Computing Betweenness Centrality in External Memory

Lars Arge
MADALGO
Dept. of Computer Science
Aarhus University
Email: large@madalgo.au.dk

Michael T. Goodrich
Dept. of Computer Science
School of Info. & Comp. Sci.
University of California, Irvine
Email: goodrich@ieee.org

Freek van Walderveen
MADALGO
Dept. of Computer Science
Aarhus University
Email: freek@madalgo.au.dk

***Abstract*—Betweenness centrality is one of the most well-known measures of the importance of nodes in a social-network graph. In this paper we describe the first known external-memory and cache-oblivious algorithms for computing betweenness centrality. We present four different external-memory algorithms exhibiting various tradeoffs with respect to performance. Two of the algorithms are cache-oblivious. We describe general algorithms for networks with weighted and unweighted edges and a specialized algorithm for networks with small diameters, as is common in social networks exhibiting the "small worlds" phenomenon.**

***Keywords*-betweenness centrality; external memory; social networks;**

I. Introduction

A valuable component of social network analysis involves assigning numerical scores to each vertex in a network based on the "influence" or "importance" of that node in the network, which is commonly referred to as that node's *centrality*. Nodes with high centrality scores are considered to represent entities with high influence or importance. For instance, such nodes are often considered to play a crucial role in the flow of commodities (such as information, drugs, disease, or technology) in their network. Likewise, vertices with low centrality scores are considered to exert relatively less influence and have less of an impact on flow.

One of the most well-known centrality measures is *betweenness centrality* (e.g., see [1]–[8]). Given an undirected graph $G = (V, E)$, the betweenness centrality of a vertex $v \in V$ is defined as

$$C_B(v) := \frac{1}{2} \cdot \sum_{s \neq t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}} \ ,$$

where σ_{st} is the number of shortest paths between s and t ($\sigma_{ss} = 1$ by convention) and $\sigma_{st}(v)$ is the number of shortest paths between s and t that contain v. Since $\sigma_{st}(v) = 0$ if s and t are in different connected components, we consider connected (and undirected) graphs in this paper.

There are also other definitions of centrality, ranging from simple statistics, like degree centrality [9] and closeness centrality [10], to sophisticated statistics, like Katz centrality [11] and random-walk/current-flow centrality [12], [13]. Compared to these other measures, betweenness centrality is particularly useful in measuring the importance of vertices for the sake of the flow of information (or other commodities) along geodesic paths. For instance, it has been used as a way to identify interdisciplinary journals in scientific collaboration networks [7]. From a big-data algorithmic perspective betweenness centrality is an interesting measure as well, since it is neither trivial to compute nor as hard as measures like Katz centrality and random-walk centrality, which seem to require matrix inversion computations.

Of course, focusing on large networks from an algorithmic perspective implies that we should accommodate the likely situation when the space required for our algorithm is larger than our computer's internal memory. Thus, it is useful to develop betweenness centrality algorithms in the *external-memory* framework [14], [15]. Recall that in this framework, we assume that memory is subdivided into blocks of size B and that input and output (I/O) operations on such blocks are atomic operations. Moreover, we assume that the cost of these I/O operations is so large that it is most useful to characterize algorithmic performance in terms of the number of such operations performed, assuming an internal memory of size M connected to an external memory device that supports block transfers to/from internal memory as a single I/O operation. Therefore, our focus in this paper is on external-memory algorithms for computing the betweenness centrality of each vertex in a large network.

A. Related Prior Work

The best internal algorithm for computing betweenness centrality is due to Brandes [3] and runs in $O(V^2 \log V + VE)$ time for weighted graphs and $O(VE)$ time for unweighted graphs[1]. Unfortunately, the algorithm, which we review in Section II, does not easily translate into an efficient external-memory algorithm, as it involves a large number of pointer hops (random memory accesses).

Although main memory size is often the primary constraint on the size of the graphs for which one can compute betweenness centrality efficiently in practice [3], the betweenness centrality problem has not previously been

[1]We use "V" and "E" to denote the set of vertices and edges in a graph, respectively, as well as the sizes of these sets, whenever the context is clear as to whether we are referencing these as sets or numbers.

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

considered in the external-memory model. The external-memory results of most relevance to this paper are results on shortest paths in undirected graphs (a thorough survey of results can be found in [16]). In the unweighted (breadth-first search) case the best know algorithms, due to Munagala and Ranade [17] and Mehlhorn and Meyer [18], use $O(V + \text{SORT}(E))$ and $O(\sqrt{VE/B} + \text{SORT}(E))$ I/Os, respectively, where $\text{SORT}(N) = O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ is the number of I/Os needed to sort N elements [14]. Algorithms for the unweighted all-pairs shortest-paths (all-pairs breadth-first search) problem using $O(V \cdot \text{SORT}(E))$ I/Os have also been developed by Arge et al. [19] and Chowdhury and Ramachandran [20]. The latter algorithm can also compute the unweighted diameter using only $O(E)$ space. In the weighted case the best known single-source shortest-paths algorithm is due to Kumar and Schwabe [21] and uses $O(V + \frac{E}{B}\log\frac{E}{B})$ I/Os. The best known algorithm for the weighted all-pairs shortest-paths problem is due to Chowdhury and Ramachandran [20] and uses $O(V \cdot \sqrt{VE/B} + V\frac{E}{B}\log\frac{E}{B}))$ I/Os. The latter algorithm computes the weighted diameter using $O(V^2)$ space. It can also be computed using only $O(E)$ space (but $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/Os) by running the Kumar and Schwabe algorithm [21] V times.

