Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs

Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs

Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis

Abstract—Betweenness centrality quantifies the importance of nodes in a graph in many applications, including network analysis, community detection and identification of influential users. Typically, graphs in such applications evolve over time. Thus, the computation of betweenness centrality should be performed incrementally. This is challenging because updating even a single edge may trigger the computation of all-pairs shortest paths in the entire graph. Existing approaches cannot scale to large graphs: they either require excessive memory (i.e., quadratic to the size of the input graph) or perform unnecessary computations rendering them prohibitively slow. We propose iCENTRAL; a novel incremental algorithm for computing betweenness centrality in evolving graphs. We decompose the graph into biconnected components and prove that processing can be localized within the affected components. iCENTRAL is the first algorithm to support incremental betweeness centrality computation *within* a graph component. This is done efficiently, in linear space; consequently, iCENTRAL scales to large graphs. We demonstrate with real datasets that the serial implementation of iCENTRAL is up to 3.7 times faster than existing serial methods. Our parallel implementation that scales to large graphs, is an order of magnitude faster than the state-of-the-art parallel algorithm, while using an order of magnitude less computational resources.

Index Terms—Betweenness centrality, dynamic graph algorithms, parallel graph algorithms

1 INTRODUCTION

GRAPHS model complex relationships among objects in a plethora of applications ranging from chemistry and bioinformatics to social networks and web analysis. An essential tool for graph analysis is centrality metrics defined on graph nodes. These metrics rank nodes according to their position in the graph and are typically interpreted as the prominence of the corresponding entities [1]. For instance, a node with high centrality in a citation graph may represent an influential paper or a prominent researcher. Many centrality metrics are defined using the number of paths that link pairs of nodes, or the ratio of the shortest paths a node lies on.

In this paper, we study the *betweenness centrality* metric [2], [3]; which for a node v of a graph G is defined as the fraction of the shortest paths between all pairs of nodes that pass through v. Betweenness centrality is used in a variety of applications, including community detection in social networks [4], identifying gene-disease associations in gene-interaction graphs [5] and message routing in mobile networks [6]. Computing betweenness centrality is an intensive task that requires the computation of all-pairs shortest paths. The fastest known algorithm was proposed by Brandes [1] and takes $\mathcal{O}(|V||E|)$ time, where $|V|$ is the number of nodes and $|E|$ is the number of edges of the input graph.

Modern graphs are inherently evolving [7]. For instance, users join and leave social networks over time, and connections between users (e.g., friendships) are established or demolished. Consider the relatively small twitter-munmun dataset with 460K nodes and 833K edges (see Section 4 for details) and assume one new edge is inserted. To recompute betweenness centrality from scratch, Brandes algorithm requires a day. Consequently, in evolving graphs, betweenness centrality should be updated in an incremental way that avoids complete recomputation. This is challenging because it requires all-pairs shortest path information in the entire graph. Even a small graph update can result in many shortest path alternations, causing many betweenness centrality values updates.

Algorithm	Time (sec)	Space (GB)	Cores
Green [8]	crashed	4,000.0	1
QUBE [9]	4,210	0.2	1
Lee 2016[1] [10]	2,634	0.06	1
Kourtellis [11]	2,376	4,000.0	100
iCENTRAL	**190**	**1.6**	**20**

TABLE 1: Performance of the best incremental algorithms for updating betweenness centrality after inserting one edge; twitter-munmun dataset (460K nodes, 833K edges).

In this work we concentrate on the incremental computation of betweenness centrality. In the related literature, three approaches stand out for updating betweenness centrality in evolving graphs:

1) Green's algorithm [8] stores and maintains the all-pairs shortest path information of the graph in the main memory. Although this approach is fast, memory requirements are prohibitive. For example, for the rel-

- *Fuad Jamour and Panos Kalnis are with the King Abdullah University of Science and Technology (KAUST), Saudi Arabia. E-mails: fuad.jamour@kaust.edu.sa and panos.kalnis@kaust.edu.sa*
- *Spiros Skiadopoulos is with the University of the Peloponnese, Greece. E-mail: spiros@uop.gr*

1. Estimated runtime. (see Section 4 for details)

atively small twitter-munmun graph mentioned above, Green requires more than 4TB of RAM, causing the algorithm to crash on our machine with 256GB of RAM, as shown in Table 1.

2) QUBE [9] decomposes the input graph into independent components using minimum union cycles, such that a change in one component does not affect the betweenness centrality values of nodes in other components. After a graph update, complete recomputation is only performed within the affected component. QUBE has linear space requirements and needs only 0.2GB of RAM for the twitter-munmun graph, but it is very slow (see Table 1). Recently, the same authors proposed using biconnected components (a graph decomposition that results in smaller components compared to the minimum union cycles) [10]. This approach is faster than the original QUBE (refer to Lee 2016 in Table 1), but it is still slow, since it requires complete recomputation within each affected biconnected component.
3) Kourtellis' algorithm [11] is a parallel map-reduce variation of Green's algorithm that uses the Hadoop file system, instead of RAM, to store the all-pairs shortest path information. As shown in Table 1, Kourtellis needs 4TB of distributed disk space for the twitter-munmun graph, and finishes roughly 2 times faster than QUBE, but uses 100 cores whereas QUBE needs only one.

In this paper we propose iCENTRAL: an incremental algorithm for computing betweenness centrality in evolving graphs, that needs space linear to the size of the graph allowing it to scale to large graphs. The main novelty of iCENTRAL is the incremental computation *within* each graph component, allowing it to avoid many unnecessary recomputations and be much faster than the state-of-the-art; for the example in Table 1, iCENTRAL on 20 cores is 10x faster than Kourtellis on 100 cores. iCENTRAL is based on three key ideas:

Avoid BFSs. Betweenness centrality needs the all-pairs shortest path information, which entails the computation of breadth-first search DAGs of all nodes in the graph [1]. We observe that after a graph update, many breadth-first search DAGs do not change at all. iCENTRAL implements a novel method that identifies efficiently the breadth-first search DAGs that remain intact *without* realizing the actual DAGs.

Incremental BFSs. For those DAGs that are affected by a graph update, iCENTRAL implements a novel algorithm that incrementally maintains only the parts of the breadth-first search DAGs that are affected *without* recomputing the actual DAGs.

Biconnected components. iCENTRAL decomposes the graph into biconnected components [12]. We formally prove (Theorem 1) that the betweenness centrality values of nodes not belonging to the affected biconnected components do not change after the update. We also show how to incrementally update betweenness centrality values *within* the affected components (Theorem 2). This contrasts existing approaches [9], [10] that need to recompute everything within each affected component.

The ideas above result in significant speedups over recomputation of betweenness centrality from scratch. Still, the computational intensity of betweenness centrality cannot be handled effectively by a single machine. Thus, exploiting parallelism is necessary to scale to large graphs. To this end, we develop a parallel version of iCENTRAL that runs on multiple cores on one machine, or on many machines. The main idea is to perform in parallel many breadth-first searches from different starting nodes. Other than some synchronization at the beginning and the end of the breadth-first searches, our implementation is embarrassingly parallel; therefore it scales well to large computer clusters. Parallel iCENTRAL can handle the largest graphs reported in previous work, using an order of magnitude less computational resources than the best existing parallel algorithm [11].

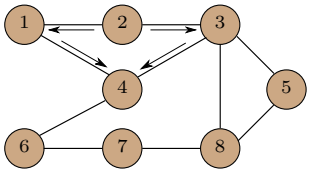

Fig. 1: Graph G contains two shortest paths from node 2 to node 4; node 1 lies on one of them, so σ_{24}=2 and $\sigma_{24}(1)$=1.

We make the following contributions in this paper:

- We propose iCENTRAL, a novel incremental method for updating betweenness centrality in evolving graphs. iCENTRAL scales to large graphs by requiring space linear to the graph size.
- We couple iCENTRAL with biconnected components graph decomposition and prove formally the correctness of our algorithm. iCENTRAL is the first algorithm to support incremental betweenness centrality computation within each graph component.
- We evaluate our method experimentally by using large real graphs. We show that the serial version of iCENTRAL is up to 3.7 times faster than the best serial methods.
- Finally, we develop a scalable parallel implementation of our algorithm. Compared to the state-of-the-art parallel incremental algorithm, parallel iCENTRAL is an order of magnitude faster and requires an order of magnitude fewer machines.

The rest of this paper is organized as follows: Section 2 contains essential background on the incremental computation of betweenness centrality. Section 3 introduces iCENTRAL. Section 4 presents the experimental analysis, followed by related work in Section 5 and conclusions in Section 6.

2 Background and Definitions

A graph $G = (V, E)$ is composed of a set of nodes V and a set of edges $E \subseteq V \times V$. For simplicity, we assume that G is unweighted, undirected and connected. Our results can be generalized to directed and weighted graphs.

2.1 Betweenness centrality

The *betweenness centrality* [1] value $BC_G[v]$ of node v in graph G is the fraction of the shortest paths between all

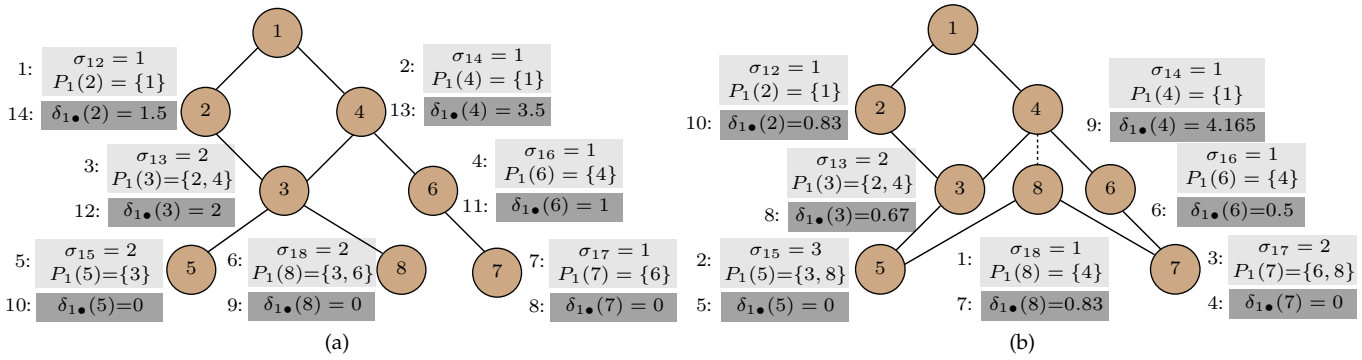

Fig. 2: Computing $\delta_{1\bullet}(v)$ (illustrations of Examples 1 and 3). The breadth-first search DAGs of node 1 in the example graph of Fig. 1 before and after inserting edge (4, 8) are shown in (a) and (b) respectively. For clarity, we do not illustrate edges connecting nodes at the same level.

pairs of nodes in the graph that pass through v. Formally:

$$BC_G[v] = \sum_{s,t \in V,\ s \neq t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

where σ_{st} is the number of shortest paths from node s to node t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through v. For example in Fig. 1, σ_{24}=2, $\sigma_{24}(1)$=1, and $BC_G[1]$=$\frac{\sigma_{24}(1)}{\sigma_{24}} + \frac{\sigma_{26}(1)}{\sigma_{26}}$=$\frac{1}{2} + \frac{1}{2}$=1.

