A Round-Efficient Distributed Betweenness Centrality Algorithm

Loc Hoang
The University of Texas at Austin
loc@cs.utexas.edu

Matteo Pontecorvi
Nokia Bell Labs
matteo.pontecorvi@nokia.com

Roshan Dathathri
The University of Texas at Austin
roshan@cs.utexas.edu

Gurbinder Gill
The University of Texas at Austin
gill@cs.utexas.edu

Bozhi You
Xi'an Jiaotong University
youbozhi@stu.xjtu.edu.cn

Keshav Pingali
The University of Texas at Austin
pingali@cs.utexas.edu

Vijaya Ramachandran
The University of Texas at Austin
vlr@cs.utexas.edu

Abstract

We present Min-Rounds BC (MRBC), a distributed-memory algorithm in the CONGEST model that computes the betweenness centrality (BC) of every vertex in a directed unweighted n-node graph in $O(n)$ rounds. Min-Rounds BC also computes all-pairs-shortest-paths (APSP) in such graphs. It improves the number of rounds by at least a constant factor over previous results for unweighted directed APSP and for unweighted BC, both directed and undirected.

We implemented MRBC in D-Galois, a state-of-the-art distributed graph analytics system, incorporated additional optimizations enabled by the D-Galois model, and evaluated its performance on a production cluster with up to 256 hosts using power-law and road networks. Compared to the BC algorithm of Brandes, on average, MRBC reduces the number of rounds by 14.0× and the communication time by 2.8× for the graphs in our test suite. As a result, MRBC is 2.1× faster on average than Brandes BC for real-world web-crawls on 256 hosts.

CCS Concepts • **Theory of computation → Shortest paths**; **Massively parallel algorithms**; **Distributed algorithms**.

ACM Reference Format:
Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. 2019. A Round-Efficient Distributed Betweenness Centrality Algorithm. In *24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '19), February 16–20, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3293883.3295729

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295729

1 Introduction

Centrality metrics are useful for analyzing network structure since they capture the relative importance of individual vertices in the network. In this paper, we focus on *Betweenness Centrality* (BC) [22], a metric based on the computation of shortest paths between vertices in the network graph. Intuitively, BC measures the degree of control a vertex has over communication between vertices in the network. If $G = (V, E)$ is a graph and s, t are a pair of vertices, the betweenness score of a vertex v for this vertex pair is the fraction of shortest paths between s and t that include v. The BC of v is the sum of its betweenness scores for all pairs of vertices in the graph.

BC has been used to find key actors in terrorist networks [15, 36], study the spread of sexual diseases [17, 33, 41], and analyze power grid component failures [34]. In most applications, the networks are *unweighted*, directed graphs with billions of vertices and edges [44], so we focus on such graphs.

Most implementations of BC [7, 18, 19, 26, 28, 43, 52, 56, 57, 61] use a standard implementation of the Brandes BC algorithm [13] (Section 2) and perform breadth first search from each vertex to compute shortest paths. Solomonik *et al.* [53] implement a sparse-matrix based BC algorithm called Maximal-Frontier BC that uses the Bellman-Ford algorithm [53] to compute shortest paths from each vertex. We present a new distributed-memory algorithm called Min-Rounds BC (MRBC) formulated in the CONGEST model (Section 2) [21, 25, 37, 42, 45, 47, 49] that computes the BC of every vertex in an unweighted, directed graph based on a solution to the all-pairs-shortest-paths (APSP) problem in such graphs. For an unweighted, directed graph with n vertices and m edges, it executes $2n + O(D)$ rounds, where D is the (finite) directed diameter, and sends no more than $2mn + 2m$ messages in the CONGEST model.

To evaluate MRBC, we implement it in D-Galois, the state-of-the-art distributed graph analytics system created using the Gluon communication substrate [16], and establish an optimization of MRBC in this system to reduce communication volume. We compare the performance of MRBC with that of other BC algorithms on power-law and road networks using a large production cluster. Compared to the classical Brandes BC algorithm, MRBC reduces the number of executed rounds by 14.0× and the communication time by 2.8× on average in our evaluation. Since the execution time of graph algorithms on distributed-memory is dominated by communication time, MRBC is faster than other BC algorithms in our evaluation for non-trivial diameter graphs even though it may perform more computation. On average, MRBC is 3.0× faster than Maximal-Frontier BC [53], and for real-world web-crawls on 256 hosts, MRBC is 2.1× faster than Brandes BC.

Our paper makes the following contributions:

- We present a new distributed betweenness centrality algorithm called Min-Rounds BC in the CONGEST model that is provably round efficient compared to past betweenness centrality algorithms (Section 3).
- We implement Min-Rounds BC in the state-of-the-art distributed graph analytics system, D-Galois, and present an optimization that exploits algorithmic properties to optimize communication (Section 4).
- We evaluate our implementation against Brandes BC [13] and Maximal-Frontier BC [53] on a large production cluster and show that it outperforms these algorithms on real-world graphs at scale (Section 5).

