A Space-Efficient Parallel Algorithm for Computing Betweenness Centrality in Distributed Memory

Nick Edmonds
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
ngedmond@osl.iu.edu

Torsten Hoefler*
National Center of Supercomputing Applications
University of Illinois at Urbana-Champaign
Urbana, IL 61801
htor@illinois.edu

Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
lums@osl.iu.edu

***Abstract*—Betweenness centrality is a measure based on shortest paths that attempts to quantify the relative importance of nodes in a network. As computation of betweenness centrality becomes increasingly important in areas such as social network analysis, networks of interest are becoming too large to fit in the memory of a single processing unit, making parallel execution a necessity. Parallelization over the vertex set of the standard algorithm, with a final reduction of the centrality for each vertex, is straightforward but requires $\Omega(|V|^2)$ storage. In this paper we present a new parallelizable algorithm with low spatial complexity that is based on the best known sequential algorithm. Our algorithm requires $\mathcal{O}(|V| + |E|)$ storage and enables efficient parallel execution. Our algorithm is especially well suited to distributed memory processing because it can be implemented using coarse-grained parallelism. The presented time bounds for parallel execution of our algorithm on CRCW PRAM and on distributed memory systems both show good asymptotic performance. Experimental results with a distributed memory computer show the practical applicability of our algorithm.**

I. Introduction

Centrality indices are an important measure of the relative importance of nodes in sparse networks [29]. Here we discuss betweenness centrality [2], [16] as it is one of the more commonly used metrics and one of the more difficult to compute efficiently. Betweenness centrality is based on determining the number of shortest paths from s to t (σ_{st}) in a graph $\mathcal{G} = (V, E)$ for all possible vertex pairs $(s, t) \in V \times V$. The betweenness centrality of v is defined as follows:

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

where $\sigma_{st}(v)$ is the number of shortest paths from s to t which pass through v. A straightforward method for computing betweenness centrality is to solve the all-pairs shortest paths (APSP) problem. Fast sequential methods for solving APSP are known [15], [22], [31]. These dynamic-programming methods are straightforward to parallelize but require $\mathcal{O}(|V|^2)$ space to store their result. A space and time efficient algorithm has been proposed by Brandes [9] but it is difficult to parallelize in a coarse grained fashion because it is based on a label-setting single-source shortest path algorithm.

The key idea in Brandes' algoritm is that pairwise dependencies $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ can be aggregated without storing all of them explicitly. By defining the *dependency* of a source vertex $s \in V$ on a vertex $v \in V$ as $\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$ the betweenness centrality of a vertex v can be expressed as $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$. Brandes shows that the dependency values $\delta_s(v)$ satisfy the following recursive relation:

$$\delta_s(v) = \sum_{w : d(s,w) = d(s,v) + c(u,v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w))$$

Where the weight function $c(u, v)$ returns the positive weight of an edge from $u \to v$.

The sequential algorithm described in Algoritm 1 thus computes betweenness by first determining the distance and shortest path counts from s to each vertex (lines 9 to 18). Second, S is used to revisit the vertices by non-increasing distance from s and dependencies are accumulated (lines 20 to 24).

In this work we present a space efficient algorithm for computing the betweenness centrality of the vertices in a sparse graph. We present analyses using the CRCW PRAM [20] and LogGP [1] models which demonstrate that our algorithm is appropriate for both fine and coarse-grained parallelism and discuss tradeoffs that we made during the algorithm design. Finally we present experimental results which demonstrate the strong and weak scalability of our algorithm using the Parallel Boost Graph Library [18].

* Most work performed while the author was at Indiana University

Algorithm 1: Sequential algorithm for computing betweenness centrality in a weighted graph using Brandes' technique for aggregating dependencies

Input: Graph $\mathcal{G}$, a weight function $c(u, v)$ returning the weight of an edge from $u \to v$
Output: $\forall\, v \in V$: $C_B[v]$ the betweenness centrality

```
1  ∀ v ∈ V: C_B(v) = 0;
2  foreach s ∈ V do
3      S ← empty stack;
4      ∀ w ∈ V: P[w] ← empty list;
5      ∀ t ∈ V: σ[t] ← 0; σ[s] ← 1;
6      ∀ t ∈ V: dist[t] ← ∞; dist[s] ← 0;
7      PQ ← empty priority queue which returns
       elements in non-decreasing order by dist;
8      enqueue s → PQ;
9      while PQ not empty do
10         dequeue v ← PQ;
11         push v → S;
12         foreach neighbor w of v do
13             if dist[v] + c(v, w) < dist[w] then
14                 dist[w] = dist[v] + c(v, w);
15                 enqueue w → PQ;
16             if dist[w] = dist[v] + c(v, w) then
17                 σ[w] ← σ[w] + σ[v];
18                 append v → P[w];
19     ∀ w ∈ V: δ[v] ← 0;
20     while S not empty do
21         pop w ← S;
22         foreach v ∈ P[w] do
23             δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
24         if w ≠ s then C_B[w] ← C_B[w] + δ[w];
```

A. Previous Work

Our algorithm makes use of a parallelizable label-correcting single-source shortest path (SSSP) algorithm such as [12], [27]. A variety of performance results for parallel solutions to the single-source shortest paths problem have been presented [14], [24]. Parallel solutions to the all-pairs shortest paths problem have also been presented [21], [23].

A parallel algorithm for betweenness centrality has been presented in [4], however the algorithm uses a label-setting algorithm to solve the SSSP problem and leverages fine-grained parallelism by relaxing edges incident to each vertex in parallel. This approach exposes some parallelism for symmetric multiprocessors but is unsuitable for distributed memory systems due to the high overhead of distributing the available work, and the relatively small amount of work available at any given time. The algorithm presented also leverages coarse grained parallelism by solving the SSSP problem from multiple sources in parallel. This approach requires storing the solution to the SSSP problem for each source and is thus contrary to our goal of a space efficient algorithm. While greater speedup per processing unit may be attained by leveraging fine-grained parallelism on SMPs, our distributed memory implementation is capable of scaling beyond the size of the available memory on a single SMP. Moreover, each node in a distributed memory cluster is not constrained to a single thread of execution. This presents the possibility of leveraging fine-grained parallelism at the node level and coarse-grained parallelism to enable scaling as well as providing additional computational resources.