The fastest known algorithm for computing betweenness centrality of the nodes of a static graph was proposed by Brandes [1]. For any triplet of nodes s, t, and v in V, Brandes defines pair dependency, denoted by $\delta_{st}(v)$, and source dependency, denoted by $\delta_{s\bullet}(v)$ as:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \quad \text{and} \quad \delta_{s\bullet}(v) = \sum_{t \in V,\ t \neq v} \delta_{st}(v) \tag{2}$$

respectively. Using the above notions, Equation 1 can be equivalently written as:

$$BC_G[v] = \sum_{s \in V,\ s \neq v} \delta_{s\bullet}(v) \tag{3}$$

Brandes proved that $\delta_{s\bullet}(v)$ can be computed using the following formula:

$$\delta_{s\bullet}(v) = \sum_{v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)) \tag{4}$$

where σ_{ij} is the number of shortest paths from i to j and $P_s(w)$ is the list of parents of w in the breadth-first search DAG of s. Equation 4 defines the source dependency of a source s on a node v as a function of the source dependencies of v's children, however, it does not define the process that needs to be performed to compute the source dependencies. To implement Equation 4, Brandes performs a two-phase process. The first phase executes a breadth-first search initiated from s to compute σ_{st} and $P_s(t)$ for all nodes $t \in V$ with $s \neq t$. The second phase performs a reverse breadth-first search (i.e., from the leaves to the root) and uses Equation 4 to compute $\delta_{s\bullet}(v)$.

The process is highlighted in the following example:

Example 1. *Consider graph G of Fig. 1. The computation of $\delta_{1\bullet}(v)$ for all $v \in V$ is illustrated in Fig. 2a. The first phase of Brandes algorithm performs breadth-first search and computes σ_{1v} and $P_1(v)$ (these results are illustrated in light gray boxes numbered according to the search order). For instance, in the 3rd step, σ_{13}=2 and $P_1(3)$=$\{2,4\}$. The second phase performs reverse breadth-first search and uses σ_{1v}, $P_1(v)$ and Equation 4 to compute $\delta_{1\bullet}(v)$ (these results are illustrated in dark gray boxes numbered according to the search order). For instance, in the 12th step, $\delta_{1\bullet}(3)$=$\frac{\sigma_{13}}{\sigma_{15}}(1+\delta_{1\bullet}(5)) + \frac{\sigma_{13}}{\sigma_{18}}(1+\delta_{1\bullet}(8))$=2.*

In Brandes method, the cost of computing the betweenness centrality of a single node does not differ much from the cost of computing betweenness centrality of *all* nodes of the graph. This is because $BC[v]$ is a function of the source dependencies $\delta_{s\bullet}(v)$ of all other nodes $s \in V, s \neq v$ (see Equation 3), and computing each $\delta_{s\bullet}(v)$ requires computing $\delta_{s\bullet}(v_{dec})$ for all nodes v_{dec} that are descendants of v in the breadth-first search DAG of s (see Equation 4). This means that computing $BC[v]$ involves computing the source dependencies on v and on other nodes as well. For instance, in Fig. 2a, to compute $\delta_{1\bullet}(4)$, the values of $\delta_{1\bullet}(i)$ for all $i \in \{3,5,6,7,8\}$ are required. With minor effort we may also compute all other values $\delta_{s\bullet}(i)$, $i \in V$ (i.e., $\delta_{1\bullet}(2)$ in the previous example). Using these values, we may compute the betweenness centrality of all other nodes.

2.2 Updating betweenness centrality

Graphs are rarely static; edges and nodes are inserted and deleted as a graph evolves. The most important update operation is the insertion of a new edge between two existing nodes; deletions can be performed analogously.

Consider graph $G(V,E)$ and the betweenness centrality values $BC_G[v]$ for all of its nodes. Assume that a new edge e is inserted into G. The new graph is denoted by $G'(V',E')$ where V'=V and E'=$E \cup \{e\}$. As explained in Section 2.1, computing $BC_G[v]$ is done by computing of $\delta_{s\bullet}(v)$ for all $s, v \in V$ (Equation 3). For each $s \in V$, the breadth-first search DAG of s is used to compute σ_{sv}, $P_s(v)$ and finally $\delta_{s\bullet}(v)$ for all $v \in V$ (see also Example 1). Adding edge e to G affects some of these DAGs and the corresponding values of $\delta_{s\bullet}(v)$. In our example, consider the breadth-first search DAG of node 1 (Fig. 2a) and add a new edge e between nodes 2 and 4. e does not change the DAG of node 1, so it does not

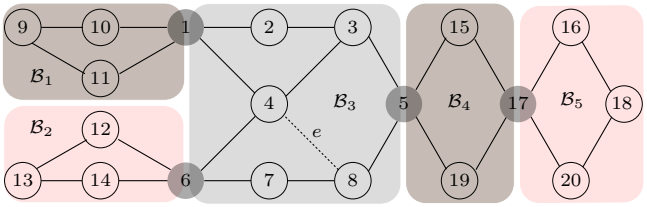

Fig. 3: Biconnected components of a graph

affect the values of σ_{1v}, $P_1(v)$ and $\delta_{1\bullet}(v)$. This observation is captured in the following proposition [8].

Proposition 1. *Let us add a new edge e to a graph G. If for a node s, edge e connects two nodes at the same level in its breath-first search DAG (or equivalently s has the same distance to both nodes of edge e) then all $\delta_{s\bullet}(v)$ values remain unchanged for all nodes v.*

This proposition allows updating betweenness centrality values after a graph update without recomputing all the breadth-first search DAGs in the graph. Previous works [8], [11] utilize Proposition 1 by *storing* the breadth-first search DAGs for all nodes in the graph, and querying these DAGs upon an edge insertion to identify the DAGs that need to be considered to update the betweenness centrality values. Such approach requires $\mathcal{O}(|V|^2)$ storage, which is prohibitive for large graphs. iCENTRAL scales to large graphs by removing this storage requirement.

2.3 Biconnected components

A *biconnected graph* is a graph that cannot be disconnected by removing any node [12]. *Biconnected component* of a graph is a maximal biconnected subgraph. An *articulation point* is a node whose removal disconnects the graph. A node can belong to multiple biconnected components while an edge belongs to only one biconnected component. Intuitively, a biconnected component is a maximal subgraph separated from the rest of the graph by articulation points. A graph can be decomposed into multiple biconnected components; two such components may share only one articulation point. In Fig. 3, we present a graph with 5 biconnected components $\mathcal{B}_1, \ldots, \mathcal{B}_5$ highlighted with different colors. Articulation points 1, 6, 5 and 17 connect the biconnected components and are illustrated in dark gray. The following example illustrates our notation:

Example 2. *Let G (respectively, G') be the graph of Fig. 3 without (respectively, with) the dotted edge e. The biconnected component affected by the insertion of edge e is denoted by $\mathcal{B}'_e$. In this case, $\mathcal{B}'_e$ is $\mathcal{B}_3$ and has 3 articulation points, namely 1, 5 and 6, that connect $\mathcal{B}'_e$ with the rest of the graph. We refer to the subgraphs connected to the affected biconnected component $\mathcal{B}'_e$ by $G_1, G_2, \ldots$ In Fig. 3 there are three such subgraphs composed of nodes 9 – 11, 15 – 20 and 12 – 14, respectively.*

Inserting an edge e in graph G may affect the partitioning. If the nodes of edge e belong to the same biconnected component then the partitioning remains intact. In the general case, edge e is between two distinct biconnected components $\mathcal{B}_i$ and $\mathcal{B}_j$. In this case, after the insertion of e all biconnected components on the path from $\mathcal{B}_i$ to $\mathcal{B}_j$ are merged to form a new biconnected component $\mathcal{B}'_e$.

Theorem 1. *Let G' be the graph constructed by adding edge e to graph G. Let also $\mathcal{B}'_e$ be the biconnected component of G' that edge e belongs to. $BC_{G'}[v]{=}BC_G[v]$ for all nodes v of graph G' that do not belong to $\mathcal{B}'_e$ (proof in the appendix).*

Theorem 1 states that the betweenness centrality values do not change outside biconnected component $\mathcal{B}'_e$; therefore, an incremental method does not need to update such values. Now let us consider the betweenness centrality values for nodes $v \in \mathcal{B}'_e$ that need to be updated. To compute $BC_{G'}[v]$, we need $\delta'_{st}(v)$ of all nodes s and t; some of these nodes are inside and others are outside of $\mathcal{B}'_e$. However, we prove in Lemma 1 (proof in the appendix) that $\delta'_{st}(v)$ can be computed by only considering nodes in $\mathcal{B}'_e$.

Lemma 1. *Let G' be the graph constructed by adding edge e to graph G. Let $\mathcal{B}'_e$ be the biconnected component of G' containing edge e. For all nodes v of G' that belong to $\mathcal{B}'_e$ and all pairs of nodes s and t of G', the pair dependency $\delta'_{st}(v)$ of G' either does not change or can be computed within biconnected component $\mathcal{B}'_e$.*

Previous works used biconnected components [10], as well as the minimum union cycles[2] decomposition [9] in conjunction with Theorem 1 and Lemma 1 to update betweenness centrality values. However, they recompute everything inside $\mathcal{B}'_e$. In contrast, iCENTRAL performs *incremental* computation *within* the affected component.

3 iCENTRAL

In this section we present our approach, iCENTRAL, which incrementally updates betweeness centrality values in a graph after an edge insertion or deletion. There are two main novelties: (*i*) In contrast to existing methods [8], [11] that need $\mathcal{O}(|V|^2)$ space, iCENTRAL needs linear space; and (*ii*) we prove theoretically (Theorem 2) that iCENTRAL performs incremental updates within the affected biconnected component, whereas existing work [10] must recompute everything from scratch. In what follows we focus on edge insertions; in Section 3.5 we discuss edge deletions.