Sometimes it is possible to design I/O-efficient algorithms that do not make explicit use of knowledge of the parameters B and M. Such so-called *cache-oblivious* algorithms that work efficiently without explicit tuning for the memory parameters are efficient on all levels of even an unknown multilevel memory hierarchy (e.g., see [22]). Cache-oblivious algorithms have been developed for a number of problems, including for sorting [22] and various tree problems [23], where the same $O(\text{SORT}(N))$ I/O-bound can be achieved as in the classic external-memory model. For the unweighted single-source and all-pairs (breadth-first search) problems, cache-oblivious algorithms that match the previously mentioned cache-aware bounds, are known as well [20], [23].

B. Our Results

In this paper, we provide I/O-efficient and cache-oblivious algorithms for computing betweenness centrality, both in weighted and unweighted graphs. As mentioned above, to the best of our knowledge, no such algorithms were previously known, even though I/O is often a constraint on the size of the graphs one can compute betweenness centrality on in practice. In particular, we present algorithms for weighted and unweighted graphs, as well as algorithms that can exploit additional properties in the input, including sparsity and low-diameter, which are common in social network applications. In addition, two of our algorithms, namely the one for unweighted graphs and the one for graphs with low-diameter, are cache-oblivious. Not surprisingly, all of our algorithms build on methods for performing all-pairs shortest-path computations, adding several additional components for the sake of I/O-efficiency and dovetailing these with the additional complexities that come from computations of betweenness centrality scores. Additional techniques that we introduce, which may be of use in other I/O-efficient graph algorithms, include the following:

- annotated incremental BFS construction
- K-restricted parallel execution of slim buffer heaps
- external-memory simulation of Bellman-Ford-type distributed computations.

Specifically, in Section III, we consider unweighted graphs and describe how to obtain a betweenness centrality algorithm using $O(V \cdot \text{SORT}(E))$ I/Os and $O(E)$ space. This algorithm is cache-oblivious. In Section IV, we consider weighted graphs and describe a number of new external-memory algorithms. First, we show how to obtain an $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/O and $O(E)$ space algorithm, which works for general graphs but is admittedly not cache-oblivious. Next we describe an improved $O(V \cdot \sqrt{VE/B} + V \cdot \frac{E}{B}\log\frac{V}{B})$ I/O and $O(V \cdot \sqrt{VE/B})$ space algorithm for sparse graphs (where $E < VB/\log V$). The algorithm performs several phases with a number of concurrent instances of a shortest-path algorithm rather than just one phase with V concurrent instances. Interestingly, this algorithm also improves the space of the best known weighted diameter algorithm [20] from $O(V^2)$ to $O(V \cdot \sqrt{VE/B})$, which may be of independent interest. Finally, we describe a further improved algorithm for low-diameter graphs, which are common for social networks that exhibit the "small world" phenomenon. This algorithm uses $O(VE \cdot \text{diam}(G)/B)$ I/Os and $O(V^2)$ space on a graph G with (unweighted) diameter $\text{diam}(G)$; the algorithm is cache-oblivious.

Our methods involve a number of new techniques with respect to the theory of external-memory algorithms for computing betweenness centrality for big-data social networking applications. Thus, although we do not include in this paper an experimental analysis of our algorithms, our results nevertheless provide a first step towards the full implementation of I/O-efficient betweenness centrality algorithms for large social networks.

II. The Betweenness Centrality Algorithm of Brandes

In this section we review the internal-memory algorithm of Brandes [3] for computing betweenness centrality of a graph $G = (V, E)$. The main idea of the algorithm is to decompose the betweenness centrality of a node v into the contributions of the shortest paths passing through v in G that start in each of the other nodes. The sum of the contributions of the shortest paths starting in a node $s \in V \setminus \{v\}$ to the betweenness centrality of v is called the *dependency* of s on v, denoted $\delta_s(v)$, and is defined as follows:

$$\delta_s(v) := \sum_{t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}} .$$

The betweenness centrality of a vertex v is then equal to the sum of the dependencies of all vertices on v:

$$C_B(v) = \frac{1}{2} \cdot \sum_{s \neq t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{1}{2} \cdot \sum_{s \in V \setminus \{v\}} \delta_s(v) \ .$$

Below we describe an $O(E)$-time and space algorithm for computing the dependency values $\delta_s(v)$ for all $v \in V$ for a particular $s \in V$ in an unweighted graph G (a graph where all edges have weight one). To compute $C_B(v)$ for all $v \in V$ we simply run this algorithm for each $s \in V$, and maintain for each $v \in V$ the sum of all $\delta_s(v)$ computed so far. Thus overall the betweenness centrality is computed for all vertices in an unweighted graph G in $O(VE)$ time using $O(E)$ space. At the end of the section we discuss how to extend the algorithm to the weighted case.

Computing $\delta_s(v)$ for fixed s: Let $P_s(v)$ be the set of *predecessors* of v with respect to s, that is, the neighbours of v that are part of at least one shortest path between s and v, and $S_s(v)$ the set of *successors* of v with respect to s, that is, the set of vertices w that are neighbours of v such that v is on at least one shortest path from s to w. We now have the following:

Lemma 1 ([3], Lemma 3): For $v \neq s$,

$$\sigma_{sv} = \sum_{u \in P_s(v)} \sigma_{su} \ .$$

Lemma 2 ([3], Theorem 6): For $v \neq s$,

$$\delta_s(v) = \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w)) \ .$$

The algorithm of Brandes [3] for computing $\delta_s(v)$ for all $v \in V$ in $O(E)$ time and space first computes $P_s(v)$ and $S_s(v)$ for all $v \in V$ in $O(E)$ time and space using a breadth-first search from s. During the breadth-first search σ_{sv} can also easily be computed when visiting node v using Lemma 1, since all vertices $u \in P_s(v)$ are visited before v. Finally, the vertices are visited in reverse breadth-first search order and $\delta_s(v)$ is computed when visiting v using Lemma 2; this is possible since all vertices $w \in S_s(v)$ have already been visited when visiting v.