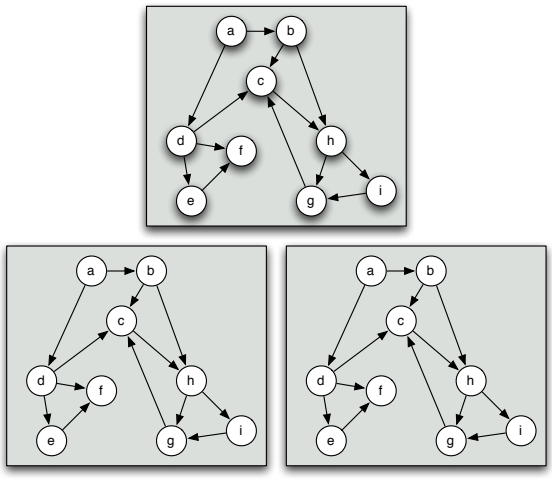


Fig. 1: Single graph replicated across three processes.

Subsequent work observes that successor sets yield better locality than predecessor sets for unweighted betweenness centrality [25], but this work still leverages only fine-grained parallelism and uses breadth-first search to compute unweighted betweenness centrality rather than SSSP to compute weighted betweenness centrality.

The time-optimal coarse-grained method of solving each SSSP in parallel has been presented [32] but as previously stated, is space inefficient. This algorithm requires the graph to be replicated as in Figure 1. We have previously implemented this algorithm but it proved incapable of dealing with large scale graphs which cannot be stored in the memory of a single processing unit. Given the large networks which need to be analyzed such as web graphs or global social networks, a scalable algorithm able to operate on a distributed representation of the network is essential. Our algorithm operates on graphs distributed by vertex with edges stored with their sources, as shown in Figure 2.

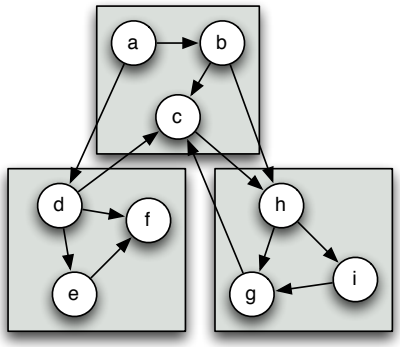


Fig. 2: Single graph distributed across three processes.

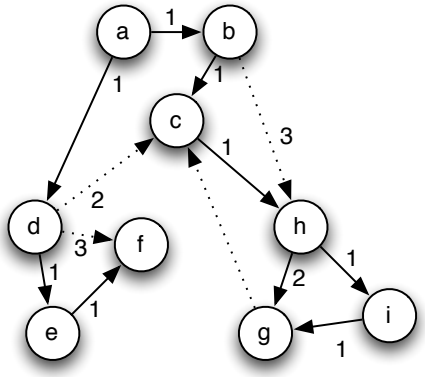


Fig. 3: An example shortest path DAG ($\mathcal{G}'$). Dotted edges represent edges in $\mathcal{G}$ but not in $\mathcal{G}'$.

B. Notation

Here we introduce the notation we will use in the remainder of the paper. The input graph is defined by $\mathcal{G} = (V, E)$, the number of vertices in $\mathcal{G}$ is $|V| = n$ and the number of edges is $|E| = m$, d denotes the average vertex degree. $U(X, Y]$ denotes a uniform distribution on the interval $(X, Y]$. $\sigma_s[v]$ denotes the number of shortest paths which pass through a vertex v for a given source s, while $d(s, t)$ denotes the minimal distance between s and t, i.e., the length of a shortest path from $s \to t$.

We describe our algorithm in detail in the next section. In Section III, we analyze the time and space complexity of the algorithm on random Erdős-Rényi graphs with random edge weights. Erdős-Rényi graphs were chosen for these analyses due to the rich theoretical foundations available on which to build. Performance and simulation results of our algorithm on both Erdős-Rényi and scale-free graphs are presented in Section IV.

II. Revised Betweenness Centrality Algorithm

Our proposed algorithm consists of three phases. The first phase is, similarly to Brandes' betweenness centrality algorithm, the computation of all shortest path counts and the subgraph $\mathcal{G}'_s = (V, E')$ of $\mathcal{G}$ representing all shortest paths in $\mathcal{G}$ from s. Informally, an edge $(u, v) \in E$ is also an edge in $\mathcal{G}'_s$ iff there exists a shortest path from s to v that contains (u, v). In Brandes' original algorithm [9], the subgraph $\mathcal{G}'_s$ was represented by the set of predecessors $\mathcal{P}_s(v) = \{u \in V : (u, v) \in E, d(s, v) = d(s, u) + c(u, v)\}$ of a vertex v. The shortest-path count $\sigma_s[v]$ of a vertex v is updated when a vertex is finished (all shortest paths from $s \to v$ have been determined). The combinatorial shortest path counting (Line 23 in Algorithm 1 and Lemma 3 in [9]) is only correct for label-setting SSSP algorithms, such as Dijkstra's algorithm [13] in weighted graphs or breadth first search (BFS) in unweighted graphs. However, label-setting algorithms often offer very low parallelism due to the requirement that only the vertices with equal minimal distance to the start vertex can be relaxed in parallel. Label-correcting algorithms, such as Bellman-Ford [5] and Δ-stepping [27] offer more parallelism and linear average runtime on random graphs. For a discussion of label-setting versus label-correcting algorithms, refer to [26]. Thus, it is often useful to employ label-correcting algorithms in parallel environments.

In our method, outlined in Algorithm 2, we relax the label-setting requirement of Brandes' algorithm in order to enable the use of different label-correcting algorithms that can be implemented on parallel computers. We do this by storing not only the predecessor set $\mathcal{P}_s(v)$ for each vertex, but also the successor set $\mathcal{S}_s(v) = \{u \in V : (v, u) \in E, d(s, v) = d(s, u) + c(u, v)\}$. This set can be used to traverse $\mathcal{G}'_s$ in non-decreasing distance from s after all shortest paths have been found. This allows us to accumulate the number of shortest paths, σ_s, accordingly. An example shortest path DAG $\mathcal{G}'$ (and thus P_s and S_s) is shown in Figure 3. For example, $P_s(\mathrm{g}) = \{\mathrm{h}, \mathrm{i}\}$ and $S_s(\mathrm{i}) = \{\mathrm{g}\}$ in the depicted graph.