3.1 Incremental computation

Let $\delta_{s\bullet}(v)$ (respectively, $\delta'_{s\bullet}(v)$) be the node dependencies of the original graph G (respectively, of the updated graph G'). From Equation 3, we have:

$$BC_G[v] = \sum_{s\in V,\ s\neq v} \delta_{s\bullet}(v) \text{ and } BC_{G'}[v] = \sum_{s\in V,\ s\neq v} \delta'_{s\bullet}(v) \quad (4)$$

Let $Q{\subseteq}V$ be the set of all nodes for which the values of $\delta_{s\bullet}(v)$ change with the insertion of edge e (which means that the node dependencies for all nodes in $V-Q$ remain intact). $BC_G[v]$ and $BC_{G'}[v]$ in Equation 4 can be combined:

$$BC_{G'}[v] = BC_G[v] - \sum_{s\in Q,\ s\neq v} \delta_{s\bullet}(v) + \sum_{s\in Q,\ s\neq v} \delta'_{s\bullet}(v) \quad (5)$$

Equation 5 provides an incremental way to update betweenness centrality. To compute $\delta_{s\bullet}(v)$ and $\delta'_{s\bullet}(v)$, we perform breadth-first and reverse breadth-first traversals,

2. Minimum union cycles provide similar guaranties to biconnected components, but result in an inferior decomposition with larger components; refer to Lemma A.3 in the appendix.

respectively, on the breadth-first search DAG of s. Specifically, to identify set Q, we need the distances between all nodes of the graph and the nodes of the inserted edge e. Let $e = (v_1, v_2)$. A node s belongs to Q if $d_{sv_1} \neq d_{sv_2}$; since G is undirected, this is equivalent to $d_{v_1 s} \neq d_{v_2 s}$. Thus, instead of storing the breadth-first search DAGs, we can perform two breadth-first traversals from v_1 and from v_2 to determine Q.

Example 3. *Consider graph G of Fig. 1 (the computation of $\delta_{1\bullet}(v)$ for all $v \in V$ is discussed in Example 1 and is illustrated in Fig. 2a). Adding a new edge e, illustrated with a dotted line, that connects nodes 4 and 8 modifies the breadth-first search DAG of node 1 and the values of $\delta_{s\bullet}(v)$ as presented in Fig. 2b.*

Until here we discussed the main block of iCENTRAL, which is the incremental computation of betweenness centrality values. This is the part of iCENTRAL that utilizes the fact that not all shortest paths change after a graph update and allows avoiding many breadth-first traversals. It is worth noting that the incremental computation method described in this section is generic and can be applied independently of the graph decomposition. That is, it can be applied either to the input graph directly; or with the biconnected components decomposition; or with any other suitable decomposition.

3.2 Using biconnected components

In Theorem 2 we combine the incremental computation idea with Lemma 1 to show how iCENTRAL incrementally computes the affected betweenness centrality values within the affected biconnected component.

Theorem 2. *Let G' be the graph constructed by adding edge e to graph G, $\mathcal{B}'_e$ be the biconnected component of G' that edge e belongs to, $a_1, \ldots, a_k$ be the articulation points of $\mathcal{B}'_e$ and $G_1, \ldots, G_k$ be the subgraphs of G' connected to $\mathcal{B}'_e$ through $a_1, \ldots, a_k$ respectively. For all nodes v of graph G' that belong to $\mathcal{B}'_e$ we have:*

$$BC_{G'}[v] = BC_G[v] - A[v] + A'[v] - B[v] + B'[v] - C[v] + C'[v] \quad (6)$$

where:

$$A[v] = \sum_{\substack{s,t \in \mathcal{B}'_e \\ s \neq t \neq v}} \delta_{st}(v), A'[v] = \sum_{\substack{s,t \in \mathcal{B}'_e \\ s \neq t \neq v}} \delta'_{st}(v),$$

$$B[v] = \sum_{\substack{s \in G_i, t \in \mathcal{B}'_e \\ s \neq t \neq v, \\ i=1 \cdots k}} \delta_{st}(v), B'[v] = \sum_{\substack{s \in G_i, t \in \mathcal{B}'_e \\ s \neq t \neq v, \\ i=1 \cdots k}} \delta'_{st}(v),$$

$$C[v] = \sum_{\substack{s \in G_i, t \in G_j \\ s \neq t \neq v,\ i=1 \cdots k, \\ j=1 \cdots k,\ i \neq j}} \delta_{st}(v), C'[v] = \sum_{\substack{s \in G_i, t \in G_j \\ s \neq t \neq v,\ i=1 \cdots k, \\ j=1 \cdots k,\ i \neq j}} \delta'_{st}(v)$$

Note that:

Expression $A[v]$ **(resp.** $A'[v]$**)** *is the contribution of nodes s and t, that both belong to the affected biconnected component $\mathcal{B}'_e$, to the betweenness centrality of v in graph G (resp. G'). Both expressions can be computed by considering only biconnected component $\mathcal{B}'_e$.*

Expression $B[v]$ **(resp.** $B'[v]$**)** *is the contribution of node s that does not belong, and node t that belongs to the affected biconnected component $\mathcal{B}'_e$, to the betweenness centrality of v and can be computed by:*

$$B[v] = \sum_{\substack{i=1 \cdots k \\ t \in \mathcal{B}'_e, t \neq v}} |V_{G_i}| \cdot \delta_{a_i t}(v), B'[v] = \sum_{\substack{i=1 \cdots k \\ t \in \mathcal{B}'_e, t \neq v}} |V_{G_i}| \cdot \delta'_{a_i t}(v)$$

where $\delta_{a_i t}(v)$, $\delta'_{a_i t}(v)$ can be computed by considering only biconnected component $\mathcal{B}'_e$.

Expression $C[v]$ **(resp.** $C'[v]$**)** *is the contribution of nodes s and t, that both do not belong to the affected biconnected component $\mathcal{B}'_e$, to the betweenness centrality of v and can be computed by:*

$$C[v] = \sum_{\substack{i=1 \cdots k, \\ j=1 \cdots k, \\ i \neq j}} |V_{G_i}| \cdot |V_{G_j}| \cdot \delta_{a_i a_j}(v), C'[v] = \sum_{\substack{i=1 \cdots k, \\ j=1 \cdots k, \\ i \neq j}} |V_{G_i}| \cdot |V_{G_j}| \cdot \delta'_{a_i a_j}(v)$$

where $\delta_{a_i a_j}(v)$ and $\delta'_{a_i a_j}(v)$ can be computed by considering only biconnected component $\mathcal{B}'_e$ (proof in the appendix).

Theorem 2 is crucial. It states that the affected betweenness centrality values $BC_{G'}[v]$ can be *incrementally* computed after a graph update by considering the nodes within the affected biconnected component $\mathcal{B}'_e$. This theorem distinguishes iCENTRAL from existing work [10], which requires recomputation from scratch inside the affected component.

3.3 Computing source dependencies

In this subsection we show how to compute the source dependencies that encapsulate the pair dependencies in each of the expressions $A[v]$, $A'[v]$, $B[v]$, $B'[v]$, $C[v]$, and $C'[v]$ of Theorem 2 to enable updating the betweenness centrality with breadth-first and reverse breadth-first traversals. Below we show how the source dependencies are computed for $A[v]$, $B[v]$, and $C[v]$. The source dependency computation of $A'[v]$, $B'[v]$, and $C'[v]$ can be shown similarly.

Expression $A[v]$ **(**$s, t \in \mathcal{B}'_e$**).** This case is similar to computing the source dependencies in a graph. Thus, the source dependencies that encapsulate the pair dependencies in $A[v]$ are computed with Equation 4, (i.e., $\delta_{s\bullet}(v) = \sum_{v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))$).

Expression $B[v]$ **(**$s \notin \mathcal{B}'_e$**,** $t \in \mathcal{B}'_e$**).** Let $s \in G_i$. In this case, each pair dependency $\delta_{st}(v)$ is equal to $\delta_{a_i t}(v)$, (from the proof of Lemma 1). Consequently, the source dependency of each node $s \in G_i$ is equal to the source dependency of a_i (i.e., $\delta_{s\bullet}(v) = \delta_{a_i \bullet}(v)$). This means that the summation of the source dependencies of all nodes $s \in G_i$ can be computed at once with $|V_{G_i}| \cdot \delta_{a_i \bullet}(v)$, which encapsulates the pair dependencies in $B[v]$.

Expression $C[v]$ **(**$s, t \notin \mathcal{B}'_e$**).** Let $s \in G_i$ and $t \in G_j$ where $i \neq j$. For this case, we define the *external graph pair dependency* and the *external graph dependency* as:

$$\delta_{G_i G_j}(v) = \sum_{s \in G_i, t \in G_j} \delta_{st}(v) \text{ and } \delta_{G_i \bullet}(v) = \sum_{j=1 \cdots k,\ i \neq j} \delta_{G_i G_j}(v).$$

The external graph dependencies encapsulate the pair dependencies in $C[v]$, and can be computed using the following equation: (derivation in the appendix)

$$\delta_{G_i \bullet}(v) = \begin{cases} |V_{G_i}| \cdot |V_{G_v}| & v \text{ is articulation point} \\ \sum_{\substack{v \in \\ P_{a_i}(w)}} \frac{\sigma_{a_i v}}{\sigma_{a_i w}} \cdot \delta_{G_i \bullet}(w) & \text{otherwise} \end{cases} \quad (7)$$

Algorithm: iCENTRAL

Input: Graph $G(V, E)$, the betweenness centrality values BC_G of G, and new edge $e \in V \times V$

Output: The betweenness centrality values $BC_{G'}$ of graph G' that is constructed by inserting edge e to graph G