Weighted graphs: The above algorithm can easily be modified to work for weighted graphs, where we are given a weight function $w : E \to \mathbb{R}^+$. To construct $P_s(v)$ and $S_s(v)$ for all v we use Dijkstra's algorithm [24], rather than a breadth-first search. The rest of the algorithm remains unchanged, so the total running time becomes $O(VE + V^2 \log V)$, since Dijkstra's algorithm uses $O(E + V \log V)$ time.

III. An I/O-Efficient Algorithm for Unweighted Graphs

In this section, we show how to compute betweenness centrality for the vertices of an unweighted graph $G = (V, E)$ in $O(V \cdot \text{SORT}(E))$ I/Os and $O(E)$ space. Similar to the internal-memory algorithm of Brandes [3], our external-memory algorithm computes a breadth-first search (BFS) tree from each vertex s in G while using Lemma 1 to compute σ_{sv} for each vertex $v \neq s$ (while also maintaining for each vertex the sum of all $\delta_s(v)$ computed so far). However, unlike the internal case we do not construct the BFS trees for each vertex $s \in V$ independently, since even with the best known external BFS algorithm [18] this would require $\Omega(V \cdot \sqrt{VE/B})$ I/Os. Instead we modify the algorithm by Chowdhury and Ramachandran [20] that constructs BFS trees from all vertices s in $O(V \cdot \text{SORT}(E))$ I/Os in total. The main idea of the algorithm is that a BFS tree from vertex s can be constructed more I/O-efficiently when a BFS tree from a vertex s' close to s is already known. This allows us to perform reverse-order I/O-efficient processing of the BFS trees.

The algorithm first computes an ordering π of the vertices of G such that the distance between consecutive vertices in π is small, and then it constructs the BFS trees from the vertices in π one after the other, utilizing an *incremental BFS* algorithm. Below we first describe how to compute π and then we describe the incremental BFS algorithm. After that we discuss how to modify the BFS algorithm to compute betweenness centrality, and finally we analyse the algorithm.

Computing π: The ordering π of the vertices used in the algorithm by Chowdhury and Ramachandran [20] is constructed as follows. First an (arbitrary) spanning tree T of G is computed and then two directed edges (u, v) and (v, u) are constructed for each undirected edge (u, v) in T to obtain directed graph T'. Then the edges of T' are ordered to form an Euler tour and the order $\pi = \langle s_1, \ldots, s_{|V|} \rangle$ of the vertices in V is defined by the order in which they first occur in the Euler tour.

Note that although the distance $d(s_i, s_{i+1})$ between two consecutive vertices s_i and s_{i+1} in π may be large, the sum of the distances between all consecutive vertices in π is $O(V)$ since the length of the Euler tour is $O(V)$. Furthermore, Chowdhury and Ramachandran [20] also prove the following property of π:

Lemma 3 ([20]): Let $d(u, v)$ be the length of the shortest path between u and v, and $d_i = d(s_{i-1}, s_i)$ for $1 < i \leq V$. Then for any vertex $v \in V$, $d(s_i, v) - d_i \leq d(s_{i-1}, v) \leq d(s_i, v) + d_i$.

Incremental BFS: The main idea in the incremental BFS algorithm for computing a BFS tree from s_i given a BFS tree from s_{i-1} is to order the adjacency lists of all vertices according to their level in the BFS tree for s_{i-1}. More precisely, assume the vertices in G are labelled with their distance from s_{i-1} in the BFS tree from s_{i-1} and store the adjacency lists of vertices at distance d in a list $\mathcal{A}_d$; within $\mathcal{A}_d$ the adjacency lists are ordered by vertex. Lemma 3 above then implies that we are guaranteed to find the adjacency lists for vertices at distance ℓ from s_i in some list $\mathcal{A}_k$ for $\ell - d_i \leq k \leq \ell + d_i$. Using this property we now

construct the BFS tree from s_i one level at a time as in the algorithm by Munagala and Ranade [17]: Let $\mathcal{L}_\ell$ denote the sorted list of vertices at level ℓ in the BFS tree from s_i; level $\mathcal{L}_1$ consists of vertex s_i. Given the previous two BFS levels $\mathcal{L}_{\ell-2}$ and $\mathcal{L}_{\ell-1}$, we construct $\mathcal{L}_\ell$ by first scanning $\mathcal{L}_{\ell-1}$ simultaneously with each list $\mathcal{A}_k$, where $\ell-d_i \leq k \leq \ell+d_i$, and inserting an edge (u,v) in a list $\mathcal{E}_\ell$ for each vertex v that is a neighbour of a vertex $u \in \mathcal{L}_{\ell-1}$. We then sort $\mathcal{E}_\ell$ by the second vertex and scan it simultaneously with $\mathcal{L}_{\ell-1}$ and $\mathcal{L}_{\ell-2}$ to remove any edges $(u,v) \in \mathcal{E}_\ell$ for which $v \in \mathcal{L}_{\ell-1}$ or $v \in \mathcal{L}_{\ell-2}$. This means that $\mathcal{E}_\ell$ now contains edges (u,v) connecting a vertex u in level $\ell-1$ with a vertex v in level ℓ of the BFS tree. Finally, we scan the sorted $\mathcal{E}_\ell$ to aggregate all edges $(u_1,v),\ldots,(u_m,v)$ for a vertex v and append v to $\mathcal{L}_\ell$.

Annotated Incremental BFS: To compute betweenness centrality, we first modify the incremental BFS algorithm above such that it also computes σ_{s_iv} for each vertex v when computing the BFS tree from s_i. To do so we augment each vertex u in the sorted list $\mathcal{L}_{\ell-1}$ of vertices at level $\ell-1$ in the BFS tree from s_i with σ_{s_iu}. We then modify our algorithm for computing $\mathcal{L}_\ell$ from $\mathcal{L}_{\ell-1}$ and $\mathcal{L}_{\ell-2}$ in order to be able to compute σ_{s_iv} for each vertex v in $\mathcal{L}_\ell$. The modification consists of *annotating* each edge (u,v) in $\mathcal{E}_\ell$ with σ_{s_iu}, which allows us to compute σ_{s_iv} using Lemma 1 when aggregating edges $(u_1,v),\ldots,(u_m,v)$ and inserting v in $\mathcal{L}_\ell$. The annotation is performed simply by sorting the list $\mathcal{E}_\ell$ of edges (u,v) by first vertex and then simultaneously scanning through $\mathcal{E}_\ell$ and $\mathcal{L}_{\ell-1}$ to annotate each edge (u,v) in $\mathcal{E}_\ell$ with σ_{s_iu} from $\mathcal{L}_{\ell-1}$.