2 Background

Let $G = (V, E)$ be a directed graph. For a vertex $u \in V$ we define $\Gamma_{\text{in}}(u) = \{v \in V \mid (v, u) \in E\}$ as the set of *incoming neighbors* of u and $\Gamma_{\text{out}}(u) = \{v \in V \mid (u, v) \in E\}$ as the set of the *outgoing neighbors* of u. A directed graph G is *strongly connected* if every vertex is reachable from every other vertex. The *diameter* of a graph is the largest distance between any pair of vertices. We let U_G denote the undirected version of G. A directed graph G is *weakly connected* if U_G is connected. We let $\delta(x, y)$ denote the shortest path distance from x to y, with $\delta(x, y) = \infty$ if there is no path.

Betweenness Centrality. Let $G = (V, E)$ be a directed graph with $|V| = n$, $|E| = m$, and with a positive edge weight $\mathbf{w}(e)$ on each edge $e \in E$. Let σ_{xy} denote the number of shortest paths (SPs) from x to y in G, and $\sigma_{xy}(v)$ the number of SPs from x to y in G that pass through v, for each pair $x, y \in V$. Then, $BC(v) = \sum_{s \neq v, t \neq v} \frac{\sigma_{st}(v)}{\sigma_{st}}$.

2.1 Brandes' Betweenness Centrality Algorithm

Brandes [13] noted that if single source shortest path (SSSP) DAGs are available for each $v \in V$ we can compute BC values with a recursive *accumulation* technique.

Algorithm 1 Betweenness-centrality($G = (V, E)$) ([13])

1: **for** every $v \in V$ **do** BC(v) $\leftarrow 0$
2: **for** every $s \in V$ **do**
3: run Dijkstra SSSP from s (or BFS if G is unweighted)
4: $\forall$ t $\in$ V\{s}. compute σ_{st} and $P_s(t)$
5: store vertices in stack S in non-increasing distance from s
6: accumulate dependency $\delta_{s\bullet}(t)$ of s on all $t \in V \setminus s$ using Algorithm 2

Algorithm 2 Accumulation-phase(s, S) ([13])

Require: $\forall t \in V$: σ_{st}, $P_s(t)$; a stack S containing all $v \in V$ in non-increasing distance $d(s, v)$ value
1: **for** every $v \in V$ **do** $\delta_{s\bullet}(v) \leftarrow 0$
2: **while** $S \neq \emptyset$ **do**
3: $w \leftarrow \text{pop}(S)$
4: **for** $v \in P_s(w)$ **do** $\delta_{s\bullet}(v) \leftarrow \delta_{s\bullet}(v) + \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))$
5: **if** $w \neq s$ **then** BC(w) $\leftarrow$ BC(w) + $\delta_{s\bullet}(w)$

$$BC(v) = \sum_{s \neq v} \delta_{s\bullet}(v) \quad \text{where} \quad \delta_{s\bullet}(v) = \sum_{t \in V \setminus \{v,s\}} \frac{\sigma_{sv} \cdot \sigma_{vt}}{\sigma_{st}}$$

Moreover, $\delta_{s\bullet}(v)$ can be recursively computed as $\delta_{s\bullet}(v) = \sum_{w: v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w))$, where $P_s(w)$ is the set of predecessors of w in the SSSP DAG rooted at s.

Brandes' sequential BC algorithm consists of the following steps: for each source s compute the SSSP DAG rooted at s, $DAG(s)$ (Alg. 1), for each $DAG(s)$ compute σ_{sv} for each $v \in DAG(s)$ (Alg. 1) and, for each $DAG(s)$ starting from the leaves, apply the recursive equation for $\delta_{s\bullet}$ (given above) up to the root (Alg. 2).

2.2 CONGEST Model

In the CONGEST model, a network of processors is generally modeled by an undirected graph $G = (V, E)$, with $|V| = n$ and $|E| = m$. Each vertex has infinite computational power. We assume the vertices are numbered from 1 to n and we denote the vertex i by v_i. If the graph $G = (V, E)$ is directed then it is assumed that the communication channels (edges in G) are bidirectional, i.e., the communication network is represented by U_G.

The performance of an algorithm is measured by the number of *rounds* it needs. In a single round a vertex $v \in V$ can receive an $O(\log n)$-bit message along each incoming edge (u, v). Vertex v processes its received messages (instantaneously, given its infinite computational power) and then can send a (possibly different) $O(\log n)$-bit message along each outgoing edge; v can choose not to send a message along some of its edges in a given round. The goal is to design distributed algorithms for the graph G using a small number of rounds. Additionally, we would like the total number of messages sent across all edges in all rounds to be small.