We note that the successor set $\mathcal{S}_s$ can be derived from the predecessor set $\mathcal{P}_s$, and vice versa, by transposing the graph represented by the set (see Step 2 in Algorithm 2). Thus, a shortest path algorithm only needs to return one of the sets; however, under some circumstances, it might be beneficial to generate both sets at the same time (which would effectively merge Steps 1 and 2 in Algorithm 2).

We chose Δ-stepping to compute shortest paths (Step 1 in Algorithm 2) because it is work efficient and readily lends itself to being implemented in distributed memory. Δ-stepping replaces the priority queue in Dijkstra's algorithm with an array B of *buckets* such that $B[i]$ stores $\{v \in V : tent[v] \in [i\Delta, (i+1)\Delta]\}$ where $tent[v]$ is the tentative distance from s to v. Δ-stepping for weighted graphs is outlined in Algorithm 3 and 4 while the algorithm degenerates to breadth first search in the unweighted case.

Algorithm 2: Distributed Betweenness Centrality

Input: Graph $\mathcal{G}$

$\forall\, v \in V$: $C_B(v) = 0$;
foreach $s \in V$ **do**
    // Step 1: Compute shortest path predecessors
    $\mathcal{P}_s = shortestpaths(\mathcal{G}, s)$;
    // Step 2: Compute shortest path successors from predecessors
    $\mathcal{S}_s = transpose(\mathcal{P}_s)$;
    // Step 3: Compute path counts in DAG of shortest paths
    $(\sigma_s, Q) = pathcounts(\mathcal{S}_s, s)$;
    // Step 4: Update betweenness centrality
    $C_B = updatecentrality(\mathcal{P}_s, Q, \sigma_s, C_B)$;

If a label-correcting algorithm provides some information about when the distance to a node is settled then path counts can be computed incrementally during the course of the shortest paths algorithm and $\mathcal{S}_s$ is unnecessary (merging Step 1, 2 and 3 in Algorithm 2).

Algorithm 3: $shortestpaths(G, s)$ – find shortest paths predecessor map – Δ-stepping [27]

Input: Weighted graph $\mathcal{G} = (V, E)$, vertex s
Output: $\forall\, v \in V$: predecessors $\mathcal{P}_s = \{p_i\}$ on all i shortest paths $(s, \ldots, p_i, v)$

1 $\forall\, v \in V : tent[v] = \infty; \mathcal{P}_s[v] = \emptyset$;
2 $i = 0$; $B[0] = s$; $tent[s] = 0$;
3 **while** B *not empty* **do**
4 $D = \emptyset$;
5 **while** $B[i] \neq \emptyset$ **do**
6 $R = \{(v, w) | \forall\, v \in B[i] \wedge c(v, w) \leq \Delta\}$;
7 $D = D \cup B[i]$; $B[i] = \emptyset$;
8 **foreach** $(v, w) \in R$ **do** relax(v, w);
9 $R = \{(v, w) | \forall\, v \in D \wedge c(v, w) > \Delta\}$;
10 **foreach** $(v, w) \in R$ **do** relax(v, w);
11 $i = i + 1$;

For Δ-stepping, this means that once a bucket $B[i]$ is emptied (line 11 in Algorithm 3), all vertices removed from $B[i]$ are settled and it would be possible to determine the path count $\sigma_s[v], v \in B[i]$. However, this requires finding the set $A = \{u \in B[0..i-1] : tent[u] + c(u, v) = tent[v]\}$ and then traversing $B[i]$ in non-increasing order of distance from s starting with the vertices in A. Finding A can be done in two ways. A data structure containing all settled vertices that have paths to non-settled vertices can be maintained and traversed after each bucket is settled. Alternately, a topological ordering of $v \in B[i]$ w.r.t. $\mathcal{G}'$ can be found and traversed backwards. Because each vertex in the settled set might be visited multiple times, it is more work efficient to postpone the determination of σ_s until all vertices have been settled and perform a single traversal of $\mathcal{G}'$ from s.

Algorithm 4: $relax(v, w)$ – relax part of Δ-stepping

Input: Vertices $v, w \in V$
Output: Updated B, $\mathcal{P}_s[w]$, and $tent[w]$

1 $dist = tent[v] + c(v, w)$;
2 **if** $dist < tent[w]$ **then**
3 **if** $tent[w] < \infty$ **then**
4 $B[\lfloor tent[w]/\Delta \rfloor] = B[\lfloor tent[w]/\Delta \rfloor] \backslash \{w\}$;
5 $B[\lfloor dist/\Delta \rfloor] = B[\lfloor dist/\Delta \rfloor] \cup \{w\}$;
6 $tent[w] = dist$;
7 $\mathcal{P}_s[w] = \{v\}$;
8 **else if** $dist = tent[v]$ **then**
9 $\mathcal{P}_s[w] = \mathcal{P}_s[w] \cup \{v\}$;

Algorithm 5: $pathcounts(\mathcal{S}_s, s)$ – accumulate shortest path counts

Input: Successor set $\mathcal{S}_s[v]$,$\forall v \in V$, starting vertex s
Output: Number of shortest paths $\sigma[v]$, $\forall v \in V$, Q, a queue consisting of all vertices with no successors

1 $localQ \leftarrow$ empty queue;
2 $\sigma[t] = 0, \forall t \in V; \sigma[s] = 1$;
3 enqueue $s \rightarrow localQ$;
4 **while** $localQ$ *not empty* **do**
5 dequeue $v \leftarrow localQ$;
6 **foreach** $w \in S[v]$ **do**
7 $\sigma[w] = \sigma[w] + \sigma[v]$;
8 enqueue $w \rightarrow localQ$;
9 **if** $\mathcal{S}_s[v] =$ *empty list* **then**
10 enqueue $v \rightarrow Q$;

We chose to compute the pathcount $\sigma_s[v]$ from s to all $v \in V$ after all shortest paths have been found. This is done in Step 3 of Algorithm 2 which is outlined in Algorithm 5 as a level-synchronized breadth first search. Level-synchronized means that no vertex in level $i+1$ is discovered before every vertex in level i is discovered. This traversal requires the successor set $\mathcal{S}_s[v], v \in V$. When $\mathcal{G}$ is unweighted $\mathcal{S}_s$ can easily be computed at the same time as $\mathcal{P}_s$ in the shortest paths calculation. When $\mathcal{G}$ is weighted however, $\mathcal{P}_s[v]$ is cleared when a shorter path to v is found. Clearing $\mathcal{P}_s[v]$ requires an update to $\mathcal{S}_s$ of the form $\forall w \in \mathcal{P}_s[v] : \mathcal{S}_s[w] = \mathcal{S}_s[w] \setminus v$. This

operation adds $\Theta(|\mathcal{P}_s[v]|)$ work to each edge relaxation, and might require time-consuming communication in distributed memory. For this reason when $\mathcal{G}$ is weighted, we determine $\mathcal{P}_s$ during the shortest paths calculation and calculate $\mathcal{S}_s$ from $\mathcal{P}_s$ after the shortest paths calculation is complete in the optional Step 2 of Algorithm 2.