Reverse-Order BFS Processing: After constructing the BFS tree from s_i along with σ_{s_iv} for each vertex v, that is, the lists $\mathcal{L}_1,\mathcal{L}_2,\ldots$ of vertices at each level of the tree and the lists $\mathcal{E}_2,\mathcal{E}_3,\ldots$ of edges between the levels, we compute $\delta_{s_i}(v)$ for each vertex as in Brandes' algorithm [3] by visiting the vertices (levels) in reverse order. To compute $\delta_{s_i}(v)$ for all vertices u in $\mathcal{L}_{\ell-1}$ when $\delta_{s_i}(v)$ has already been computed for all vertices v in $\mathcal{L}_\ell$, we first annotate each edge (u,v) in $\mathcal{E}_\ell$ with $\delta_{s_i}(v)$ and σ_{s_iv} by sorting $\mathcal{E}_\ell$ by second vertex and simultaneously scanning $\mathcal{E}_\ell$ and $\mathcal{L}_\ell$. After that we sort $\mathcal{E}_\ell$ by first vertex and scan the sorted list to obtain all edges $(u,v_1),\ldots(u,v_m)$ incident to each vertex u in $\mathcal{L}_{\ell-1}$ and compute $\delta_{s_i}(u)$ using Lemma 2.

I/O complexity: To analyse our algorithm, first note that the ordering π can be computed in $O(V \cdot \text{SORT}(E))$ I/Os using a minimal spanning tree algorithm by Arge et al. [25] and standard external graph algorithm techniques [16], [26] (see for example Chowdhury and Ramachandran [20]).

To initiate the incremental BFS process, we generate the BFS tree from s_1 in $O(V + \text{SORT}(E))$ I/Os using the algorithm by Munagala and Ranade [17]. Now to construct the BFS tree from s_i and compute $\delta_{s_i}(v)$ for all vertices $v \in V$, we sort the adjacency lists into lists $\mathcal{A}_k$ using $O(\text{SORT}(E))$ I/Os and then we construct the levels of the BFS tree one at a time and traverse them again in reverse order. For level ℓ, we scan lists $\mathcal{A}_{\ell-d_i},\ldots,\mathcal{A}_{\ell+d_i}$, so in total we scan each list $\mathcal{A}_k$ $O(d_i)$ times, using $O(d_i \cdot E/B)$ I/Os. For each level we also scan $\mathcal{L}_{\ell-1}$ once, so every level of the BFS tree is also scanned $O(d_i)$ times, using $O(d_i \cdot V/B)$ I/Os in total. The remainder of the algorithm consists of scanning and sorting the vertex and edge sets a constant number of times using $O(d_i \cdot E/B + \text{SORT}(E))$ I/Os in total. Overall we use $O(\sum_i (d_i \cdot E/B + \text{SORT}(E))) = O(V \cdot (E/B + \text{SORT}(E))) = O(V \cdot \text{SORT}(E))$ I/Os in total to construct and traverse all of the BFS trees. The amount of space required is $O(E)$, as we only store the set of edges and vertices a constant number of times.

Theorem 1: Betweenness centrality for an unweighted and undirected graph $G = (V,E)$ can be computed in $O(V \cdot \text{SORT}(E))$ I/Os using $O(E)$ space.

Remark: Using a cache-oblivious minimum spanning tree and Euler tour construction algorithm [23] rather than the I/O-efficient algorithms [25], [26], our algorithm can easily be made cache-oblivious with the same I/O and space bounds, since the incremental BFS algorithm is based only on sorting and scanning of lists.

IV. I/O-Efficient Algorithms for Weighted Graphs

In Section II we noted that Brandes' internal-memory algorithm can also be used for graphs $G = (V,E)$ with positive edge weights $w : E \to \mathbb{R}^+$, by using a shortest-path algorithm instead of breadth-first search. Thus we can obtain an $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/O and $O(E)$ space algorithm by running a simple modification of the single-source shortest-path algorithm of Kumar and Schwabe [21] from all V vertices. We describe this algorithm in Section IV-A. In Section IV-B, we then describe an improved $O(V \cdot (\sqrt{VE/B} + (E\log V)/B))$ I/O and $O(V \cdot \sqrt{VE/B})$ space algorithm for sparse graphs where $E < VB/\log V$. The algorithm is a modified version of the previous all-pairs shortest-paths algorithm by Chowdhury and Ramachandran [20] and by Arge et al. [19] that computes shortest paths from multiple sources concurrently. In Section IV-C, we consider the case of low-diameter graphs and describe an $O(VE \cdot \text{diam}(G)/B)$ I/O and $O(V^2)$ space algorithm, where $\text{diam}(G)$ is the (unweighted) diameter of G.

A. Time-Forward Processing with External-Memory Priority Queues

In this section, we describe our general algorithm for computing betweenness centrality in a weighted graph, based on the use of time-forward processing. Our method builds on the I/O-efficient single-source shortest-path algorithm of Kumar and Schwabe [21], dovetailing time-forward processing to compute betweenness centrality scores.

The algorithm of Kumar and Schwabe is a modified version of Dijkstra's algorithm that relies on an I/O-efficient

priority queue called the *external tournament tree*, which supports a DECREASEKEY operation with a slightly unusual semantic: Given an element x and a priority p, the operation updates the priority of x to p only if p is smaller than the current priority of x (inserting x if it does not exist in the queue).