In the next section, we present the first nontrivial distributed algorithm for BC in unweighted directed graphs. At the same

time we also improve the round and/or message complexity (by a constant factor) for APSP in both undirected and directed graphs and for BC in undirected graphs. Prior to our work, the best previous CONGEST algorithms for unweighted APSP were in [38] and the only nontrivial CONGEST algorithm for BC was the undirected unweighted BC algorithm in [31].

3 Min-Rounds BC

In this section, we present our new algorithm for computing betweenness centrality in unweighted directed graphs. It is inspired by the Lenzen-Peleg distributed unweighted APSP algorithm [38], which was presented as an APSP algorithm for unweighted undirected graphs, but whose $2n$-round version also works for directed graphs. Our APSP algorithm contains new elements discussed starting Section 3.1 up to Section 3.4. Section 3.5 gives our distributed algorithm for the accumulation phase (Alg. 2) in Brandes' algorithm and our overall BC algorithm.

3.1 Our Contributions

Theorem 1 states our main results. We use D to denote the diameter of a directed graph, and D_u for an undirected graph. Lemma 8 in Section 3.5 states a version of this theorem that applies to the implementation in our experiments.

Theorem 1. *On an unweighted graph G with n nodes and m edges,*
(I) *Algorithm 3 computes directed APSP with the following bounds in the CONGEST model:*

1. *If n is known, in $\min\{n + O(D), 2n\}$ rounds while sending $mn + O(m)$ messages in any graph.*
2. *If n is known, in $2n$ rounds while sending at most mn messages in any graph (by omitting Steps 1 and 10).*
3. *If n is not known, in $n + O(D)$ rounds while sending at most $mn + O(m)$ messages if G is strongly connected.*

(II) *Algorithm 5 computes BC values of all vertices with at most twice the number of rounds and messages as in part (I) for each of the three cases.*
(III) *If G is undirected the bounds for rounds and messages in parts (I) and (II) hold with D replaced by D_u.*

Parts I.1 and I.3 of Theorem 1 improve over the $2n$-round algorithm in [38] while sending a smaller number of messages. The number of messages sent is also improved for undirected graphs when compared to [38], where up to $2mn$ messages or $mn + O(m \cdot D_u)$ messages can be sent. Moreover, part I.3 of Theorem 1 computes APSP without knowing n when D is bounded: this case is not considered in [38] where knowledge of n is needed for directed APSP. For message count, Part I.2 of Theorem 1 further reduces the number of messages to at most one message sent by each node for each source.

In the case when exponential numbers of shortest paths exist in the graph, we can use the approximation technique introduced in [31] which uses only $O(\log n)$-size messages and computes a provably good approximation of the BC values.

Undirected versus Directed APSP (and BC). As noted earlier, the APSP algorithm in [38] is a correct $2n$-round algorithm for unweighted directed graphs even though it was presented as an undirected APSP algorithm. By using the height of a BFS-tree as a 2-approximation of D_u, an alternate $n + O(D_u)$-round bound is obtained in [38] for APSP in an undirected connected graph. However, this result does not hold for directed BFS and directed diameter. Instead, our Algorithm 4 uses a different method to achieve an $n + O(D)$-round bound for directed strongly-connected graphs.

There are other $O(n)$-round undirected APSP algorithms [29, 48] but these require bidirectional edges and do not work for directed graphs (for example, the use of distances along a pebble traversal of a BFS tree in the proof of Lemma 1 in [29]). Similarly, the undirected BC algorithm in [31] does not work for directed graphs even if we substitute a directed APSP algorithm since their method for the accumulation phase is tied to the undirected APSP method in [29].

New Techniques. We introduce the following new pipelining methods for the CONGEST model.

(i) A simple *timestamp* pipelining technique based on reversing global delays that occur during a forward execution of a distributed algorithm. This general method is applicable when certain specific operations have to be back-propagated during a reverse pass of the algorithm. We use this technique in the Accumulation Phase for the BC scores following an APSP computation (Section 3.5).

(ii) A refinement of the pipelining technique in [38] to obtain a simpler APSP algorithm for unweighted directed graphs that is faster and more message-efficient (by a constant factor) (Theorem 1 and Sections 3.3-3.5).

Additional details on the above techniques, and a novel $O(n)$-round APSP algorithm for weighted directed acyclic graphs can be found in [50]. Recently, new deterministic algorithms for weighted APSP [4] were obtained by building on these pipelining techniques.