The fourth and final phase of our algorithm consists of traversing $\mathcal{G}'$ in order of non-increasing distance from s and calculating the dependency (δ) and centrality (C_B), for all vertices similarly to Brandes' algorithm. This is shown in Algorithm 6.

Algorithm 6: $updatecentrality(\mathcal{P}_s, Q, \sigma, \delta)$ – update betweenness centrality

Input: Predecessor set $\mathcal{P}_s[v]$, queue Q, betweenness centrality $C_B[v]$, shortest path counts $\sigma[v]$, $\forall v \in V$,
Output: Updated betweenness centrality $C_B[v]$, $\forall v \in V$

```
// Compute dependency and
   centrality, Q returns vertices
   in non-increasing distance from
   s
```

$\forall t \in V : updates[t] = \delta[t] = 0$;
while Q *not empty* **do**
  dequeue $w \leftarrow Q$;
  foreach $v \in \mathcal{P}_s[w]$ **do**
    $updates[v] = updates[v] + 1$;
    if $updates[v] \geq |\mathcal{S}_s[v]|$ **then**
      $\delta[v] = \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
      enqueue $v \rightarrow Q$;
  if $w \neq s$ **then**
    $C_B[w] = C_B[w] + \delta[w]$;

III. Analysis

In our analysis, we consider random Erdős-Rényi graphs $\mathcal{G}(n, p)$ with the edge-probability $p = {}^{d}/_{n}$. We analyze unweighted graphs as well as weighted graphs where the weight function $c(u, v)$ returns values from a uniform random distribution $(0, 1]$, as well as integers in $(0, C] : C < \infty$. The actual number of edges in $\mathcal{G}$ is $m = \Theta(dn)$ whp[1]. Our analysis assumes the most interesting case, $d = \mathcal{O}(\log n)$ (whp all but the $c \log n$ smallest edges per node can be ignored without changing the shortest path for some constant c [17], [19]).

Our main motivation in designing this algorithm was to be more space efficient than all-pairs shortest paths algorithms such as Floyd-Warshall [15], [31] and to be capable of benefiting from both coarse grained and fine grained parallelism with large graphs. It is straightforward to observe that betweenness centrality can be implemented in terms of n independent SSSP calculations and a reduction operation on the results of those n SSSPs. The time-optimal method to compute betweenness centrality would thus be to perform n SSSPs in parallel. This approach requires $\Omega(n^2)$ space to store any of the several $\mathcal{O}(n)$ data structures such as the path count or dependency of each vertex. Our algorithm requires $\mathcal{O}(m + n)$ space in practice and is therefore more appropriate for the analysis of very large graphs on distributed memory machines.

First, we analyze the structure of the shortest paths subgraph $\mathcal{G}'$; m' denotes the number of edges in $\mathcal{G}'$.

Lemma 1. *The number of equal weight paths in $\mathcal{G}$, with $c(u, v)$ returning values from $U(0, 1]$, is 0 whp.*

Proof: By the Central Limit Theorem, the path length $\mathcal{F}$, a sum of independent samples from the continuous uniform distribution $U(0, 1]$ approximates a normal distribution for large n. If we assume that $U(0, 1] \subset \mathbb{R}^+$, i.e., $U(0, 1]$ is infinitely discretizable, then the probability that a random sample from $\mathcal{F}$ is equal to a specific $b \in U(0, 1]$ is ≈ 0.
In general, given a path of weight δ, the probability that there exists another path of weight δ approaches 0 as the number of possible edge weights approaches ∞. Thus there are no equal weight paths in $\mathcal{G}$, with $c(u, v)$ returning values from $U(0, 1]$, whp. ■

Lemma 2. *When $\mathcal{G}'$ has edges weighted from $U(0, 1]$, then $m' \leq n - 1$ whp.*

Proof: In a connected graph any edge set which connects all the vertices contains at least $n-1$ edges. We conclude with Lemma 1 that there are no equal length paths whp, so that the number of edges m' can be at most $n - 1$ if the graph is connected and must be less than $n - 1$ if the graph is not connected. ■

Lemma 3. *The number of edges m' in $\mathcal{G}'$, when $\mathcal{G}'$ has integer weighted edges in $(0, C] : C < \infty$, is $\mathcal{O}(dn)$.*

Proof: The number of edges in $\mathcal{G}$ is $\Theta(dn)$ whp, $E' \subseteq E$ thus m' must be $\mathcal{O}(dn)$. ■

The term m' appears in a number of subsequent bounds. Determining a tight bound on m' when $c(u, v)$ returns integers in $(0, C], C < \infty$ remains an open problem [6]. We have shown that when $c(u, v)$ returns values from $U(0, 1]$, $m' \leq n - 1$ whp. We conclude that in the most interesting cases of finitely discretizable edge weights and $d \in \mathcal{O}(\log n)$, m' is bounded between $\mathcal{O}(n)$ (Lemma 2) and $\mathcal{O}(n \log n)$ (Lemma 3).

[1]We use the term "whp" throughout the document to say "with high probability", i.e., the probability for some event is at least $1 - n^{\epsilon}$ for a constant $\epsilon > 0$.

A. Space complexity

Our algorithm contains a number of data structures which store data for each vertex including C_B, σ, δ, the tentative distance ($tent$) for the SSSP calculations, and $updates$, a counter we utilize to traverse the graph in dependency order from sink(s) to source. Additionally we maintain $\mathcal{P}_s$ and $\mathcal{S}_s$ which record adjacencies in $\mathcal{G}'$. We maintain several queues including the queue for the SSSP calculation (which in our implementation is actually the array of buckets for Δ-stepping) as well as the queues used to traverse the graph in path count and dependency/centrality computations.