Lemma 4 ([19], [21]): Given a universe of N elements, an external tournament tree using $O(B)$ internal memory can process a sequence of k DELETEMIN, DELETE, and DECREASEKEY operations in $O(\frac{k}{B}\log\frac{N}{B})$ I/Os in total.

I/O-efficient Dijkstra: To compute the shortest paths from vertex s to all other vertices, Dijkstra's algorithm [24] processes vertices in order of their distance from s while maintaining a tentative shortest distance $d_t(s,v)$ from s to all non-processed vertices. When processing a vertex v, each neighbour u of v is retrieved and if $d_t(s,u) > d(s,v) + w(v,u)$ the tentative distance to u is updated. To find the next vertex to process, unprocessed vertices are kept in a priority queue with their tentative distance as their priority.

The two main challenges faced by external-memory implementations of Dijkstra's algorithm are to determine which neighbour vertices are already processed when processing a vertex, and which unprocessed neighbours need to have their priority updated (decreased). The I/O-efficient shortest-path algorithm of Kumar and Schwabe [21] solves the second issue using the special semantic of the external tournament tree. The first issue is solved by updating the priority of a vertex irrespective of whether that vertex has already been processed, and using a second tournament tree to take care of removing *spurious updates*, that is, updates to already processed vertices, from the first one. Refer to [16], [21] for details[2]. This results in an algorithm using $O(V + \frac{E}{B})$ I/Os in total for accessing the adjacency lists of the vertices, and $O(E)$ operations on the two tournament trees using $O(\frac{E}{B}\log\frac{E}{B})$ I/Os (Lemma 4). Hence, in total the algorithm uses $O(V + \frac{E}{B}\log\frac{E}{B})$ I/Os for computing the shortest paths from one source.

Computing betweenness centrality: For computing betweenness centrality, we first describe how we can compute the number of shortest paths σ_{sv} and then the dependency $\delta_s(v)$ of s on each vertex v after having computed $d(s,v)$ for all vertices using the Kumar and Schwabe algorithm. To do so we first compute the actual directed acyclic graph (DAG) of shortest paths from s by identifying all edges (v,u) where $d(s,u) = d(s,v) + w(v,u)$. We can easily do so by annotating edge (v,u) with $d(s,v)$ and $d(s,u)$ in a few sorting and scanning steps of the edges and vertices, similar to the way we annotated edges in the algorithm in Section III. Next we compute σ_{sv} for each vertex v by traversing the vertices in order of increasing shortest path distance $d(s,v)$, and for a vertex v summing σ_{sw} for each predecessor w of v (edge (w,v)) in the DAG (Lemma 1). We do so I/O-efficiently using the *time-forward processing* technique [26]: Initially we sort the edges by the shortest path distance of their first vertex (and secondarily by second vertex) and insert s annotated with $\sigma_{ss} = 0$ in an external priority queue [27] with priority $d(s,s) = 0$. Then we visit the vertices in increasing shortest-path order, and repeatedly obtain σ_{sw} for each predecessor w of vertex v by performing DELETEMIN operations on the priority queue, computing σ_{sv}, and then accessing all edges (v,u) in the sorted list of edges and inserting each successor u of v in the priority queue, annotated with σ_{sv}. Overall the algorithm will scan through the sorted list of edges and perform $2E$ INSERT and DELETEMIN operations on the priority queue. After having computed σ_{sv} for each vertex v, we can compute $\delta_s(v)$ for each vertex v in a similar way using Lemma 2 by processing the vertices in reverse order of shortest path distance using time-forward processing. Overall the algorithm uses $O(\text{SORT}(E))$ I/Os to compute σ_{sv} and $\delta_s(v)$, since it performs a constant number of scans and sort steps along with $2E$ priority queue operations, which can be performed in $\text{SORT}(E)$ I/Os [27].

Now to compute the betweenness centrality of each vertex v we simply run the above algorithm with each vertex as source s, and compute $C_B(v)$ by maintaining the partial sum of all $\delta_s(v)$, for all $v \in V$ as in the unweighted case in Section III. Overall we use $O(V \cdot (V + \frac{E}{B}\log\frac{E}{B} + \text{SORT}(E))) = O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/Os and $O(E)$ space.

Theorem 2: Betweenness centrality for an undirected graph $G = (V,E)$ with positive edge weights can be computed in $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/Os and $O(E)$ space.

B. The Algorithm for Sparse Weighted Graphs

The I/O bound in Theorem 2 is dominated by the $O(V^2)$ term in case the graph is sparse (when $E < VB/\log V$). The term is a result of each of the V individual shortest path computations making V accesses to the adjacency lists. In this section we describe an algorithm that reduces the I/O cost when $E < VB/\log V$ at the expense of using more space. More precisely, we show how to compute betweenness centrality in $O(V \cdot (\sqrt{VE/B} + (E\log V)/B))$ I/Os using $O(V \cdot \sqrt{VE/B})$ space.

Our algorithm builds on the all-pairs shortest-paths algorithms of Chowdhury and Ramachandran [20] and of Arge et al. [19], adding a technique based on K-restricted parallel execution of slim buffer heaps. The idea is to run multiple instances of Kumar and Schwabe's version of Dijkstra's algorithm simultaneously to allow for faster access to the adjacency lists. However, in this case we may not have enough space in main memory for the $\Theta(B)$ space of each of the external tournament trees.

[2]The algorithm as presented by Kumar and Schwabe only works under the assumption that no two vertices have the same shortest-path distance to s. However, this assumption can be removed by carefully managing the elements in the priority queues to make sure that such vertices are handled at the same time [16].

Therefore, we below first describe how to modify the external tournament tree to only use $O(B/L)$ space in main memory for a parameter $L < B$. We then run multiple shortest path algorithms simultaneously, using only L instances of the modified tournament tree at the same time, so we save $O(L)$ accesses to load the single priority queues.