3.2 The Lenzen-Peleg APSP Algorithm [38]

We first review some notation common to [38] and our Alg. 3 for directed APSP. L_v is an ordered list at vertex v which stores pairs (d_{sv}, s), where s is a source and d_{sv} is the current estimate on the shortest distance from s to v. These pairs are stored in lexicographically sorted order, with $(d_{rv}, r) < (d_{sv}, s)$ if either $d_{rv} < d_{sv}$, or $d_{rv} = d_{sv}$ and $r < s$. L_v^r denotes the state of L_v at the beginning of round r.

In each round r of the Lenzen-Peleg algorithm [38], each vertex v sends along its outgoing edges the pair with smallest index in L_v^r whose $status$ (a conditional flag) is set to $ready$; v then sets the $status$ of this pair to $sent$. As noted in [38] this

Algorithm 3 Directed-APSP(G)

1: compute (in parallel with Step 7) a BFS tree B rooted at vertex v_1 (node with smallest ID); each vertex u computes its set of children C_u and its parent p_u in B ▷ This will be used in Alg. 4
2: **for** each vertex v in G **do**
3: $L_v \leftarrow ((0, v))$; set flag $f_v \leftarrow 0$ ▷ Initialize
4: **for** each source s in G **do if** $s = v$ **then** $\sigma_{vv} \leftarrow 1$ **else** $\sigma_{sv} \leftarrow 0$; $P_s(v) \leftarrow \emptyset$
5: **if** n is not known **then** ▷ Assumes G is weakly-connected
6: compute and broadcast n to every node in at most $2 \cdot D_u$ rounds, where D_u is the diameter of U_G
7: **for** rounds $1 \leq r \leq 2n$ **do** ▷ Step 10 could cause termination before round $2n$ when G is strongly connected
8: **if** $r = d_{sv} + \ell_v^r(d_{sv}, s)$ **then**
9: $\tau_{sv} \leftarrow r$; send (d_{sv}, s, σ_{sv}) to all vertices in $\Gamma_{\text{out}}(v)$ ▷ Timestamp τ_{sv} will be used in Alg. 5
10: run APSP-Finalizer(v, p_v, C_v, n) ▷ See Alg. 4
11: **for** a received (d_{su}, s, σ_{su}) from an incoming neighbor u **do**
12: **if** $\nexists (d_{sv}, s) \in L_v^r$ **then**
13: vertex v adds (d_{sv}, s) in L_v with $d_{sv} = d_{su} + 1$, sets $\sigma_{sv} \leftarrow \sigma_{su}$; $P_s(v) \leftarrow \{u\}$
14: **else if** $\exists (d_{sv}, s) \in L_v^r$ with $d_{sv} = d_{su} + 1$ **then**
15: vertex v updates $\sigma_{sv} \leftarrow \sigma_{sv} + \sigma_{su}$; $P_s(v) \leftarrow P_s(v) \cup \{u\}$
16: **else if** $\exists (d_{sv}, s) \in L_v^r$ with $d_{sv} > d_{su} + 1$ **then**
17: vertex v replaces (d_{sv}, s) in L_v with $(d_{su} + 1, s)$; vertex v sets $\sigma_{sv} \leftarrow \sigma_{su}$; $P_s(v) \leftarrow \{u\}$

approach can result in multiple messages being sent from v for the same source s (in different rounds). This is simplified in our algorithm, where only one correct message is sent from each vertex v for each source. This message is sent in a specific round without the need for a *status* flag.

The Lenzen-Peleg algorithm [38] completes in $n + O(D_u)$ rounds and correctly computes shortest path distances to v from each vertex s that has a path to v (the undirected diameter, which we denote by D_u here, is called D in [38] because they only consider undirected graphs). Although this is claimed in [38] only for undirected APSP, their techniques can be adjusted to work for directed APSP as well. In particular, if the total number of vertices n is known (or computed), the undirected APSP algorithm in [38] can be modified to terminate in $2n$ rounds and compute APSP in a directed graph.

In Section 3.3 we present a method to improve the number of rounds from $2n$ to $\min\{2n, n + O(D)\}$. Our algorithm terminates in $n + 5D$ rounds on strongly connected graphs without knowing n; if n is known, it terminates in $2n$ rounds in any directed graph. Moreover, our algorithm reduces the total number of messages sent to $mn + 2m$ even for the undirected case. Finally, since we are interested in computing BC, our APSP algorithm also computes for each vertex v the set $P_s(v)$ of predecessors of v in the shortest path DAG rooted at each source s, and the number of shortest paths σ_{sv} from s to v. These enhancements appear in our new Algorithm 3, together with a call to Algorithm 4 to reduce the number of rounds to $n + O(D)$ (when D is finite). We use the output of Algorithm 3 to compute directed BC in Section 3.5.