Data Distribution: Vertices in $\mathcal{G}$ can be randomly assigned to processing units (PUs) by generating an array of random PU indices. This can be performed in $\mathcal{O}(n/P)$ time. Edges can be stored on the PU which owns the source of the edge. Storing the graph in this fashion requires $\Theta(m+n)$ space.

Values Associated With Vertices: $C_B, \sigma, \delta, tent$, and dep are all arrays of size $\Theta(n)$.

SSSP Queue: Each of the P PUs maintains its own queues and stores there the queued vertices it is responsible for. At most m edges can be queued, therefore, each queue is $\mathcal{O}(m/P)$ whp.

Predecessor and Successor Adjacencies: Each node will have m'/n expected entries in the predecessor map. For random graphs, predecessors are evenly distributed and each node has an expected number of predecessors m'/n. By Chernoff bounds, a buffer of size $\mathcal{O}(m'/n + \log m')$ per node suffices whp. Data can be placed in the buffer using randomized dart throwing [28]. Periodically checking to see if dart throwing has terminated and increasing the buffer size if necessary preserves correctness in the unlikely case that a buffer is too small. This additional space is only necessary for arbitrary-write CRCW PRAMs to handle concurrent updates to the predecessor set of a single vertex. In models that do not have to handle concurrent writes (including our implementation), $\mathcal{O}(m'/n)$ space suffices by using a dynamic table [11]. Thus the space required to store all predecessor and successor adjacencies is $\mathcal{O}(m' + n \log m')$.

Shortest Paths Queue: Traversing $\mathcal{G}'$ can be performed in $\mathcal{O}(\log n)$ phases whp. Each phase contains $\mathcal{O}(n/\log n)$ nodes therefore $\mathcal{O}(n/P \log n)$ space per PU suffices whp.

Dependency/Centrality Dequeue: Computing dependency/centrality is equivalent to a breadth first search from sink to source and thus the space from the shortest paths queue above can be reused.

Theorem 1. *Our modified betweenness centrality algorithm on random graphs from $\mathcal{G}(n, \frac{d}{n})$ with $d \in \mathcal{O}(\log n)$ requires $\mathcal{O}(m + n \log m')$ space on an arbitrary-write CRCW PRAM. In a machine which does not allow concurrent writes $\mathcal{O}(m+n)$ space suffices.*

Proof: The space required by the various $\Theta(n)$ data structures and the queues used in the shortest paths and dependency/centrality computation is subsumed by the space required by $\mathcal{P}_s$ and $\mathcal{S}_s$, which are $\mathcal{O}(m'+n \log m')$ and the size of the SSSP queue which is $\mathcal{O}(m)$. Because $m' \leq m$, $\mathcal{O}(m' + n \log m' + m) = \mathcal{O}(m + n \log m')$. In a machine where concurrent writes to the same location are not allowed a reduction must be done requiring $\mathcal{O}(\log P)$ time, reducing the space required to $\mathcal{O}(m+n)$. ■

B. PRAM Analysis

We now explain how Algorithm 2 can be efficiently implemented on an arbitrary-write CRCW PRAM.

Shortest Paths Calculation: A variety of shortest paths algorithms could be used in the portion of the algorithm which computes predecessor and successor maps in $\mathcal{G}$ [12], [27]. Δ-stepping has reasonable expected runtime and is straightforward to implement in distributed memory. Δ-stepping can solve the single source shortest path problem on graphs of the aforementioned class in $\mathcal{O}(\log^3 n / \log\log n)$ time using $\frac{dn \log\log n}{\log^3 n}$ processing units (PUs) on a CRCW PRAM [27]. When $\mathcal{G}$ is unweighted the shortest paths calculation degenerates to a breadth first search and can be solved in $\mathcal{O}(\log n)$ time.

Placing predecessors in $\mathcal{P}_s$ can be performed using randomized dart throwing without adding more than $\mathcal{O}(1)$ time per edge relaxation. The output of the shortest paths computation is $\mathcal{P}_s$ which represents the edges in $\mathcal{G}'$.

Calculating Successors given Predecessors: The predecessor lists allow $\mathcal{G}'$ to be traversed from sink to source. If $\mathcal{G}$ is unweighted, successor lists can be computed at the same time as the predecessor lists with no cost because there are no deletions in $\mathcal{P}_s$. In the case of weighted graphs it is straightforward to calculate successors given predecessors in $\mathcal{O}(\log^2 n)$ time. Each vertex is assigned to a PU, the expected number of predecessors per vertex is m'/n which is $\mathcal{O}(\log n)$ when edges have integer weights uniformly distributed in $(0, C] : C < \infty$ ($\mathcal{O}(1)$ if edge weights are infinitely discretizable). Each PU writes the corresponding successor entry in $\mathcal{S}_s$ for each of its predecessors. Reducing the writes by each of the n PUs to $\mathcal{S}_s$ requires $\mathcal{O}(\log n)$ time.

Computing Path Counts: Computing path counts in $\mathcal{G}'$ can be done in $\mathcal{O}(\log^2 n)$ time using n PUs. Each vertex is assigned to a PU, in each iteration every PU checks to see if the path count at its vertex is non-zero. If so it increments the path count at each of its successors in $\mathcal{G}'$

	Unweighted	Weighted
SSSP	$\mathcal{O}(\log n)$	$\mathcal{O}(\frac{\log^3 n}{\log\log n})$ (expected)
Compute $\mathcal{S}$	0 (in SSSP)	$\mathcal{O}(\log^2 n)$
Compute σ_s	0 (in SSSP)	$\mathcal{O}(\log^2 n)$
Compute δ and C_B	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(\log^2 n)$
Total ($\forall s \in V$)	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(\frac{n \log^3 n}{\log\log n})$

Fig. 4: Expected case runtime for each phase of the presented betweenness centrality algorithm for each of the possible classes of edge weights

by its own path count and is finished. The diameter of the giant component $\mathcal{G}'$ is $\mathcal{O}(\log n)$ whp [7], thus calculating path counts requires $\mathcal{O}(\log n)$ iterations each of which requires $\mathcal{O}(\log n)$ time to reduce the writes by each of the n PUs.