Priority queue buffers: We now show how an $O(B/L)$-sized buffer in internal memory can help reduce the I/O cost for a sequence of operations. Let Q be an external tournament tree as in Lemma 4, without using any space in internal memory. We store a *tournament tree buffer* for Q containing a set $s(Q)$ of the $|s(Q)| = O(B/L)$ smallest items in Q as well as an operation buffer of size $O(B/L)$. For any operation on Q we first try to perform it on the buffer of Q, which we assume to be loaded in memory (while the root of Q is still on disk). For a DELETEMIN operation, we extract the minimum item from $s(Q)$ and return it. In case we run out of items, we load the root of Q and perform $O(B/L)$ DELETEMIN operations on Q to refill $s(Q)$. For a DECREASEKEY operation we first check whether the item occurs in $s(Q)$ and update it, otherwise we store the operation in the operation buffer. In case the operation buffer is full, we load the root of Q in memory and perform all $O(B/L)$ buffered operations on Q. A DELETE operation is handled in the same way.

Lemma 5: The external tournament tree can be implemented to use $O(B/L)$ internal memory such that a sequence of k operations can be processed in $O(k \cdot L/B + k/B \log(N/B))$ I/Os in total.

K-Restricted Parallel Execution of Slim Buffer Heaps: Unlike the previous all-pairs shortest-paths algorithms [19], [20] that run all V instances of the Kumar and Schwabe algorithm [21] simultaneously, we only run K instances simultaneously. We divide these instances into K/L groups using $2 \cdot L$ slim buffer heaps each. Note that this allows us to write the $L \cdot O(\frac{B}{L})$ internal memory used by the L priority queues to disk, and read it again from disk, in a constant number of I/Os. We now perform the V rounds (each processing one vertex) of the Kumar and Schwabe algorithm one round at a time simultaneously for all K instances. A round starts by loading the priority queue blocks for each of the K/L groups in order, and for each group use the priority queues to determine the next vertex to be processed in the round for L of the instances. This results in a list $\mathcal{V}$ with a node to be processed for each of the K instances. To find the adjacency lists of all the vertices in $\mathcal{V}$, we then sort $\mathcal{V}$ by vertex and scan it simultaneously with the adjacency lists (also sorted by vertex). This produces a list $\mathcal{A}$ of the relevant neighbour vertices for all K instances. We then sort the vertices in $\mathcal{A}$ according to the group and instance they belong to. We end the round by scanning $\mathcal{A}$ while loading the priority queue blocks for each of the K/L groups in order, and update the priority queues using the information in $\mathcal{A}$. After finishing a set of K instances, we compute the number of shortest paths σ_{sv} and the dependency values $\delta_s(v)$ for these instances as in Section IV-A, that is, we build for each instance the shortest-path DAG from its source s and use time-forward processing on this DAG to first compute σ_{sv} for all $v \in V$, and then $\delta_s(v)$ for all $v \in V$ in a reverse time-forward processing step.

Analysis: To run all V instances of the Kumar and Schwabe algorithm we perform $\frac{V}{K} \cdot V = \frac{V^2}{K}$ rounds in total. In each of these rounds we use $O(K/L)$ I/Os to read and write priority queue blocks. Sorting and scanning all adjacency lists takes $O(\text{SORT}(K) + E/B)$ I/Os per round. Sorting $\mathcal{A}$ takes $O(V \cdot \text{SORT}(E))$ I/Os in total for all instances, as each edge is used at most twice per instance. We also perform $O(E)$ operations on each tournament tree using $O(VE\frac{L}{B} + V\frac{E}{B}\log\frac{V}{B})$ I/Os (Lemma 5). Finally, we use $O(\text{SORT}(E))$ I/Os per instance for computing σ_{sv} and $\delta_s(v)$, while computing the partial sums for $C_B(v)$ at no extra cost. In total we use

$$\begin{aligned} & O\left(\frac{V^2}{L} + \frac{V^2}{K} \cdot \left(\text{SORT}(K) + \frac{E}{B}\right)\right. \\ + \quad & \left. V \cdot \text{SORT}(E) + VE \cdot \frac{L}{B} + V \cdot \frac{E}{B}\log\frac{V}{B}\right) \\ = \quad & O\left(\frac{V^2}{L} + \frac{VEL}{B} + \frac{V^2E}{KB} + \frac{VE}{B}\log\frac{V}{B}\right) \end{aligned}$$

I/Os, where $1 \leq L \leq B$, and $L \leq K$. As we need to maintain $2 \cdot K$ tournament trees, we use $O(K \cdot V + E)$ space.

We balance the number of I/Os in the first two terms in the I/O bound above by setting $L = \sqrt{VB/E}$. We can now adjust K to trade space for I/Os. To optimize for I/Os, we set $K = \sqrt{VE/B} \geq L$, in which our algorithm uses

$$O\left(V \cdot \left(\sqrt{\frac{VE}{B}} + \frac{E}{B}\log\frac{V}{B}\right)\right)$$

I/Os and $O(V \cdot \sqrt{VE/B})$ space.

Theorem 3: Betweenness centrality for an undirected graph $G = (V, E)$ with positive edge weights can be computed in $O(V \cdot (\sqrt{VE/B} + \frac{E}{B}\log\frac{V}{B}))$ I/Os using $O(V \cdot \sqrt{VE/B})$ space, given that $E < VB/\log V$.

Remarks:

(i) For graphs with $E = O(V)$ the I/O improvement over the algorithm of Section IV-A is a factor of $\sqrt{B}$.

(ii) By choosing a value for $K < \sqrt{VE/B}$, one can trade I/Os for space.

(iii) The algorithm can also compute the weighted diameter, improving the space of the best known such algorithm from $O(V^2)$ to $O(V \cdot \sqrt{VE/B})$.