3.3 APSP and Number of Shortest Paths

In our directed APSP algorithm (Alg. 3) initially each vertex v has just the pair $(0, v)$ in L_v (Step 3, Alg. 3). Let L_v^r be the state of L_v at the beginning of round r, and let $\ell_v^{(r)}(d_{sv}, s)$ be the index of the pair (d_{sv}, s) in L_v^r. If there is an entry on L_v with $d_{sv} + \ell_v^r(d_{sv}, s) = r$ (and there can be at most one), then this value is sent out along with the associated σ_{sv} value (Steps 8-9), otherwise v does not send out anything in round r. A received message for source s is either added to L_v or updates an existing value for s in L_v (if it improves the distance value for its source). If new shortest paths from s to v are added by this received message, the σ_{sv} value and $P_s(v)$ are updated to reflect this (Steps 11-17). Steps 1 and 10 are used to reduce the number of rounds from $2n$ to $n + O(D)$ and are discussed in Section 3.4.

Algorithm 3 may need to send more than one value from a vertex v in a round because of the parallel computation of Step 1, but it never sends more than a constant number of values. In this case, v will combine all these values into a single $O(B)$-bit message.

We now establish the correctness of Algorithm 3. We start by showing that every d_{sv} value arrives at v before the round in which it will need to be sent by v in Step 8.

Lemma 2. *If an entry (d_{sv}, s) is inserted in L_v at position k in round r then $d_{sv} + k > r$.*

Proof. We prove the lemma by contradiction. Assume the lemma does not hold and consider the first round in which it is violated. In the first round, any entry (d_{sv}, s) inserted in L_v has $d_{sv} = 1$ and the minimum value for k is 1. Hence $d_{sv} + k \geq 2 > 1$ so the lemma must hold for round 1. Now, let $r > 1$ be the first round in which the lemma does not hold and an entry (d_{sv}, s) is inserted in L_v at a position k with $d_{sv} + k \leq r$. Let this d_{sv} be inserted due to a message (d_{su}, s, σ_{su}) received by v in round r in Step 11. Then, if (d_{su}, u) was in position i in L_u in round r, $r = d_{su} + i$ and the entries in L_u in positions 1 to $i - 1$ must have been sent to v in rounds earlier than r. Each of these entries correspond to a different source, and a corresponding entry for that source will be present at a position less than k in L_v (either because a corresponding entry was inserted at L_v when the message

for it from u was received or an entry with an even smaller value for d_{sv} was already present in L_v). Hence $k \geq i$. But for the values in round r, $d_{sv} + k = d_{su} + 1 + k \geq d_{su} + i + 1$ since $d_{sv} = d_{su} + 1$ and $k \geq i$ in round r. Since $r = d_{su} + i$ we have $d_{sv} + k \geq r + 1$. This gives the desired contradiction, and the lemma is established. □

Next we show that the position of an entry for a source s in L_v can never decrease unless its value is changed.

Lemma 3. *If an entry (d_{sv}, s) in L_v remains unchanged at v between rounds r and r', with $r' > r$, then $\ell_v^{(r')}(d_{sv}, s) \geq \ell_v^{(r)}(d_{sv}, s)$.*

Proof. Once an entry is added to the list L_v it can only be replaced by a lexicographic smaller one, but it never disappears. Thus, every entry in L_v that is below (d_{sv}, s) in round r either remains in its position or moves to an even lower position in subsequent rounds. Hence if d_{sv} does not change between r and r', every entry below (d_{sv}, s) in round r remains below it until round r'. It is possible that new entries could be added below the position of (d_{sv}, s) in L_v, but this can only increase the position of (d_{sv}, s) in round r'. □

Lemma 4. *At each vertex v, the distance values in the sequence of messages sent by v are non-decreasing.*

Proof. Suppose v sends a message with value d_{sv} in round r and then sends a message with a smaller d value in a later round. Then this smaller d value must be received by v in round r or later since otherwise it would have been placed in L_v (and thus sent) before d_{sv}.

Let $k = \ell_v^r(d_{sv}, s)$. Let $d_{s'v}$ be the first d value smaller than d_{sv} that is inserted in L_v in a round $r' \geq r$. Then, $d_{s'v}$ is inserted in a position $k' \leq k$ since the d values are in non-decreasing order on L_v. But then $d_{s'v} + k' < d_{sv} + k = r \leq r'$. But this contradicts Lemma 2. □

Lemmas 2 and 3 establish that every entry that remains in L_v at the end of the algorithm was sent out at a prescribed round number (Step 9, Alg. 3) since the entry was placed at its assigned spot before that round number is reached and after it was placed in L_v its position can only increase, and hence it will be available to be sent out at the round corresponding to its new higher position. Lemma 4 shows that the distance messages are sent out in non-decreasing order, and hence at most one message is sent by each vertex for each source. The next lemma, which can be proved by induction on $\delta(s, v)$ (see [50]) shows that the shortest path counts σ_{sv} and the predecessor lists are also correctly computed.