Updating Dependency and Centrality: Rather than traversing $\mathcal{G}'$ from source to sink as when computing path counts, computing dependency and centrality requires traversing $\mathcal{G}'$ from sink to source. The operation is fundamentally the same however, updating the dependency of the predecessors of a vertex, rather than the path count of its successors. Care must be taken that all successors of a vertex v have updated the dependency of v before v in turn updates the dependencies of its predecessors. This is easily accounted for by counting the updates to v.

Theorem 2. *Betweenness centrality on random graphs from $\mathcal{G}(n, \frac{d}{n})$ with $d \in \mathcal{O}(\log n)$ can be computed in expected time $\mathcal{O}(\frac{n \log^3 n}{\log\log n})$ using $\frac{dn \log\log n}{\log^3 n}$ PUs on a CRCW PRAM. If $\mathcal{G}$ is unweighted this can be reduced to $\mathcal{O}(n \log n)$ expected time using n PUs.*

C. Distributed Memory Analysis

Our analysis utilizes the well-known LogGP model [1] as framework for formal analysis. The LogGP model incorporates three of the four main network parameters: L is the maximum latency between any two processes, o is the CPU injection overhead of a single message, g is the "gap" between two messages, i.e., the inverse of the injection rate, G is the "gap" per byte, i.e., the inverse bandwidth, and P is the number of processes.

Let $\mathcal{T}_{p2p}(s)$ denote the time that is needed to transmit a message of size s between two arbitrary PUs. Furthermore, let $\mathcal{T}_{coll}(s)$ denote the time that is needed to perform a global reduction operation where the result of a binary function $\mathcal{F}(a, b)$, applied to the values on all PUs, is returned on all PUs. This simplified network model was used in [27] to analyze the Δ-stepping algorithm on a distributed system. In the BSP [30] model, we would simply substitute $\mathcal{T}_{p2p}(s) = \mathcal{O}(l + g(s + \log P))$ and $\mathcal{T}_{coll}(s) = \mathcal{O}(\log p(l + gk))$. In LogGP, we could replace $\mathcal{T}_{p2p}(s) = 2o + L + g + (s-1)G$ if we assume that messages are sent in batches (g has to be charged) and $\mathcal{T}_{coll}(s) = \mathcal{O}(\log P) \cdot \mathcal{T}_{p2p}(s)$ for small s, if we assume an implementation based on point-to-point messages in a tree pattern.

Shortest Paths Calculation: Meyer *et al.* showed in [27] that for Δ-stepping on weighted random graphs, the number of phases $h = \mathcal{O}(\frac{\log^2 n}{\log\log n})$ whp. In our analysis, we assume the practically most interesting case $P \leq \frac{n}{h \log n}$. Each PU stores n/P rows of the adjacency matrix which contain whp, nd/P edges of $\mathcal{G}$. Vertices have a global identifier that can be used to determine the owner PU of that vertex in constant time (e.g., a hash function). It is intuitive that load balancing is already achieved due to the properties of the random graph $\mathcal{G}$. Whenever an edge (u, v) leading to a remote PU is relaxed, a relaxation request with the vertex and $dist(u) + w(u, v)$ is sent to the owner of v. If all requests are collected, and communicated at the end of each phase (i.e., only the shortest edge to a vertex v is communicated), then the runtime can be estimated by $\mathcal{O}(nd/P + h(\mathcal{T}_{coll}(1) + \mathcal{T}_{p2p}(dn/Ph)))$.

Calculating Successors given Predecessors: This step can simply be achieved by sending all remote edges to the PUs that own the target vertex and by "flipping" all edges, i.e., $\forall s \in V$ do $\forall v \in P_s[u]$: add u to $S[v]$. Each process has $\Theta(dn\frac{P-1}{P})$ remote edges evenly distributed on $P-1$ peer PUs whp. Thus, the expected time for this step is $\mathcal{O}(\mathcal{T}_{p2p}(dn\frac{P-1}{P^2}) \cdot P + dn/P)$.

Computing Path Counts: For this, we use the same distribution as in the previous step. The algorithm is started at s and runs until all queues on all PUs are empty. Dequeue operations act on a separate queue on each PU and enqueue operations enqueue the vertex at the owning PU by sending a control message. When a PU's queue is empty, it starts a new communication round and waits for the other nodes in order to check if all queues are empty. Communication complexity can be quantified by the maximum number of communication rounds r which is the longest path in the communication tree (i.e., when the last node finishes the algorithm).

Lemma 4. *The number of communication rounds $r = \mathcal{O}(\log n)$ whp.*

Proof: The longest path in the subgraph $\mathcal{G}'$ has as many edges as the diameter of the giant component of $\mathcal{G}$ which is $\mathcal{O}(\log n)$ whp. Only remote edges (edges leading to other PUs) cause message sends and can potentially cause a new communication round. $\mathcal{G}'$ is traversed in breadth first search order, and thus, whp, $r = \mathcal{O}(\log n)$. ■

Lemma 5. *The time to compute the shortest path counts on a distributed memory parallel machine is* $\mathcal{O}(n \log n/P + h(\mathcal{T}_{coll}(1) + \mathcal{T}_{p2p}(n \log n/Ph)))$.

Proof: The number of edges in $\mathcal{G}'$ is bounded by $n \log n$ whp (cf. Lemma 3). Thus, in a random graph, the expected number of edges per process is $n \log n/P$ and each edge is traversed once. Lemma 4 shows that a maximum of $r < h$ rounds are performed in the algorithm. Each of the h phases and the r rounds consists of a collective operation that checks if the next round/phase can be started and the same number of point-to-point message sends. It is easy to see that each edge is communicated at most once during the computation of the shortest path counts. As $P \to \infty$, this is also the expected communication volume. ∎

Updating Dependency and Centrality: This computation is similar to the computation of the shortest path counts, with the only difference that $\mathcal{G}'$ is traversed backwards. Lemma 4 can also be applied to this traversal. Each edge $e \in E'$ needs to be considered and the distributed queue implementation could enforce a collective synchronization operation during each round. Thus, the time required to update the dependency is of the the same order as for the path count computation in Lemma 5.