1 Find the biconnected components of G'
2 Let $\mathcal{B}'_e(V_{\mathcal{B}'_e}, E_{\mathcal{B}'_e})$ be the biconnected component of G' that edge e belongs to
3 Let $\mathcal{B}_e(V_{\mathcal{B}_e}, E_{\mathcal{B}_e})$ be $\mathcal{B}'_e(V_{\mathcal{B}'_e}, E_{\mathcal{B}'_e} - \{e\})$
4 Let $e = (v_1, v_2)$
5 Perform a breadth-first search to compute the distance $d_{v_1 s}$ between v_1 and s in $\mathcal{B}_e$
6 Perform a breadth-first search to compute the distance $d_{v_2 s}$ between v_2 and s in $\mathcal{B}_e$
7 **for** *all nodes* $s \in V_{\mathcal{B}_e}$ **do**
8 **if** $d_{v_1 s} \neq d_{v_2 s}$ **then**
9 Add s to Q
10 **for** *all nodes* $s \in Q$ **do**
11 Find $\sigma_s[v]$ and $P_s[v]$ for $v \in V_{\mathcal{B}_e}$ (BFS from s)
12 $\delta_{s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}_e}$
13 $\delta_{G_s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}_e}$
14 **for** *all nodes* $w \in V_{\mathcal{B}_e}$ *in reverse BFS order from* s **do**
15 **if** s *and* w *are articulation points* **then**
16 $\delta_{G_s\bullet}[w] = |V_{G_s}| \cdot |V_{G_w}|$
17 **for** $p \in P_s[w]$ **do**
18 $\delta_{s\bullet}[p] = \delta_{s\bullet}[p] + \frac{\sigma_s[p]}{\sigma_s[w]} \cdot (1 + \delta_{s\bullet}[w])$
19 **if** s *is an articulation point* **then**
20 $\delta_{G_s\bullet}[p] = \delta_{G_s\bullet}[p] + \delta_{G_s\bullet}[w] \cdot \frac{\sigma_s[p]}{\sigma_s[w]}$
21 **if** $w \neq s$ **then**
22 $BC_{G'}[w] = BC_{G'}[w] - \delta_{s\bullet}[w]/2.0$
23 **if** s *is an articulation point* **then**
24 $BC_{G'}[w] = BC_{G'}[w] - \delta_{s\bullet}[w] \cdot |V_{G_s}|$
25 $BC_{G'}[w] = BC_{G'}[w] - \delta_{G_s\bullet}[w]/2.0$
26 Find $\sigma'_s[v]$ and $P'_s[v]$ for $v \in V_{\mathcal{B}'_e}$ (partial BFS from s)
27 $\delta'_{s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}'_e}$
28 $\delta'_{G_s\bullet}[v] = 0$ for $v \in V_{\mathcal{B}'_e}$
29 **for** *all nodes* $w \in V_{\mathcal{B}'_e}$ *in reverse BFS order from* s **do**
30 **if** s *and* w *are articulation points* **then**
31 $\delta'_{G_s\bullet}[w] = |V_{G_s}| \cdot |V_{G_w}|$
32 **for** $p \in P'_s[w]$ **do**
33 $\delta'_{s\bullet}[p] = \delta'_{s\bullet}[p] + \frac{\sigma'_s[p]}{\sigma'_s[w]} \cdot (1 + \delta'_{s\bullet}[w])$
34 **if** s *is an articulation point* **then**
35 $\delta'_{G_s\bullet}[p] = \delta'_{G_s\bullet}[p] + \delta'_{G_s\bullet}[w] \cdot \frac{\sigma'_s[p]}{\sigma'_s[w]}$
36 **if** $w \neq s$ **then**
37 $BC_{G'}[w] = BC_{G'}[w] + \delta'_{s\bullet}[w]/2.0$
38 **if** s *is an articulation point* **then**
39 $BC_{G'}[w] = BC_{G'}[w] + \delta'_{s\bullet}[w] \cdot |V_{G_s}|$
40 $BC_{G'}[w] = BC_{G'}[w] + \delta'_{G_s\bullet}[w]/2.0$
41 **return** $BC_{G'}$

3.4 iCENTRAL algorithm

Algorithm iCENTRAL works as follows: Line 1 decomposes the input graph G' into its biconnected components using Hopcroft and Tarjan algorithm [13]. Then, Line 2 identifies biconnected component $\mathcal{B}'_e$ that is affected by the graph update. iCENTRAL performs the biconnected components decomposition on G' instead of G to support the general case when the inserted edge e connects multiple biconnected components of G. Following, Lines 5–9 identify set Q by performing two breadth-first traversals in $\mathcal{B}_e$.

For all nodes in set Q, Lines 10 – 40 iteratively update the betweenness centrality of nodes in $\mathcal{B}'_e$ by performing a breadth-first and a reverse breadth-first search, respectively. The updates are done in two steps: subtracting the old source and external graph dependencies; i.e., $A[v]$, $B[v]$, and $C[v]$ (Lines 11 – 25), and adding the new source and external graph dependencies; i.e., $A'[v]$, $B'[v]$, and $C'[v]$ (Lines 26 – 40). In the next paragraph we explain how to use the equations in Section 3.3 to compute and subtract $A[v]$, $B[v]$, and $C[v]$ (Lines 11 – 25). Computing and adding $A'[v]$, $B'[v]$, and $C'[v]$ are done similarly in Lines 26 – 40.

Computing the source dependencies that encapsulate node pairs in $A[v]$ is done with a direct application of Equation 4 in Line 18. These source dependencies are subtracted from the betweenness centrality values in Line 22. The pair dependencies in $B[v]$ are encapsulated in $\delta_{s\bullet}(w) \cdot |V_{G_s}|$, which is subtracted in Line 24. This is done only if s is an articulation point as explained in Section 3.3. The case of pair dependencies in $C[v]$ is more involved, and requires maintaining the structure $\delta_{G_s\bullet}[v]$ to compute the external graph dependencies (Line 13). Computing $\delta_{G_s\bullet}[v]$ starts with the base case of Equation 7 when both the source s and the node considered in the reverse breadth-first search (i.e, w) are articulation points. This is done in Line 16. The external graph dependency on node w is computed with Equation 7 if the source s is an articulation point in Line 20. Note that $\delta_{G_s\bullet}(w)$ is defined only if s is an articulation point, because that is the case when external graph dependencies are defined and needed to be propagated to nodes in $\mathcal{B}'_e$ (see Section 3.3). Note that $\delta_{s\bullet}(w)$ and $\delta_{G_s\bullet}(w)$ are divided by 2 in Lines 22 and 25 to avoid counting the same path twice, since the input graph is undirected.

iCENTRAL utilizes the fact that many σ_{sv} values remain unchanged in the breadth-first search DAG of s after inserting edge e, even when $s \in Q$. Let l be the node of e further from s in its breadth-first search DAG. To find the nodes for which σ_{sv} changes, we only need to start a breadth-first search from l in the breadth-first DAG of s. We refer to this optimized traversal as *partial breadth-first search* (Line 26).

3.5 Edge deletions

Minimal modifications are required to support edge deletions. Specifically, Lines 1, 2, and 3, and are modified as follows. Line 1 becomes:

Find the biconnected components of G

The biconnected component decomposition is computed on G to support the case when the removal of e breaks an existing component into multiple components. The subgraph that needs to be considered is the union of the resulting smaller components, which is the biconnected component that e belongs to in G, denoted by $\mathcal{B}_e$ (as reflected in the updated Lines 2 and 3 below).

Let $\mathcal{B}_e(V_{\mathcal{B}_e}, E_{\mathcal{B}_e})$ be the biconnected component of G that edge e belongs to
Let $\mathcal{B}'_e(V_{\mathcal{B}'_e}, E_{\mathcal{B}'_e})$ be $\mathcal{B}_e(V_{\mathcal{B}_e}, E_{\mathcal{B}_e} \cup \{e\})$

3.6 Weighted and directed graphs

iCENTRAL can be generalized to weighted and directed graphs. To handle weighted graphs, Dijkstra's algorithm should be used instead of breadth-first traversal in Lines 11 and 26. To handle directed graphs, the main change is identifying set Q by performing breadth-first traversals on the transpose of $\mathcal{B}'_e$ rather than on $\mathcal{B}'_e$ itself.

3.7 Parallel implementation of iCENTRAL

Even though iCENTRAL saves a lot of computation, updating betweenness centrality in large graphs requires a many independent breadth-first traversals that can be done in parallel. Thus, in this subsection we describe two versions of parallel iCENTRAL; a shared-memory (multi-threaded) implementation and a distributed-memory implementation. Note that the iterations of iCENTRAL (Lines 10–40) are independent of each other. Thus, we are able to follow a simple coarse-grained approach and compute the contributions of subsets of Q in parallel.

Shared-memory implementation. The main thread computes the biconnected components decomposition and set Q (i.e, the set of nodes for which $\delta_{s\bullet}$ changes). Following, set Q is divided uniformly among the available threads and each thread computes the changes of $\delta_{s\bullet}$ for its assigned sources. To avoid excessive allocations and deallocations of intermediate structures, memory allocation is performed at the initialization of each thread. All iterations in a particular thread share the same intermediate structures. After all threads complete their computation, the main thread summarizes all partial contributions to produce the final updated betweenness centrality values. Note that the shared-memory implementation needs only one copy of the graph, and each thread maintains intermediate structures (i.e., shortest distances, shortest distance counts, and parent lists).

Distributed-memory implementation. This implementation adds another level of parallelism. The nodes in set Q are first divided uniformly among the available machines. The assigned subset of Q within a machine is further divided uniformly among the cores of the machine similarly to the shared-memory implementation. The final result accumulation is done in two steps. The partial results are accumulated at the master thread of each machine, and then the master machine collects the accumulated partial results from all other machines and summarizes them for each node to produce the final updated betweenness centrality values. A copy of the graph is stored in each of the machines to avoid communication among machines to retrieve the graph structure.

iCENTRAL uses a serial implementation of Hopcroft and Tarjan algorithm [13] to compute the biconnected components. This algorithm is fast (i.e., takes linear time), and it takes an insignificant portion of the total execution time of iCENTRAL. To give a concrete example, the serial version of iCENTRAL needs about 1800 seconds to compute the betweenness centrality updates for the twitter-munmun dataset after an edge insertion, out of which 1.3 seconds are spent to compute the biconnected components (i.e., less than 0.1% of the total execution time). Additionally, parallel biconnected components computation methods do not achieve more than 0.4 parallel speedup efficiency [14]. For the above two reasons, we do not consider parallel algorithms for finding biconnected components and focus on the optimization of the remaining tasks that are quadratic and dominate the execution time.

3.8 Complexity analysis of iCENTRAL

Let V and E be the sets of nodes and edges in the input graph G. Let $V_{\mathcal{B}'_e}$ and $E_{\mathcal{B}'_e}$ be the sets of nodes and edges of the biconnected component of the updated graph G' that the inserted edge e belongs to.