Lemma 5. *During the execution of Algorithm 3, for each source s from which v is reachable, v sends exactly one message (d_{sv}, s, σ_{sv}). This message has $d_{sv} = \delta(s, v)$ and has the total number of shortest paths from s to v in σ_{sv}. Also, when this message is sent, $P_s(v)$ contains exactly the predecessors of v in s's SP DAG.*

3.4 Improving the Round Complexity

We now describe Algorithm 4 which guarantees that Algorithm 3 will terminate in $\min\{2n, n + O(D)\}$ rounds. More precisely, Alg. 4 terminates the computation before $n + 5D$ rounds provided G is strongly connected with $D < n/5$. Otherwise, the computation terminates necessarily within $2n$ rounds because of step 7 of Alg. 3. We now focus on the non-trivial case where G is strongly connected and D is bounded.

Let B be a BFS tree rooted at v_1 (node with smallest ID) and created in Step 1, Alg. 3. Also, let C_v be the set of children of v in B. Note that, if n is not known, Step 6 of Alg. 3 computes it in at most $2D_u \leq 2D$ rounds. Thus, n is always available during the execution of Alg. 4. The special vertex v_1 is used only to uniquely select a source node for the BFS (as in [38]). If we omit Alg. 4 (and terminate in $2n$ rounds), or if the unique BFS source vertex can be efficiently selected in some other way, there is no need to identify vertex 1, or to assume that vertices are numbered from 1 to n.

In Alg. 3 the parent and child pointers in B will be computed in D rounds, and the activity of Alg. 4 for a vertex v becomes relevant only after n rounds. In the first step, the algorithm checks if v has received the diameter D from its parent p_v in B. In this case, v broadcasts D to all its children in C_v and it stops. Otherwise, the algorithm checks if v has received one finite distance estimate from every vertex in G (Step 2, Alg. 4). (The flag f_v is initialized in Step 3 of Algorithm 3 and is used to ensure that steps 3–9 are performed only once.) These distances will be correct when round $r \geq \max_s(d_{sv} + \ell_v^{(r)}(d_{sv}, s))$ (see Lemma 5), and Algorithm 4 proceeds by distinguishing two cases: if a vertex v is a leaf in the tree B (Step 3, Alg. 4), it computes the maximum shortest distance d_v^* from any other vertex s and broadcasts d_v^* to its parent p_v in B (Step 4, Alg. 4). Then, v will wait up to round $2n$ to receive the diameter D from its parent p_v in B (because of the check in step 1, Alg. 4).

In the second case, when v is not a leaf (and not v_1), if it has collected (for the first time) the distances d_c^* from all its children in C_v (Step 6, Alg. 4), it will execute the following steps only once (thanks to the flag f_v initialized to 0 in Alg. 3, and updated to 1 in Step 8, Alg. 4): v computes the maximum shortest distance d_v^* from any source s (Step 7, Alg. 4) and the largest distance value $d_{C_v}^*$ received from its children in C_v (Step 7, Alg. 4). Then v sends the larger of d_v^* and $d_{C_v}^*$ to its parent p_v (Step 8, Alg. 4), and it waits for D from p_v as in the first case. Finally, when v is in fact v_1, after receiving the distances from all its children, it broadcasts the diameter D to its children in C_{v_1} (Step 9, Alg. 4).

It is readily seen that Algorithm 4 broadcasts the correct diameter to all vertices in G since after round $r = \max_s(d_{sv} + \ell_v^{(r)}(d_{sv}, s))$ the d_{sv} values at v are the correct shortest path lengths to v (by Lemma 5). Moreover, since $\max_s(d_{sv} + \ell_v^{(r)}(d_{sv}, s)) > n$ when $|L_v^r| = n$, Step 1 of Alg. 3 is completed and each vertex v knows its parent and its children in B. Thus,

Algorithm 4 APSP-Finalizer(v, f_v, p_v, C_v, n)
▷ p_v, C_v computed in Step 1, Alg. 3