Theorem 3. *Betweenness centrality on random graphs from* $\mathcal{G}(n, \frac{d}{n})$ *with* $d = \mathcal{O}(\log n)$ *and random edge weights can be computed in expected time* $\mathcal{O}(n^2 \log n/P + nh(\mathcal{T}_{coll}(1) + \mathcal{T}_{p2p}(dn/Ph)))$ *whp on a distributed memory parallel machine with* P *processors.*

In unweighted graphs, where breadth first search can be used to compute all shortest paths and the predecessor set, the shortest path computation is bounded by $\mathcal{O}(nd/P + r(\mathcal{T}_{coll}(1) + \mathcal{T}_{p2p}(dn/Ph))$. Computing betweenness centrality is bounded by $\mathcal{O}(n^2 \log n/P + n \log n(\mathcal{T}_{coll}(1) + \mathcal{T}_{p2p}(dn/P \log n))$ whp.

IV. Simulation and Implementation

In this section, we evaluate the performance and scalability of the revised betweenness centrality algorithm. All performance evaluations were performed on the Indiana University Computer Science Department's "Odin" research cluster. Odin consists of 128 InfiniBand-connected compute nodes with 4 GiB memory each. We used a single process per node in our tests. We implemented our algorithm in the Parallel Boost Graph Library (PBGL) [18] and built it against Boost 1.42.0 [8] (containing the sequential BGL) in our tests.

Calculating betweenness centrality requires solving the SSSP problem starting at each vertex. For graphs with tens of millions of vertices or larger, solving n SSSPs is infeasible. We instead solve the SSSPs for a randomly chosen subset of vertices in V. In practice this approach generates a reasonably good approximation of the centrality scores for several real-world networks [3]. We assume in this case that one instance of the full graph needs all available memory.

	Unweighted $[s]$	Weighted $[s]$ ($C = 10^6$)
$d = 4$	8.35	11.21
$d = 8$	13.56	15.39
$d = 12$	16.82	16.85
$d = 30$	23.35	22.72

Table I: Performance of the sequential BGL implementation of Betweenness Centrality [9] ($n = 2^{20}$, time is per-source vertex).

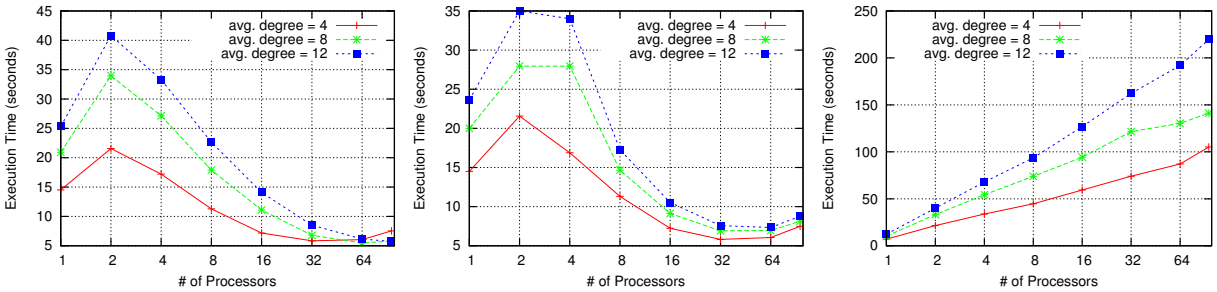
Although our focus lies on the parallel performance of Algorithm 2, we present the performance of Brandes' sequential algorithm in order to provide a baseline. Table I demonstrates the performance of the sequential algorithm as implemented in BGL. Due to the additional overhead imposed by the distributed data structures and communication code needed to support parallel computation, the parallel implementation is noticeably slower on a single processor. Differences in runtime between the weighted and unweighted version illustrate the tradeoff between the lower complexity of the SSSP algorithm in the unweighted case vs. smaller m' in the weighted case which reduces the complexity of subsequent stages of the algorithm. All single processor numbers beyond this table are the results of running the parallel algorithm with the required parallel data structures on a single processor.

The Δ parameter to the Δ-stepping algorithm determines the width of each bucket in the data structure used to sort edges by weight. This parameter determines the amount of work available to be performed in parallel at each step, and consequently how work-inefficient the algorithm is. We have set Δ to $\frac{maximum\ edge\ weight}{maximum\ vertex\ degree}$ for all results presented. It is likely that further tuning of the Δ parameter would lead to additional performance improvement.

A. Strong Scaling

To understand how well Algorithm 2 scales as more computational resources are provided, we evaluated the performance on fixed-size graphs.

Figures 5(a) and 5(b) illustrate the strong scalability (scaling with a fixed-size input) of Algorithm 2. Once the communication overhead is subsumed, the algorithm scales well until insufficient work is available to benefit from the additional resources. This occurs around 64 to 96 processors and depends both on the number of edges and number of equal weight paths. It is clear that the parallel implementation exhibits significant overhead due to parallel data structures and communication, in most cases, the parallel algorithm is not faster than



(a) Unweighted Strong ($n = 2^{20}$) (b) Weighted Strong ($C = 10^6$, $n = 2^{20}$) (c) Unweighted Weak ($n = 2^{19}$)

Fig. 5: Scaling of Algorithm 2 applied to Erdős-Rényi graphs (time is per source vertex).

the sequential version until 16 processors or more are available. Due to the generic programming techniques employed in the Parallel BGL most of the distributed data structures exhibit no overhead in the $P = 1$ case. One source of overhead that cannot be avoided is the determination of ownership of elements of the graph. In order to locate vertices and edges in the graph and their associated properties a processor performs either a computation or a lookup in an array that maps vertices to PUs. In the $P = 1$ case this lookup always returns the index of the only existing processor, but cannot be eliminated by the compiler. When the amount of work per vertex is small this ownership determination can have a large effect on the runtime of the algorithm.

B. Weak Scaling

To understand how the parallel implementation of Algorithm 2 scales as the problem size scales, we evaluated the performance of each algorithm on graphs where $n \propto m \propto P$. Weak scalability is perhaps the most appropriate test of Algorithm 2 because it illustrates the algorithm's ability to compute betweenness centrality on graphs which the sequential implementation is unable to process due to memory constraints per node. Figures 5(c) and 6(a) show that the runtime increases even though the amount of data per processor remains constant. This is because the time complexity of Algorithm 2 is $\mathcal{O}(\frac{n \log^3 n}{\log \log n})$. As we vary n linearly with the number of processors the amount of work increases faster than the number of processors. This yields more work per processor which gives rise to the sub-linear speedup exhibited.

C. Scale-Free Graphs

We also evaluated the performance of Algorithm 2 on scale-free graphs which are representative of real-world networks. We used the Recursive MATrix (R-MAT) [10] random graph generation algorithm to generate input data sampled from a Kronecker product.