Initially, iCENTRAL computes the biconnected components (Line 1). This is done by depth-first traversal [13], requiring $\mathcal{O}(|V|+|E|)$ time and $\mathcal{O}(|V|+|E|)$ space. Then iCENTRAL computes set Q by doing two breadth-first traversals and one loop through the nodes of $\mathcal{B}'_e$ to check for sources that qualify for Q membership (Lines 5–9), which take $\mathcal{O}(|V_{\mathcal{B}'_e}|+|E_{\mathcal{B}'_e}|)$ time and space. Following, iCENTRAL considers all nodes s in Q (which is a subset of $V_{\mathcal{B}'_e}$). Each iteration (Lines 10 – 40) performs two breadth-first and two reverse breadth-first traversals and is performed in $\mathcal{O}(|V_{\mathcal{B}'_e}|+|E_{\mathcal{B}'_e}|)$ time and $\mathcal{O}(|V_{\mathcal{B}'_e}|+|E_{\mathcal{B}'_e}|)$ space. Thus, the complexity of Lines 10 – 40 is $\mathcal{O}(|Q||E_{\mathcal{B}'_e}|)$ time and $\mathcal{O}(|V_{\mathcal{B}'_e}|+|E_{\mathcal{B}'_e}|)$ space (the space bound does not change because the reserved space is freed after each iteration). To summarize, the overall complexity of iCENTRAL is $\mathcal{O}(|Q||E_{\mathcal{B}'_e}|)$ time and $\mathcal{O}(|V|+|E|)$ space.

With T processing units we may parallelize the execution of Lines 10 – 40. iCENTRAL can be performed in $\mathcal{O}(|Q||E_{\mathcal{B}'_e}|/T)$ time and requires $\mathcal{O}(T(|V|+|E|))$ space (since each unit needs independent storage). Since Q is bounded by V and $E_{\mathcal{B}'_e}$ is bounded by E, the time bound can be also expressed as $\mathcal{O}(|V||E|)$ for the serial and $\mathcal{O}(|V||E|/T)$ for the parallel case. In real graphs, though, we typically have $|Q|<|V|$, so these bounds are crude and the expected time is much lower. This is also verified in our experiments (Section 4).

4 EXPERIMENTAL EVALUATION

We experimentally evaluate iCENTRAL and compare it with several state-of-the-art methods [1], [8], [9], [10], [11]. Our results illustrate that iCENTRAL is more than an order of magnitude faster than non-incremental methods, and 3.7 times faster than the state-of-the-art serial incremental algorithm. We also demonstrate that the parallel version of iCENTRAL scales to graphs with millions of nodes and edges, is an order of magnitude faster, and uses an order of magnitude less resources than the state-of-the-art parallel incremental algorithm.

4.1 Experimental setup

Datasets. We use the synthetic and real graph datasets listed in Table 2. If a graph dataset is directed, we consider its undirected version and if a graph is not connected we consider its largest connected component. We refer to graphs with less than 100,000 nodes as *small*, and those with more nodes as *large*. Small graphs were used in [9], and large graphs were downloaded from KONECT[3] and SNAP[4]. We

3. http://konect.uni-koblenz.de/networks
4. http://snap.stanford.edu/data

	Graph dataset	$\|V_G\|$	$\|E_G\|$	Diam.
Synthetic	ER	10,000	100,000	5
	PA	10,000	99,945	4
	FF	10,000	151,433	14
Real datasets	Cagr	4,158	13,428	16
	Epa	4,253	8,897	10
	Eva	4,475	4,654	17
	Erdos972	5,440	8,940	11
	Erdos02	6,927	11,850	4
	Wiki-Vote	7,066	10,0736	6
	Contact	13,373	79,823	9
	slashdot	51,083	117,378	13
	facebook	63,392	816,831	12
	epinions	119,130	704,572	13
	email-EuAll	224,832	340,795	11
	com-dblp	317,080	1,049,866	20
	web-NotreDame	325,729	1,117,563	31
	twitter-munmun	465,017	833,540	8
	amazon	2,146,057	5,743,146	11

TABLE 2: Graph datasets used in the experiments.

keep the dataset names of the original sources to avoid confusion. The datasets cover a wide variety of properties and domains. Specifically:

- Synthetic graphs: ER, PA, and FF are synthetic graphs generated by SNAPPY[5] using the following random graph generation models, respectively: Erdős-Rényi [15], Preferential Attachment [16] (with node out-degree of 10), and Forest Fire [17] (with forward and backward probabilities of 0.35).
- Real small graphs: Cagr, Erdos02, and Erdos972 are collaboration networks; Epa is a web graph; Eva is a media ownership network; Wiki-Vote is vote network between users of Wikipedia; and Contact is a communication network. slashdot and facebook are snapshots of the Slashdot and the Facebook social networks; edges in these two networks are timestamped, and they represent real evolving graph data.
- Real large graphs: epinions is a trust network between Epinions users; email-EuAll is a communication network between an EU's institution researchers; com-dblp is a co-authorship network; web-NotreDame is the web graph of the University of Notre Dame; twitter-munmun is a snapshot of the Twitter follower network; and amazon is an Amazon product rating network.

Detailed information about these datasets is available in the respective sources. Note that the amazon dataset (2M nodes and 6M edges) is the largest dataset ever considered for incremental betweenness centrality. iCENTRAL scales to this dataset using a computer cluster with 19 machines, whereas the only existing work [11] that scales to it requires a cluster with hundreds of machines.

Graph updates. For insertions, we insert random edges to update a graph, except for the facebook and the slashdot datasets where we remove the most recent edges and insert them back. For deletions, we select random edges from the graph excluding bridge edges to maintain the connectedness of the graph. In our study, we evaluate the average runtime for all edge insertions or deletions.

5. http://snap.stanford.edu/snappy/

Competitors. We compare against the following algorithms:

KOURTELLIS: This is the original parallel map-reduce implementation[6] by Kourtellis et al. [11]. It is incremental and stores intermediate results in HDFS files. We execute it on Hadoop 1.2.1 with 100 mappers and one reducer. KOURTELLIS is a direct competitor to the parallel version of iCENTRAL.

GREEN: This is the original implementation of Green's algorithm [8] provided by its authors. GREEN is incremental and stores intermediate results in memory.

QUBE: This is our implementation QUBE [9]; the original implementation was not available to us. As in the original version [9], we use Brandes algorithm and the minimum union cycle decomposition. In our implementation, we replace the original minimum union cycles decomposition algorithm, which has a time complexity of $\mathcal{O}(|E|^3)$ with a linear time alternative [18]. QUBE is incremental and does not store intermediate results; thus, is a direct competitor to serial iCENTRAL.

BRANDES: This is our implementation of Brandes algorithm [1]. BRANDES is the best algorithm for static graphs and can be used as baseline for the incremental solutions. We also use the parallel implementation of Brandes[7] from [19] for the parallel Brandes experiment.

LEE-BCC: This is our implementation that mimics the improved QUBE algorithm [10]. Since we did not have access to the original implementation, we developed a simulated version that inserts an edge within the largest biconnected component of the input graph and runs Brandes in that component, but avoids all other overheads. Therefore, the reported runtimes consist a *lower-bound* of the actual LEE-BCC implementation.

Implementation. All algorithms are coded in C++. The parallel version of iCENTRAL[8] is implemented using MPI and C++11 threads. All serial and multi-threaded experiments are executed on a Linux machine with a 20-core Intel Xeon 2.80GHz CPU and 128GB main memory. The distributed implementations of iCENTRAL and KOURTELLIS are executed on a Linux cluster with 19 machines, each with 2x12-core AMD Opteron 2.10GHz CPUs and 140GB of main memory.

4.2 Comparison with state-of-the-art

In this subsection, we show that iCENTRAL can support real evolving graphs, and its real-time update capabilities outperform the state-of-the-art parallel algorithm. We also illustrate that iCENTRAL is able to handle graphs with millions of nodes and edges. Finally, we compare serial iCENTRAL with the existing state-of-the-art serial algorithms.

4.2.1 Parallel algorithms

We start by comparing iCENTRAL with KOURTELLIS the state-of-the-art parallel incremental algorithm. We use the

6. https://github.com/nicolaskourtellis/StreamingBetweenness
7. https://ecrc.github.io/BeBeCA/
8. iCENTRAL is available at: https://github.com/fjamour/icentral

	Insertions		Deletions	
Graph dataset	*i*CENTRAL	KOURTELLIS	*i*CENTRAL	KOURTELLIS
epinions	14.82	214.23	26.59	12.75
email-EuAll	4.15	80.89	6.01	140.39
com-dblp	151.63	1,027.25	108.65	314.70
web-NotreDame	42.06	322.68	30.19	272.87
twitter-munmun	35.01	2,376.28	66.73	2,522.56
amazon	13,567.79	N/A	14,803.24	N/A

TABLE 3: Average execution time (in sec) of *i*CENTRAL and KOURTELLIS for the large datasets on 19 machines. KOURTELLIS could not process the amazon dataset within 24h.

	Algorithm	$\mathcal{M}$	Missed edges	Avg. delay (sec)
facebook	*i*CENTRAL	1	17	6.47
	***i*CENTRAL**	**19**	**5**	**0.35**
	KOURTELLIS	19	23	117.86
slashdot	*i*CENTRAL	1	4	0.24
	***i*CENTRAL**	**19**	**4**	**0.09**
	KOURTELLIS	19	4	0.62

TABLE 4: Number of missed edge updates and average delays for different algorithms. $\mathcal{M}$ is the number of machines used.

facebook and the slashdot datasets in this evaluation, for which timestamped edges are available. Edge timestamps make it possible to compute edge inter-arrival times, which allows evaluating the real-time update capabilities of an incremental algorithm.

Let t_i (respectively t_{i+1}) be the arrival time of edge e_i (respectively of the following edge e_{i+1}) and let T_i be the time required for an algorithm to insert edge e_i. When $T_i \leq t_{i+1} - t_i$ the algorithm is able to successfully insert e_i before e_{i+1} arrives. On the contrary, when $t_{i+1} - t_i < T_i$ the algorithm is not able to process edge e_i. Thus, we say that the algorithm *missed* edge e_i. In such a case, we say that the algorithm showed a *delay* of $T_i - (t_{i+1} - t_i)$ time. Missed edges and delays are two valuable metrics for our comparison. An efficient algorithm should report a small number for both metrics.