Ensure: Compute and broadcast the network directed diameter $D < \infty$
1: **if** v receives diameter D from parent p_v in round $r < 2n$, it broadcasts D to all vertices in C_v and stops
2: **if** $|L_v^r| = n$ and $f_v = 0$ **then**
3: **if** $r = \max_s(d_{sv} + \ell_v^{(r)}(d_{sv}, s))$ and $C_v = \emptyset$ **then** ▷ v is a leaf in the BFS tree B
4: $d_v^* \leftarrow \max_s(d_{sv})$; send d_v^* to parent p_v; $f_v \leftarrow 1$
5: **if** $r \geq \max_s(d_{sv} + \ell_v^{(r)}(d_{sv}, s))$ **then** ▷ completed only once
6: **if** v has received d_x^* from all children $x \in C_v$ **then**
7: $d_v^* \leftarrow \max_s(d_{sv})$; $d_{C_v}^* \leftarrow \max_{x \in C_v}(d_x^*)$
8: **if** $v \neq v_1$ **then** send $\max(d_v^*, d_{C_v}^*)$ to parent p_v; $f_v \leftarrow 1$
9: **else** broadcast $D = \max(d_{v_1}^*, d_{C_{v_1}}^*)$ to C_{v_1}; stop

the value sent by v to its parent in Step 8 of Alg. 4 is the largest shortest path length to any descendant of v in B, including v itself. Thus, vertex v_1 computes the correct diameter of G in Step 9, Alg. 4.

Lemma 6. *The execution of Algorithm 3 requires at most* $\min\{2n, n + 5D\}$ *rounds.*

Proof. Step 1 of Alg. 3 can be completed in D rounds using standard techniques, and it is executed in parallel with the loop in step 7, Alg. 3. If n is not known, Step 6 of Alg. 3 computes it in at most $2D_u \leq 2D$ rounds. Moreover, when $D = \infty$ each vertex stops after $2n$ rounds because of step 7 of Alg. 3.

When D is bounded, each $v \in V$ will have $|L_v^r| = n$ at some round r. In Alg. 4 (called in Step 10, Alg. 3), using the parent pointers of the BFS tree B already computed (Step 1, Alg. 3), the longest shortest path value reaches v_1 within D rounds after the last vertex computes its local maximum value. At this point v_1 computes the diameter D and broadcasts it to all vertices v in at most D steps. Since $\max_v \max_s\{d_{sv} + \ell_v^{(r)}(d_{sv}, s)\} \leq n + D$, the total number of rounds is at most $n + 5D$ (including $2D_u \leq 2D$ rounds for computing n). The lemma is proved. □

3.5 Accumulation Technique and BC Computation

Algorithm 5 gives a simple distributed algorithm to implement the accumulation phase in the Brandes algorithm (Alg. 2). Our new accumulation technique is a general method that works for any distributed BC algorithm where each node can keep track of the round in which step 4 of Algorithm 1 is finalized for each source. This is the case not only for Algorithm 3 for both directed and undirected unweighted graphs, but also for the BC algorithm in [31] for undirected unweighted graphs (though our Alg. 3 uses a smaller number of rounds).

We now describe our approach. Recall that in Algorithm 3, in the round when vertex v broadcasts its finalized message (d_{sv}, s, σ_{sv}) in step 9, it also notes the absolute time of this round in τ_{sv}. Also, by Lemma 6, Alg. 3 completes in round $R = \min\{n+5D, 2n\}$. Alg. 5 sets the global clock to 0 in Step 3

Algorithm 5 BC(G)

1: run Algorithm 3 (Directed-APSP(G)) on G; let R be the termination round for Alg 3
2: {Recall that τ_{sv} is the round when v broadcasts (d_{sv}, σ_{sv}) to $\Gamma_{\text{out}}(v)$ in Step 9, Alg. 3}
3: set absolute time to 0
4: **for** each vertex v in G **do**
5: **for** all s **do** $A_{sv} = R - \tau_{sv}$
6: **for a** round $0 \leq r \leq R$ **do**
7: **if** $r = A_{sv}$ **then** send $m = \frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$ to v's predecessors
8: **for** a received m from an outgoing neighbor in $\Gamma_{\text{out}}(v)$ **do**
9: $\delta_{s\bullet}(v) \leftarrow \delta_{s\bullet}(v) + \sigma_{sv} \cdot m$

after these R rounds complete in Alg. 3. In Step 5 each vertex v computes its accumulation round A_{sv} as $R - \tau_{sv}$. Then, v computes $\delta_{s\bullet}(v)$ and broadcasts $\frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$ to its predecessors in $P_s(v)$ in round A_{sv} (Steps 6–9, Alg. 5).