C. The Algorithm for Low-Diameter Weighted Graphs

In this section we describe an improved algorithm for weighted graphs with small (unweighted) diameter. Specifically, given a graph G with unweighted diameter $\text{diam}(G)$,

our algorithm runs in $O(VE \cdot \mathrm{diam}(G)/B)$ I/Os and uses $O(V^2)$ space. The basic idea is to simulate a distributed variant of the Bellman-Ford algorithm (as used in distance-vector routing algorithms), where each vertex maintains a complete set of distance estimates to all other nodes. The algorithm repeatedly updates these distance estimates of the vertices by comparing them with those of neighbouring vertices. After all distances are found, we compute the shortest-path counts, dependency values, and betweenness centrality for all vertices. A key feature of our algorithm is that it avoids the repeated sorting and permutation steps required for a naive simulation. Below we first describe the basic distributed Bellman-Ford algorithm for computing all-pairs shortest paths. Then we describe our I/O-efficient variant of this algorithm and the extensions necessary for computing the shortest-path counts, dependency values and finally betweenness centrality for all vertices.

Distributed Bellman-Ford: Consider running the distributed Bellman-Ford algorithm on a physical network consisting of a node for each vertex of G and a connection between node u and node v in case there is an edge $(u, v) \in E$. Each node v maintains a *distance vector* containing values $d_{\mathrm{t}}(v, w) < d(v, w)$ for all $w \in V$. Initially, $d_{\mathrm{t}}(u, v) = 0$ in case $u = v$, and $d_{\mathrm{t}}(u, v) = \infty$ otherwise. Then, the algorithm runs at most $\mathrm{diam}(G) + 1$ rounds in which the distance vector of each node v is sent to each of its neighbours in the network , after which $d'_{\mathrm{t}}(s, v) = \min_{u \in N(v)}\{d_{\mathrm{t}}(s, u) + w(u, v)\}$ for all $s \in V$ is computed, where $N(v)$ is the set of neighbours of v, and then sets $d_{\mathrm{t}}(s, v) = d'_{\mathrm{t}}(s, v)$ for all $s \in V$. Since any shortest path contains at most $\mathrm{diam}(G)$ edges, we are sure that $d_{\mathrm{t}}(s, t) = d(s, t)$ for all $s, t \in V$ after $\mathrm{diam}(G)$ rounds. Since no distances change in the next round, we can detect when to finish without knowing $\mathrm{diam}(G)$.

I/O-efficient shortest paths: We maintain the distance vector $(d_{\mathrm{t}}(v, v_1), \ldots, d_{\mathrm{t}}(v, v_{|V|}))$ for each vertex v as an ordered list in external memory. To simulate a round, instead of making all updates for one *vertex* at a time, which would require copying and sorting the distances, we make the updates involving one *edge* at a time. In more detail, for each round we first set $d'_{\mathrm{t}}(u, v) = \infty$ for all $u, v \in V$. Then, we scan the list of edges once, while for each edge $e = (u, v)$ scanning the distance vectors of u and v simultaneously. For each $1 \leq i \leq V$, we set $d'_{\mathrm{t}}(u, v_i) := \min(d'_{\mathrm{t}}(u, v_i), w(u, v) + d_{\mathrm{t}}(v, v_i))$ and $d'_{\mathrm{t}}(v, v_i) := \min(d'_{\mathrm{t}}(v, v_i), w(u, v) + d_{\mathrm{t}}(u, v_i))$. After finishing each round we set $d_{\mathrm{t}}(s, t) := d'_{\mathrm{t}}(s, t)$ for all $s, t \in V$. We stop after the first round in which none of the distance estimates change.

Shortest-path counts: Next we compute σ_{st} for all $s, t \in V$. Again, we run $\mathrm{diam}(G)$ rounds, going through all edges once per round. For each $s \in V$, we store estimates of the number of shortest paths from s to all other vertices in a *path-count vector* (list) $(\sigma_{sv_1}, \ldots, \sigma_{sv_{|V|}})$. We maintain for each estimate σ_{st} whether it is (currently known to be) *final*. In each round we finalize the path counts σ_{st} for which all predecessors of t with respect to s had final path counts in the last round. In order to efficiently find out which path counts need to be finalized in a round, we also maintain for each σ_{st} whether it is known to be *incomputable* in the current round (because some predecessor has no final path count yet).

Initially we only set $\sigma_{ss} := 1$ for $s \in V$, and these are final. At the beginning of each round we reset each non-final σ_{st} to 0 and record that it is not incomputable. When processing an edge (u, v), we scan simultaneously over both the distance and path-count vectors of u and v. We check for each v_i whether (u, v) is *tight* with respect to v_i, that is, whether $d(v, v_i) = d(u, v_i) + w(u, v)$. In case it is, and σ_{vv_i} is not final and σ_{uv_i} is final, we update $\sigma_{vv_i} := \sigma_{vv_i} + \sigma_{uv_i}$. In case σ_{uv_i} is not final we record that σ_{vv_i} is incomputable. If after completing a round we have not found a certain σ_{st} to be incomputable, we are in fact sure its current value is final and record this.

Dependency values: To compute $\delta_u(v)$ for all $u, v \in V$, we again proceed in $\mathrm{diam}(G)$ rounds, but instead of checking for tight edges to predecessors, we check for tight edges to successors with final dependency values. The dependency values for each vertex $v \in V$ are stored as a *dependency vector* (list) $(\delta_{v_1}(v), \ldots, \delta_{v_{|V|}}(v))$; initially $\delta_{v_i}(v) = 0$ for all i. When processing an edge (u, v), we scan over the dependency vectors of u and v as well as their path-count and distance vectors, and update the dependency values of all v_i by setting $\delta_{v_i}(u) := \delta_{v_i}(u) + \sigma_{uv_i}/\sigma_{vv_i} \cdot (1 + \delta_{v_i}(v))$ in case (u, v) is tight with respect to v_i, $\delta_{v_i}(u)$ is not final, and $\delta_{v_i}(v)$ is final.