In this experiment, we remove the 100 most recent edges from each of the facebook and the slashdot datasets and then insert them again in order. We measure the number of missed edges and the corresponding delay for multi-threaded *i*CENTRAL, distributed memory *i*CENTRAL, and KOURTELLIS. Our results are illustrated in Table 4. Since both methods are parallel, we also report the number of used machines (denoted by $\mathcal{M}$). Our results clearly indicate that *i*CENTRAL outperforms KOURTELLIS. Specifically, for the facebook dataset, KOURTELLIS parallel update algorithm requires 19 machines to achieve 23 missed edges with an average delay of 117.86 seconds (Table 4). To compare, *i*CENTRAL misses only 5 edges with nearly 300 times lower average delay (0.35 sec). Even on a single machine (our 20-core machine), multi-threaded *i*CENTRAL misses 17 edges with 18 times lower average delay than KOURTELLIS.

The slashdot dataset is smaller (has nearly 8 times fewer edges than facebook) and can be adequately handled by both algorithms *i*CENTRAL and KOURTELLIS (Table 4). The four edges that both algorithms cannot process correspond to the ones that arrive at exactly the same time with the previous edge (i.e., $t_{i+1} - t_i = 0$). Still, *i*CENTRAL achieves lower average delay even when a single machine is used (Table 4).

We also compare the execution times of *i*CENTRAL and KOURTELLIS on the large datasets (Table 3) for both edge insertions and edge deletions.

Insertions: *i*CENTRAL is, on average, 23 times faster than KOURTELLIS for all the large datasets and can be up to 67 times faster for the twitter-munmun dataset.

Deletions: *i*CENTRAL is, on average, 14 times faster than KOURTELLIS, and is up to 37 times faster for the twitter-munmun dataset. KOURTELLIS is faster than *i*CENTRAL only for deletions on the epinions dataset.

Note that *i*CENTRAL scales to the amazon dataset using a cluster of 19 machines, whereas KOURTELLIS does not finish in 24 hours on the same cluster. To handle the amazon dataset, KOURTELLIS needs a cluster with hundreds of machines, as reported in [11].

4.2.2 Serial algorithms

We compare the performance of *i*CENTRAL against the performance of serial methods, namely BRANDES, QUBE and GREEN, to highlight the algorithmic value of our approach. For fairness, we illustrate the performance of *i*CENTRAL in two different settings: The first restricts *i*CENTRAL to use one thread; thus, making its execution serial and directly comparable with the rest. Since *i*CENTRAL is parallel, for completeness, we also illustrate the performance of multi-threaded *i*CENTRAL on one machine. We compute the average runtime of inserting 100 edges for small and 10 edges for large graph datasets.

QUBE and BRANDES: We compare *i*CENTRAL against QUBE, which is incremental and scales to large graphs, and BRANDES, the state-of-the-art static algorithm. The results are shown in Table 5. For all datasets, *i*CENTRAL outperforms QUBE and BRANDES. Specifically, the serial version of *i*CENTRAL (i.e., 1 core) is up to 2.89 times faster than QUBE and up to 75 times faster than BRANDES, for the Erdos02 and email-EuAll datasets, respectively. On average, for all datasets *i*CENTRAL is 1.93 times faster than QUBE and more than 9 times faster than BRANDES. The results for the parallel version (20 cores) are more impressive: *i*CENTRAL is up to 32.46 times faster for the com-dblp dataset and more than 20 times faster on average than QUBE. Compared to parallel BRANDES, *i*CENTRAL is up to 111.20 times faster

	Graph dataset	*i*CENTRAL		BRANDES		QUBE
		20 cores	1 core	20 cores	1 core	
Small datasets	ER	1.31	18.54	1.55	20.77	27.53
	PA	1.11	16.56	1.80	16.62	23.23
	FF	0.79	9.94	1.55	22.41	22.83
	Cagr	0.09	0.87	0.12	1.43	1.55
	Epa	0.06	0.46	0.09	0.98	0.75
	Eva	0.02	0.02	0.07	0.58	0.04
	Erdos972	0.05	0.36	0.15	1.31	0.51
	Erdos02	0.05	0.27	0.21	2.12	0.78
	Wiki-Vote	0.44	5.59	0.63	9.13	9.11
	Contact	0.73	11.25	1.58	20.32	18.78
	slashdot	4.08	45.47	42.93	184.75	94.60
	facebook	77.44	914.30	175.84	995.07	1,435.37
Large	epinions	83.56	827.27	475.33	2,836.07	1,514.37
	email-EuAll	9.77	88.71	1,086.45	6,713.19	229.76
	com-dblp	922.63	12,365.75	2,183.17	21,232.16	29,944.44
	web-NotreDame	193.25	3,297.11	999.11	10,472.47	6,038.25
	twitter-munmun	189.88	1,793.71	2,759.99	27,636.73	4,210.28

TABLE 5: Average execution time (in sec) of *i*CENTRAL (for 20 cores and 1 core), QUBE, and BRANDES.

	Graph dataset	*i*CENTRAL		GREEN			
		Time (in sec)	Memory (in GB)	Time (in sec)	Memory (in GB)	Time ratio	Memory ratio
Small datasets	ER	18.54	0.03	0.17	4.84	89.29	161.33
	PA	16.56	0.03	0.21	4.84	51.91	161.33
	FF	9.94	0.04	0.68	4.84	32.86	121.00
	Cagr	0.87	0.0156	0.06	2.99	16.14	191.67
	Epa	0.46	0.0156	0.04	3.01	8.86	192.95
	Eva	0.02	0.0137	0.09	3.05	0.08	222.63
	Erdos972	0.36	0.0146	0.05	3.26	5.22	223.29
	Erdos02	0.27	0.0156	0.04	3.68	6.40	235.90
	Wiki-Vote	5.59	0.0332	0.21	3.72	30.83	112.05
	Contact	11.25	0.0303	0.52	6.6	21.46	217.82
	slashdot	45.47	0.0397	3.33	60.9	7.66	1,534.00
	facebook	914.30	0.1709	9.31	92.4	117.85	540.67
Large	epinions	827.27	0.1611	Crashed	319	N/A	1,980.14
	email-EuAll	88.71	0.0850	Crashed	1,132	N/A	13,317.65
	com-dblp	12,365.75	0.2842	Crashed	2,249	N/A	7,913.44
	web-NotreDame	3,297.11	0.2529	Crashed	2,374	N/A	9,387.11
	twitter-munmun	1,793.71	0.2051	Crashed	4,836	N/A	23,578.74

TABLE 6: Average execution time and memory consumption comparison of *i*CENTRAL and GREEN.

for the email-EuAll dataset and more than 10 times faster on average. Note that *i*CENTRAL provides a similar scalability to BRANDES, even though it employs many more optimizations tuned for the incremental set-up.

The synthetic graphs ER and PA highlight the advantage of *i*CENTRAL over QUBE. These graphs decompose into one component. QUBE needs more time than BRANDES (i.e., simple recomputation from scratch), because the overhead of computing and maintaining the minimum union cycles decomposition is not amortized. Although *i*CENTRAL also decomposes these graphs to one large biconnected component, the overhead is amortized because of the incremental computation within the affected component.

GREEN: *i*CENTRAL and GREEN are both incremental algorithms but they are fundamentally different: GREEN stores intermediate results, while *i*CENTRAL does not. Due to the stored results, GREEN is expected to be more efficient on small graphs, but storage becomes prohibitive for large graphs. In this experiment, we restrict *i*CENTRAL to use one core. We measure the execution time and the memory consumption reported by the operating system. The results are illustrated in Table 6. GREEN crashed while processing the large graphs, due to excessive memory consumption. Specifically, GREEN requested 319GB RAM for the epinions dataset, and nearly 5TB RAM for the twitter-munmun dataset. As expected for the smaller datasets, GREEN is faster but requires significantly more memory than *i*CENTRAL. For a meaningful comparison, we report the time and memory ratio of the two algorithms. The time ratio indicates how faster GREEN is over *i*CENTRAL: Time ratio $= \frac{i\text{CENTRAL runtime}}{\text{GREEN runtime}}$ while the memory ratio shows how much more memory is required by GREEN over *i*CENTRAL: Memory ratio $= \frac{\text{GREEN memory}}{i\text{CENTRAL memory}}$. Table 6 shows that although GREEN is on average 33 times faster than *i*CENTRAL for the small datasets, it consumes on average nearly 326 times more memory, and cannot scale to the large datasets.

Graph dataset	*i*CENTRAL	LEE-BCC
epinions	518.99	970.82
email-EuAll	60.35	177.04
com-dblp	6,476.23	8,696.06
web-NotreDame	1,672.67	2,005.57
twitter-munmun	706.49	2,634.20

TABLE 7: Average execution time (in sec) of *i*CENTRAL and LEE-BCC. The largest biconnected component of the corresponding dataset is used as input in this experiment.

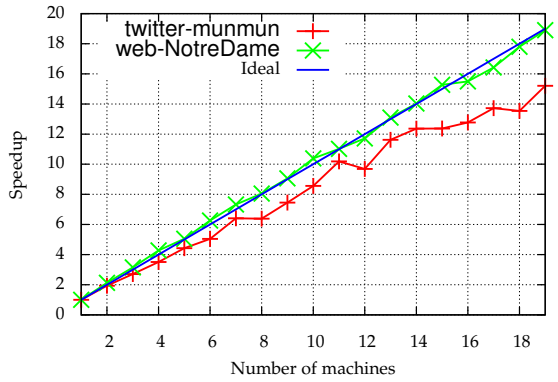

Fig. 4: Scalability of *i*CENTRAL on web-NotreDame and twitter-munmun datasets using our 19 machines computer cluster.

LEE-BCC: We compare *i*CENTRAL against LEE-BCC on large graphs, where we expect the differences to be more pronounced. Table 7 shows the runtimes of *i*CENTRAL and LEE-BCC. *i*CENTRAL is up to 3.73 times faster than LEE-BCC for the twitter-munmun dataset. *i*CENTRAL is on average 2.21 times faster than LEE-BCC over all large datasets. For a fair comparison with LEE-BCC, we select edges from the largest biconnected component when evaluating *i*CENTRAL in this experiment.

