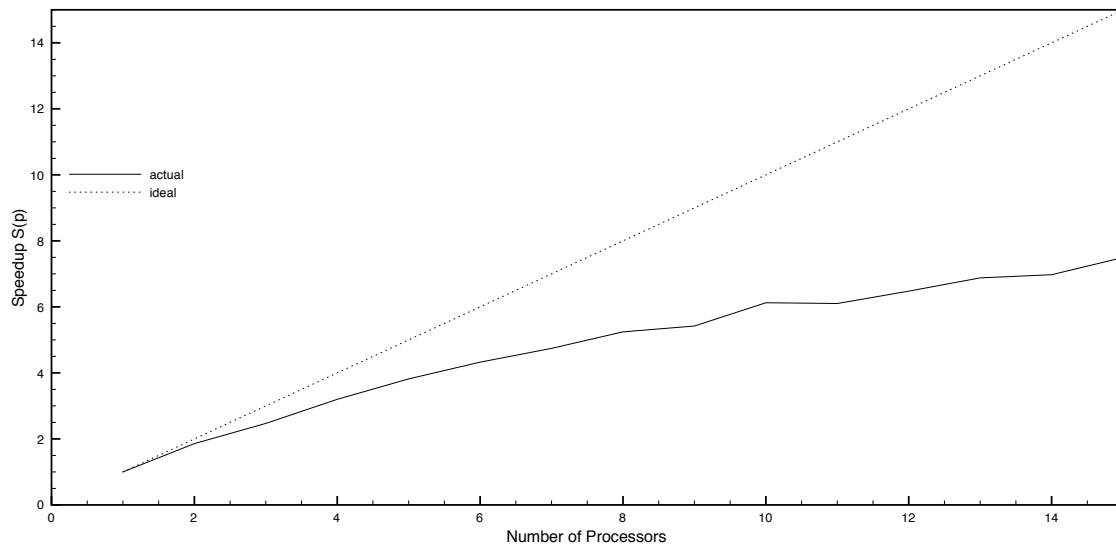


Figure 5: Subset of Figure 3



to compute the centralities, within a  $\pm 25\%$  range of error on only 4 processors in under an hour. Another interesting experiment would be to recreate the algorithm but designed for massive shared memory environments. This would allow us to only need to load one instance of the graph. It would also eliminate the second h-relation in the algorithm thus removing the additional communication needed.

## References

- [1] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality.
- [2] David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP)*. IEEE Comp. Soc. Dig. Library, 2006.
- [3] Ulrik Brandes. Faster evaluation of shortest-path based centrality indices, 2000.
- [4] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
- [5] Ulrik Brandes, Thomas Erlebach, Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, and Oliver Zlotowski. *Network Analysis – Methodological Foundations*, volume 3418 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005.
- [6] Frank K. H. A. Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Symposium on Computational Geometry*, pages 298–307, 1993.
- [7] David Eppstein and Joseph Wang. Fast approximation of centrality. *Journal of Graph Algorithms and Applications*, 8(1):39–45, 2004.
- [8] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.
- [9] Riko Jacob, Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters, and Dagmar Tenfelde-Podehl. Algorithms for Centrality Indices. *Network Analysis*, 3418:62–82, 2005.
- [10] Dirk Koschützki, Katharina Anna Lehmann, Leon Peeters, Stefan Richter, Dagmar Tenfelde-Podehl, and Oliver Zlotowski. Centrality indices. *Network Analysis*, 3418:16–61, 2005.
- [11] Eunice E. Santos, Long Pan, Dustin Arendt, and Morgan Pittkin. An effective anytime anywhere parallel approach for centrality measurements in social network analysis. *IEEE International Conference on Systems, Man, and Cybernetics*, 6:4693–4698, 2006.
- [12] Q. Yang and S. Lonardi. A parallel algorithm for clustering protein-protein interaction networks. *Computational Systems Bioinformatics Conference, 2005. Workshops and Poster Abstracts. IEEE*, pages 174–177, 2005.