Figure 6(b) shows that Algorithm 2 scales relatively well in the unweighted case, though adding more than 16 processors does not decrease the runtime. This leveling off of performance occurs earlier with R-MAT graphs than with Erdős-Rényi graphs, possibly due to the smaller diameter of the graph.

Figure 6(c) shows that Algorithm 2 is able to compute betweenness centrality on R-MAT graphs too large to fit in the memory of a single machine. As with Erdős-Rényi graphs the amount of work increases faster than the number of processes leading to sub-linear scaling. The smaller diameter of the R-MAT graphs means that the maximum size of the Δ-stepping bucket data structure is greater than with Erdős-Rényi graphs, which can lead to paging and thus reduced performance at large processor counts (this is particularly evident in the $avg.\ degree = 12$ plot in Figure 6(c)).

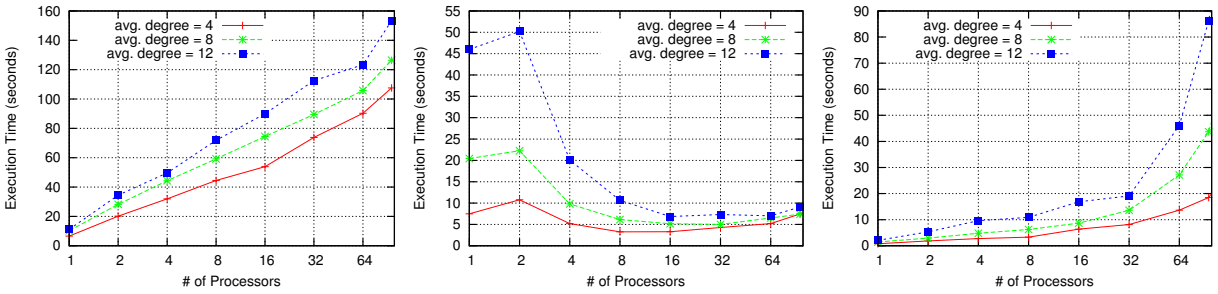
Performance results on weighted R-MAT graphs were omitted due to space constraints.

V. Conclusions and Future Work

We have presented a new parallel algorithm for betweenness centrality that has expected time in a CRCW PRAM equal to the sequential algorithm by Brandes. Rather than parallelizing betweenness centrality by solving multiple single source shortest paths problems at once, we have exposed parallelism within the shortest paths computation by leveraging existing label-correcting single-source shortest path algorithms. This method allows us to demonstrate good parallel performance while maintaining low space complexity.

This algorithm has lower time complexity on sparse graphs than solutions which utilize all-pairs shortest path algorithms such as Floyd-Warshall. In addition this algorithm has low space complexity relative to all-pairs shortest paths algorithms which makes it especially suitable the analysis of very large graphs on distributed memory machines.

We have presented results on Erdős-Rényi and R-MAT random graphs which demonstrate that our algorithm is both computationally efficient and scalable. Greater speedup could be achieved by leveraging fine-grained parallelism at the node level. Hybrid approaches



(a) Weighted ER Weak ($C = 10^6$, $n = 2^{19}$) (b) Unweighted RMAT Strong ($n = 2^{22}$) (c) Unweighted RMAT Weak ($n = 2^{19}$)

Fig. 6: Scaling of Algorithm 2 applied to Erdős-Rényi (ER) and RMAT graphs (time is per source vertex).

leverage fine-grained parallelism to maximize use of local resources and coarse-grained parallelism to allow problem scaling and provide additional performance benefits. Hybrid approaches will become increasingly important given increasing problem size and the growing processing capability of individual machines. We are currently developing a new version of the Parallel BGL which leverages hybrid parallelism and reduces overhead in the communication layer and anticipate presenting new performance results in the near future.

Acknowledgements

This work was supported by a grant from the Lilly Endowment. The authors would also like to thank Jeremiah Willcock for useful discussions regarding proofs.

REFERENCES

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model. *Journal of Par. and Distrib. Computing*, 44(1):71–79, 1995.

[2] J. Anthonisse. The rush in a directed graph. Technical Report BN9/71, Stichting Mathematisch Centrum, Amsterdam, 1971.

[3] D. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Alg. and Models for the Web-Graph*, volume 4863 of *LNCS*, pages 124–137. Springer-Verlag, 2007.

[4] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *International Conference on Parallel Processing*, pages 539–550, 2006.

[5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] V. D. Blondel, J.-L. Guillaume, J. M. Hendrickx, and R. M. Jungers. Distance distribution in random graphs and application to network exploration. *Physical Review E*, 76(066101), 2007.

[7] B. Bollobas. *Random Graphs*. Cambridge University Press, 2001.

[8] Boost. *Boost C++ Libraries*. http://www.boost.org/.

[9] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of 4th International Conference on Data Mining*, pages 442–446, April 2004.

[11] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[12] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 722–731. Springer, 1998.

[13] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Piscataway, NJ, November 2006.

[15] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[16] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, March 1977.

[17] A. Frieze and G. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10:57–77, 1985.

[18] D. Gregor, N. Edmonds, A. Breuer, P. Gottschling, B. Barrett, and A. Lumsdaine. The Parallel Boost Graph Library. `http://www.osl.iu.edu/research/pbgl`, 2005.

[19] R. Hassin and E. Zemel. On shortest paths in graphs with random weights. *Mathematics of Operations Research*, 10(4):557–564, November 1985.

[20] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[21] J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 713–716, 1987.

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

[23] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest path problem. In *International Conference on Parallel Processing*, pages 124–138, 1991.

[24] K. Madduri, D. Bader, J. Berry, and J. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, New Orleans, LA, January 2007.

[25] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Multithreaded Architectures and Applications (MTAAP 2009)*. IEEE Computer Society, May 2009.

[26] U. Meyer. *Design and analysis of sequential and parallel single-source shortest-paths algorithms*. PhD thesis, Universität Saarbrücken, 2002.

[27] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, 2003.

[28] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Washington, DC, USA, 1985. IEEE.

[29] G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31(4):581–603, December 1966.

[30] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[31] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[32] Q. Yang and S. Lonardi. A parallel algorithm for clustering protein-protein interaction networks. In *Computational Systems Bioinformatics Conference – Workshops*, pages 174–177, Washington, DC, USA, 2005. IEEE Computer Society.