4.3 Evaluating the scalability of *i*CENTRAL

In this subsection, we evaluate the scalability of *i*CENTRAL on our 19 machine computer cluster. We use two real large datasets in this experiment (web-NotreDame and twitter-munmun). We insert 5 random edges and report the average runtime for each machine count in this experiment. Fig. 4 shows the speedup as the number of machines increases. The speedup for web-NotreDame is close to ideal, and in some cases (i.e., for machine counts less than 12) is slightly super linear. We believe that the superlinear speedup is due to larger collective cache size. For the twitter-munmun dataset, the speedup is close to ideal until 14 machines. The twitter-munmun dataset has a smaller largest biconnected component (twitter-munmun decomposes into smaller biconnected components, see Table 8), which results in less work per machine, and when the number of machines becomes higher, the computation per machine is not large enough to amortize the synchronization costs. The speedup efficiency of *i*CENTRAL on web-NotreDame and twitter-munmun is at least 0.8 on our computer cluster.

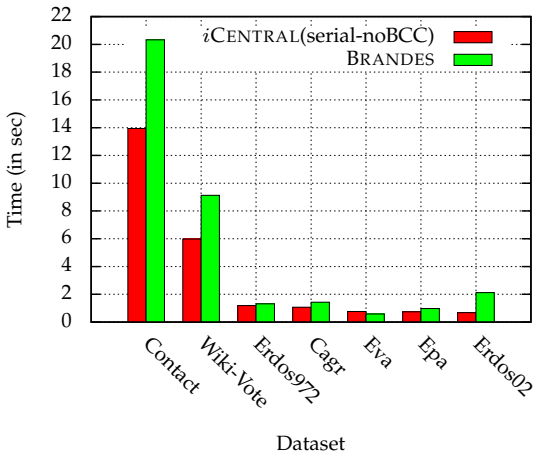

Fig. 5: Evaluating the incremental computation part of *i*CENTRAL.

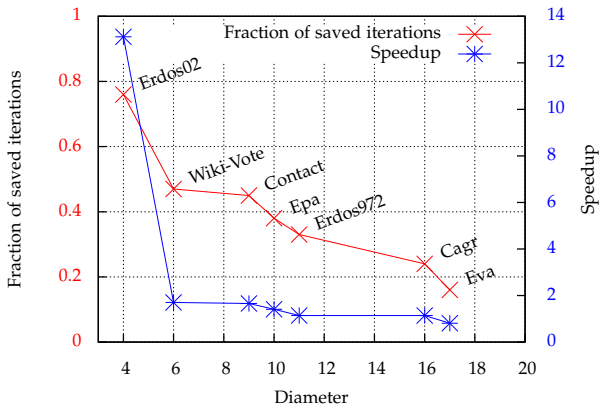

Fig. 6: Effect of diameter on the average speedup of the incremental computation part of *i*CENTRAL(serial-noBCC).

4.4 Evaluating *i*CENTRAL components

In this subsection, we evaluate the different components of *i*CENTRAL, namely; the incremental computation part and the biconnected components decomposition part, and investigate the graph properties that affect performance.

Incremental computation. In this experiment, we establish the benefits of using the incremental computation method discussed in Section 3.1 over a non-incremental method. To this end, we compare the performance of the serial version of *i*CENTRAL without the biconnected components decomposition, denoted by *i*CENTRAL(serial-noBCC) and BRANDES. We present the pseudo-code for *i*CENTRAL(serial-noBCC) in the appendix. We insert 100 random edges and measure the average time required to update the betweenness centrality values. The results are illustrated in Fig. 5. *i*CENTRAL(serial-noBCC) is faster than BRANDES for all datasets except for Eva dataset, even without using the biconnected components decomposition, and is up to 3.12 times faster for the Erdos02 dataset.

The performance improvement of the incremental computation part of *i*CENTRAL(serial-noBCC) happens because it does not consider the search DAGs of nodes whose

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2763951, IEEE Transactions on Parallel and Distributed Systems

Graph dataset	MUC $\frac{\lvert V_{MUC}\rvert}{\lvert V_G\rvert}$	MUC $\frac{\lvert MUC\rvert}{\lvert G\rvert}$	BCC $\frac{\lvert V_{BCC}\rvert}{\lvert V_G\rvert}$	BCC $\frac{\lvert BCC\rvert}{\lvert G\rvert}$
ER	1.00	1.00	1.00	1.00
PA	1.00	1.00	1.00	1.00
FF	0.76	0.97	**0.71**	**0.96**
Cagr	0.78	0.88	**0.64**	**0.75**
Epa	0.52	0.69	**0.51**	**0.68**
Eva	0.06	0.08	**0.05**	**0.07**
Erdos972	0.32	0.48	0.32	0.48
Erdos02	0.31	0.49	0.31	0.49
Wiki-Vote	0.68	0.96	0.68	0.96
Contact	0.63	0.89	0.62	0.89
slashdot	0.38	0.63	**0.38**	**0.62**
facebook	0.86	0.98	0.86	0.98
epinions	0.50	0.85	0.49	0.85
email-EuAll	0.16	0.33	0.16	0.33
com-dblp	0.83	0.92	**0.67**	**0.80**
web-NotreDame	0.46	0.74	**0.41**	**0.69**
twitter-munmun	0.24	0.46	0.24	0.46
amazon	0.64	0.81	**0.64**	**0.81**

TABLE 8: Largest component node and subgraph fraction for minimum union cycle (MUC) and biconnected component (BCC) decompositions.

source dependencies remain intact. These nodes have the same distance from both ends of the inserted edge. In this experiment, we show that in real graphs the fraction of such sources can be significant. For each measurement, we insert 160 random edges and measure the fraction of saved iterations and the average speedup for several datasets. The results are illustrated in Fig. 6 and are presented with an increasing order of the diameter of the dataset. It is clear that the fraction of saved iterations can be very high in low diameter graphs, and it becomes smaller when the diameter of the graph increases. The average speedup of the incremental computation component of iCENTRAL(serial-noBCC) follows the same trend. The reason is that in graphs with smaller diameter, the probability that the nodes of the inserted edge are on the same level is higher; thus, more iterations can be avoided.

Biconnected components decomposition. A graph decomposition can be evaluated by the number and the sizes of the components. In our case, a decomposition can be judged mainly by the size of its largest component. To explain this, note that: (*i*) the cost of inserting a new edge is mainly determined by the size of the component the new edge belongs to; and (*ii*) the insertion of a random edge is more probable to involve the largest component of the graph since it contains more nodes. Obviously, this probability increases with the largest component size.

We prove in Lemma A.3 that the biconnected components decomposition (BCC) results in finer subgraphs than the minimum union cycles (MUC) decomposition used in previous studies [9]. In Table 8, we quantitatively illustrate the benefit of using graph decomposition for both MUC and BCC decompositions over doing the computation on the input graph directly. Graph decomposition reduces the computation in two ways: (*i*) By eliminating iterations from sources outside the affected component. (*ii*) By reducing the size of the graph where these iterations are performed

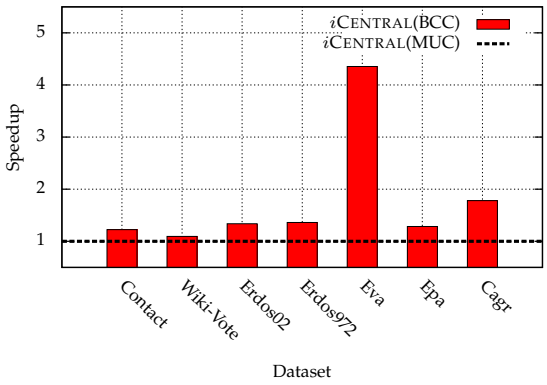

Fig. 7: Comparing graph decompositions.

(see Section 2.3 for details). The fraction of nodes in the largest component ($\frac{V_{MUC}}{V_G}$ and $\frac{V_{BCC}}{V_G}$ for MUC and BCC decompositions, respectively) shows how many iterations are saved using the respective decomposition. For example, the fraction of iterations that need to be performed if BCC decomposition is used for the slashdot dataset is 0.38, which means 62% of the iterations are skipped. The fraction of the component size to the graph size ($\frac{|MUC|}{|G|}=\frac{|V_{MUC}|+|E_{MUC}|}{|V_G|+|E_G|}$ and $\frac{|BCC|}{|G|}=\frac{|V_{BCC}|+|E_{BCC}|}{|V_G|+|E_G|}$ for MUC and BCC decompositions, respectively) shows the amount of saved work within an iteration. For example, an iteration using BCC decomposition for the slashdot dataset performs computation on a subgraph that is 38% smaller than the input graph.

We show in Table 8 that BCC decomposition is better than MUC decomposition in the sense that it produces a decomposition with smaller largest component. The largest component of BCC is never larger than the largest component of MUC. BCC can save up to 19% more iterations, with 13% less work in each iteration (for the com-dblp dataset).

To conclude this comparison, we present an evaluation of MUC and BCC decompositions for the computation of betweenness centrality for several datasets. For this evaluation, we use the incremental computation part of iCENTRAL with MUC and BCC decompositions. The results of Fig. 7 illustrate the speedup achieved by BCC over MUC. BCC is strictly faster, but the speedup ranges with different datasets and can be up to 4.3 times for the Eva dataset.

5 Related Work

Betweenness centrality was first proposed by Anthonisse [2] and Freeman [3] as a metric to quantify the centrality of a node in a graph. Freeman also proposed a $\mathcal{O}(|V|^3)$ algorithm for computing betweenness centrality. The need for more efficient algorithms emerged with the increasing popularity of this metric. Brandes proposed the fastest known algorithm that runs in $\mathcal{O}(|V||E|)$ time [1]; see Section 2.1 for details. There have been several attempts [20], [21] to improve the runtime; however, such works use heuristics that do not improve the theoretical complexity of Brandes algorithm. The aforementioned methods assume static graphs, whereas we focus on the incremental computation of betweenness centrality in evolving graphs.

One group of approaches to incremental betweenness centrality relies on storing all intermediate results, including all-pairs shortest paths and shortest path counts. When the graph is updated, the stored information is also updated and used to recalculate the affected betweenness centrality values. Green et al. [8] store all-pairs shortest distance information in the form of breadth-first DAGs; specifically, they store one DAG for every node in the graph. When an edge is inserted, incremental breadth-first search is performed in the affected DAGs and betweenness centrality values are updated in a way similar to that of Brandes algorithm. Green uses the optimization proposed in [22], where the parent lists are not stored but rather computed when needed, to decrease the storage requirement from $\mathcal{O}(|V|^2 + |E|)$ to $\mathcal{O}(|V|^2)$. Kas et al. [23], [24] also store intermediate results. They use an existing incremental all-pairs shortest path algorithm to find the updated shortest distances and shortest path counts on edge insertions, and introduce algorithms that use the updated values to compute the new betweenness centrality values. Kas' algorithm was further optimized by Wei et al. [25]. Many other approaches also store the intermediate all-pairs shortest path information in various ways [26], [27], [28], [29]. Kourtellis et al. [11] extend Green's algorithm [8] and deploy a Hadoop implementation on a large cluster to support large graphs. All aforementioned works need $\Omega(|V|^2)$ space to store intermediate data, which is prohibitive for large graphs. In contrast, *i*CENTRAL requires only linear space, and thus seamlessly scales to large graphs.