Finally, we compute $C_B(v)$ for all $v \in V$ by summing all values in the dependency vector of v.

I/O complexity: For each of the three steps in the algorithm we scan the set of edges once per round, taking $O(\frac{E}{B})$ I/Os, and for each edge we scan at most six vectors simultaneously (the distance, path-count, and dependency vectors of both endpoints), taking $O(\frac{V}{B})$ I/Os per edge. It takes $O(\frac{V^2}{B})$ I/Os per round to finalize and reset the distance, path-count, and dependency values. Hence, in total we use $O(\mathrm{diam}(G) \cdot (\frac{E}{B} + E \cdot \frac{V}{B} + \frac{V^2}{B})) = O(VE \cdot \mathrm{diam}(G)/B)$ I/Os. The algorithm uses $O(V^2)$ space because it needs to store the three lists of length V for each vertex.

Theorem 4: Betweenness centrality for a directed graph $G = (V, E)$ with diameter $\mathrm{diam}(G)$ and positive edge weights can be computed in $O(VE \cdot \mathrm{diam}(G)/B)$ I/Os using $O(V^2)$ space.

Remarks:

(i) The algorithm uses a factor $O(B/\mathrm{diam}(G))$ fewer I/Os than the algorithm of Section IV-A in case of graphs where $E = O(V)$. For graphs where $E \geq VB/\log V$, the improvement is only a factor $O(\log \frac{E}{B}/\mathrm{diam}(G))$.

(ii) As opposed to the other algorithms discussed in this paper, the current algorithm can easily be made to work for directed graphs.
(iii) The algorithm is cache-oblivious, since it only scans over lists.

V. Conclusion

In this paper, we have provided a theoretical design and analysis for efficiently computing betweenness centrality in the external memory model, so as to provide an effective method for solving this problem in big data graph applications that are too large to fit in the main memory of a standard computer. Some interesting directions for future work would include an experimental validation for this work, as well as explorations of other external-memory algorithms for big data problems relating to social network analysis.

References

[1] M. Barthélemy, "Betweenness centrality in large complex networks," *The European Physical Journal B - Condensed Matter and Complex Systems*, vol. 38, pp. 163–168, 2004.

[2] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, "Complex networks: Structure and dynamics," *Physics Reports*, vol. 424, no. 4–5, pp. 175–308, 2006.

[3] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[4] ——, "On variants of shortest-path betweenness centrality and their generic computation," *Social Networks*, vol. 30, no. 2, pp. 136–145, 2008.

[5] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

[6] ——, "Centrality in social networks conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1978–1979.

[7] L. Leydesdorff, "Betweenness centrality as an indicator of the interdisciplinarity of scientific journals," *J. Am. Soc. Info. Science and Tech.*, vol. 58, no. 9, pp. 1303–1319, 2007.

[8] M. Newman, "Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality," *Physical Review E*, vol. 64, no. 1, p. 016132, 2001.

[9] D. Knoke and S. Yang, *Social Network Analysis*. Sage Publications, Inc, 2008.

[10] M. Beauchamp, "An improved index of centrality," *Behavioral Science*, vol. 10, no. 2, pp. 161–163, 1965.

[11] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[12] U. Brandes and D. Fleischer, "Centrality measures based on current flow," in *Symp. on Theoretical Aspects of Computer Science (STACS)*, ser. LNCS. Springer, 2005, vol. 3404, pp. 533–544.

[13] M. Newman, "A measure of betweenness centrality based on random walks," *Social networks*, vol. 27, no. 1, pp. 39–54, 2005.

[14] A. Aggarwal and J. S. Vitter, "The Input/Output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[15] J. S. Vitter, "External memory algorithms and data structures: dealing with massive data," *ACM Computing Surveys*, vol. 33, pp. 209–271, 2001.

[16] N. Zeh, "I/O-efficient graph algorithms," 2002, lecture notes from EEF Summer School on Massive Data Sets, Aarhus, Denmark.

[17] K. Munagala and A. Ranade, "I/O-complexity of graph algorithms," in *Proc. 10th Symp. on Discrete Algorithms*, 1999, pp. 687–694.

[18] K. Mehlhorn and U. Meyer, "External-memory breadth-first search with sublinear I/O," in *Proc. 10th European Symp. on Algorithms*, ser. LNCS, vol. 2461, 2002, pp. 723–735.

[19] L. Arge, U. Meyer, and L. Toma, "External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs," in *31st Int. Col. on Automata, Languages and Programming (ICALP)*, ser. LNCS, vol. 3142, 2004, pp. 57–74.

[20] R. A. Chowdhury and V. Ramachandran, "External-memory exact and approximate all-pairs shortest-paths in undirected graphs," in *Proc. 16th Symp. on Discrete Algorithms*, 2005, pp. 735–744.

[21] V. Kumar and E. J. Schwabe, "Improved algorithms and data structures for solving graph problems in external memory," in *Proc. 8th IEEE Symp. on Parallel and Distributed Processing*, 1996, pp. 169–176.

[22] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 4:1–4:22, 2012.

[23] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro, "Cache-oblivious priority queue and graph algorithm applications," in *Proc. 34th Symp. on Theory of Computation*, 2002, pp. 268–276.

[24] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[25] L. Arge, G. S. Brodal, and L. Toma, "On external-memory MST, SSSP, and multi-way planar graph separation," in *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, ser. LNCS, vol. 1851, 2000, pp. 709–715.

[26] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, "External-memory graph algorithms," in *Proc. 6th Symp. on Discrete Algorithms*, 1995, pp. 139–149.

[27] L. Arge, "The buffer tree: A technique for designing batched external data structures," *Algoritmica*, vol. 37, pp. 1–24, 2003.