A second group of research efforts to compute incremental betweenness centrality decomposes the graph into components and recomputes the metric only within the affected component. QUBE [9] is the first system that facilitates this approach. It decomposes the graph into minimum union cycles (MUC), which are maximal subgraphs separated by bridge edges. Computation is performed with a modified version of Brandes algorithm. QUBE implements an expensive $\mathcal{O}(|E|^3)$ preprocessing step to extract MUCs; Goel et al. [18] propose a linear time alternative. Independently and concurrently with our approach, the same authors proposed an improved version [10] that uses biconnected components, which is a finer decomposition (Lemma A.3). Both methods recompute all values from scratch for all nodes in the affected component. However, real graphs often contain a large MUC or biconnected component; with high probability, a random edge involves a node in the large component. In such a case, both methods become very slow. In contrast, *i*CENTRAL is much faster because it implements a novel incremental method that avoids redundant recomputation *within* the affected component.

In another line of research, many research efforts consider approximating betweenness centrality [30], [31], and some more recent efforts offer approximation methods to compute the changes of betweenness centrality values on graph updates [32], [33]. In contrast, our work deals with the exact calculation of betweenness centrality. Note that computing exact betweenness centrality updates for truly massive graphs (i.e., billions of nodes and edges) might not be fast enough, even using efficient incremental methods. In such a case, approximate betweenness centrality algorithms might offer a fast alternative on the expense of accuracy. A study of the available approximation algorithms and their appropriateness for massive graphs is available in [19].

Biconnected components have been used for improving the runtime of computing and updating other shortest-path based centrality metrics. For example, biconnected components are used [34] as a heuristic for updating *closeness* centrality. Other works use biconnected components to accelerate the betweenness centrality computation in *static* graphs [21]. In contrast, focus on the more complex case of dynamically evolving graphs.

There are many research works on scaling the computation of betweenness centrality to large graphs using parallel and distributed architectures [35], [36], [37], [38], [39]. However, these works focus on static graphs, unlike in this paper where the focus is dynamic graphs.

6 CONCLUSIONS

We presented an approach for incremental betweenness centrality computation that combines a novel method for doing incremental computation with linear space, and biconnected components decomposition. We experimentally show that our approach is faster and more scalable than existing approaches. There remain some challenges related to updating betweenness centrality in real evolving graphs. For example, batches of edges arrive at very close times or exactly at the same time in many real evolving graph settings, which makes it desirable to have solutions specifically optimized for updating betweenness centrality in the batch case. Batch processing of edges is the primary focus of our future work on incremental betweenness centrality. The available literature on parallel and distributed betweenness centrality computation in static graphs can be useful for the dynamic graph case. Hence, our future work will involve investigating the applicability of available ideas on parallel betweenness centrality computation in static graphs to dynamic graphs.

REFERENCES

[1] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.
[2] J. M. Anthonisse, "The rush in a directed graph," Stichting Mathematisch Centrum, Amsterdam, Tech. Rep., 1971.
[3] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, pp. 35–41, 1977.
[4] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, pp. 7821–7826, 2002.
[5] A. Özgür, T. Vu, G. Erkan, and D. R. Radev, "Identifying gene-disease associations using centrality on a literature mined gene-interaction network," *Bioinformatics*, vol. 24, pp. 277–285, 2008.
[6] E. M. Daly and M. Haahr, "Social network analysis for routing in disconnected delay-tolerant manets," in *Proceedings of the ACM MobiHoc*, 2007, pp. 32–40.
[7] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu, "Graphscope: parameter-free mining of large time-evolving graphs," in *Proceedings of the ACM SIGKDD*, 2007, pp. 687–696.
[8] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for streaming betweenness centrality," in *Proceedings of the ASE/IEEE International Conference on Social Computing*, 2012, pp. 11–20.
[9] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung, "Qube: a quick algorithm for updating betweenness centrality," in *Proceedings of WWW*, 2012, pp. 351–360.
[10] M.-J. Lee, S. Choi, and C.-W. Chung, "Efficient algorithms for updating betweenness centrality in fully dynamic graphs," *Information Sciences*, vol. 326, pp. 278–296, 2016.

[11] N. Kourtellis, G. De Francisci Morales, and F. Bonchi, "Scalable online betweenness centrality in evolving graphs," *IEEE TKDE*, vol. 27, no. 9, pp. 2494–2506, 2015.

[12] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*. Prentice Hall College Div, 1977.

[13] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commununcations of the ACM*, vol. 16, pp. 372–378, 1973.

[14] G. Cong and D. A. Bader, "An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps)," in *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE, 2005.

[15] P. Erdős and A. Rényi, "On random graphs," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

[16] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, 1999.

[17] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the ACM SIGKDD*, 2005, pp. 177–187.

[18] K. Goel, R. R. Singh, S. Iyengar *et al.*, "A faster algorithm to update betweenness centrality after node alteration," in *Proceedings of the international workshop on Algorithms and models for the web-graph*, 2013, pp. 170–184.

[19] Z. Alghamdi, F. Jamour, S. Skiadopoulos, and P. Kalnis, "A benchmark for betweenness centrality approximation algorithms on large graphs," in *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2017.

[20] D. Erdős, V. Ishakian, A. Bestavros, and E. Terzi, "A divide-and-conquer algorithm for betweenness centrality," in *Proceedings of the SIAM SDM*, 2015, pp. 433–441.

[21] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes, "Heuristics for speeding up betweenness centrality computation," in *Proceedings of the IEEE PASSAT/SocialCom*, 2012, pp. 302–311.

[22] O. Green and D. A. Bader, "Faster betweenness centrality based on data structure experimentation," *Procedia Computer Science*, vol. 18, pp. 399–408, 2013.

[23] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley, "Incremental algorithm for updating betweenness centrality in dynamically growing networks," in *Proceedings of the IEEE/ACM ASONAM*, 2013, pp. 33–40.

[24] M. Kas, K. M. Carley, and L. R. Carley, "An incremental algorithm for updating betweenness centrality and k-betweenness centrality and its performance on realistic dynamic social network data," *Social Network Analysis and Mining*, vol. 4, pp. 1–23, 2014.

[25] W. Wei and K. Carley, "Real time closeness and betweenness centrality calculations on streaming network data," in *Proceedings of the ASE BigData Conference*, 2014.

[26] S. S. Khopkar, R. Nagi, A. G. Nikolaev, and V. Bhembre, "Efficient algorithms for incremental all pairs shortest paths, closeness and betweenness in social network analysis," *Social Network Analysis and Mining*, vol. 4, pp. 1–20, 2014.

[27] M. Nasre, M. Pontecorvi, and V. Ramachandran, "Decremental all-pairs all shortest paths and betweenness centrality," in *Proceedings of the International Symposium on Algorithms and Computation*, 2014, pp. 766–778.

[28] M. Nasre, M. Pontecorvi, and V. Ramachandran, "Betweenness centrality–incremental and faster," in *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, 2014, pp. 577–588.

[29] M. Pontecorvi and V. Ramachandran, "Fully dynamic betweenness centrality," in *Proceedings of the International Symposium on Algorithms and Computation*, 2015, pp. 331–342.

[30] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proceedings of the international workshop on Algorithms and models for the web-graph*, 2007, pp. 124–137.

[31] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality." in *Proceedings of the SIAM ALENEX*, 2008, pp. 90–100.

[32] T. Hayashi, T. Akiba, and Y. Yoshida, "Fully dynamic betweenness centrality maintenance on massive networks," *Proceedings of the VLDB Endowment*, vol. 9, pp. 48–59, 2015.

[33] E. Bergamini, H. Meyerhenke, and C. L. Staudt, "Approximating betweenness centrality in large evolving networks," in *Proceedings of the SIAM ALENEX*, 2015, pp. 133–146.

[34] A. E. Sarıyüce, K. Kaya, E. Saule, and U. V. Catalyürek, "Incremental algorithms for closeness centrality," in *Proceedings of IEEE International Conference on BigData*, 2013, pp. 487–492.

[35] G. Tan, D. Tu, and N. Sun, "A parallel algorithm for computing betweenness centrality," in *Proceedings of the International Conference on Parallel Processing*. IEEE, 2009, pp. 340–347.

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

[37] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proceedings of the IEEE IPDPS*, 2009, pp. 1–8.

[38] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory," in *Proceedings of the IEEE HiPC*, 2010, pp. 1–10.

[39] A. McLaughlin and D. A. Bader, "Revisiting edge and node parallelism for dynamic gpu graph analytics," in *Proceedings of the International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2014, pp. 1396–1406.

Fuad Jamour received his BS degree in computer engineering with Honours from the Univ. of Jordan in 2011 and his MS degree in computer science from the King Abdullah Univ. of Science and Technology (KAUST) in 2013. He is currently pursuing a PhD degree in computer science at KAUST. His research interests include Graph Algorithms and Social Network Analysis.

Spiros Skiadopoulos is currently a Professor at Dept. of Informatics and Telecommunications at University of Peloponnese. He received a diploma and a PhD degree from the National Technical University of Athens and a MPhil degree from Manchester Institute of Science and Technology (UMIST). He has worked in a variety of areas including data management, knowledge representation and reasoning. His current research interests include management of big and complex data.

Panos Kalnis is professor and chair of the Computer Science program in the King Abdullah Univ. of Science and Technology (KAUST). In 2009 he was visiting assistant professor in the CS Dept., Stanford University. Before that, he was assistant professor in the CS Dept., National University of Singapore (NUS). From 2013 to 2015 he was associate editor for TKDE. Currently, he serves on the editorial board of the VLDB Journal and the Data Science and Engineering Journal. He received his Diploma from the Computer Engineering and Informatics Dept., Univ. of Patras, Greece in 1998 and his PhD from the Computer Science Dept., Hong Kong Univ. of Science and Technology (HKUST) in 2002. His research interests include Big Data, Cloud Computing, Parallel and Distributed Systems, Large Graphs and Long Sequences.