Lemma 7. *In Algorithm 5 each vertex v computes the correct value of $\delta_{s\bullet}(v)$ at round $A_{sv} = R - \tau_{sv}$, and the only message it sends in round A_{sv} is $m = \frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$, which it sends to its predecessors in the SSSP DAG for s.*

Proof. We first show that at time A_{sv}, vertex v has received all accumulation values from its successors in DAG(s). This follows from the fact that in the forward phase, each successor w of v will send its message for source s to vertices in $\Gamma_{\text{out}}(w)$ in round τ_{sw}, which is guaranteed to be strictly greater than τ_{sv}. Thus, since $A_{sw} < A_{sv}$, vertex v will receive the accumulation value from every successor in the DAG for s before time A_{sv}, and hence it computes the correct values of $\delta_{s\bullet}(v)$ and $\frac{1+\delta_{s\bullet}(v)}{\sigma_{sv}}$. Further, since the timestamps A_{sv} are different for different sources s, only the message for source s is sent out by v in round A_{sv}. □

***k*-SSP Problem.** The k-SSP problem takes as input the given graph G together with a subset S of k vertices, and computes the shortest path distances and number of shortest paths only for the sources in S. Experimental results for BC usually compute shortest paths and $\delta_{s\bullet}$ only for a small random subset of sources s, which suffices to obtain a good approximation of exact BC as shown in [6]. The corresponding shortest path computation in the forward phase is k-SSP using this random set of sources.

The computation in Algorithm 3 can terminate at the end of round r if no node sent a message during round r, and no node has an entry in L_v such that $d_{sv} + L_v^r(d_{sv}, s) > r$. Our experimental set-up on D-Galois can detect this global termination condition efficiently, without additional overhead. The following lemma applies to the algorithm used for our experimental results.

Lemma 8. *If the distributed system can detect global termination, k-SSP can be computed in at most $k + H$ rounds and $m \cdot k$ messages, where H is the largest finite shortest path distance from the k sources. The number of rounds and*

messages for computing BC using k sources are both at most double the bound for k-SSP.

The proof of Lemma 8 is readily obtained by modifying Algorithm 3 so that the initialization in Step 3 occurs only at the k source nodes (with L_v set to $\emptyset$ for all other nodes). We omit a call to Algorithm 4 since the system can detect global termination. For the overall BC algorithm the timestamp technique in Algorithm 5 ensures that it takes at most twice the number of rounds and messages as the k-SSP computation.

4 D-Galois Implementation

We implemented Min-Rounds BC in D-Galois [16], the state-of-the-art distributed-memory graph analytics system. A description of the D-Galois model is given in Section 4.1. We discuss implementation details of Min-Rounds BC in this model in Section 4.2. Optimizations for Min-Rounds BC in the D-Galois model are described in Section 4.3.

4.1 D-Galois Programming and Execution Model

D-Galois is a distributed version of shared-memory Galois [1] built using the communication substrate Gluon [16]. D-Galois supports vertex programs: each vertex in the graph has one or more labels which are initialized at the beginning of the computation and updated by applying a computation rule called an *operator* to the *active* vertices during the program execution until a global quiescence condition is reached.

To execute these programs on distributed-memory clusters, D-Galois uses Gluon-provided graph partitioners to partition the input graph among the hosts of the cluster. Partitioning strategies supported in Gluon include general vertex-cuts, edge-cuts, and Cartesian cuts [16, 27]. Abstractly, these strategies partition the edges of the graph among the hosts using heuristics and create *proxy vertices* (*proxies* for short) on each host for the endpoints of edges assigned to that host. Since the edges connected to a given vertex in the original graph may be partitioned among several hosts, a vertex in the original graph may have proxies on several hosts.

Distributed execution in D-Galois is performed in Bulk-Synchronous Parallel (BSP) rounds [58]. Each round consists of computation followed by communication. During the computation phase, each host operates on its own portion of the graph independent of other hosts and updates labels on its proxies. Therefore, if a vertex has proxies on two or more hosts, the labels of these proxies may have different values at the end of local computation: they are reconciled during the communication phase. Generally, in graph analytics applications, it is sufficient to do an *all-reduce* on the proxies of a given node: the labels of all proxies are reduced with an application-specific reduction operation (user-specified with the Gluon API), and the labels of all proxies are updated to this value. During communication, control is passed to Gluon, which performs reconciliation for the proxies.

Gluon reduces communication volume by using a number of communication optimizations. Gluon automatically exploits partitioning constraints to avoid the default all-reduce operation (e.g., proxies are reconciled for edge-cuts using only a reduce or a broadcast). Also, users can specify the vertices whose labels have been updated in the current round (tracking updates in the *operator* is trivial) using the Gluon API, and Gluon avoids resending labels that have not been updated in the current round while compressing the metadata that identifies the proxies whose labels are sent.

4.2 Implementation of Min-Rounds BC

The CONGEST model reflects the D-Galois model since each BSP round is also a CONGEST round, and the sends and receives at each vertex in a CONGEST round map naturally on to the vertex programs supported by D-Galois. Consequently, Min-Rounds BC can be mapped to the D-Galois model in a straightforward way. Each round in Min-Rounds BC maps to a BSP round in D-Galois. In the CONGEST model, there is a host machine for each vertex, and updates are performed by sending messages between machines along graph edges. Directed-APSP (Algorithm 3) and BC (Algorithm 5) map into operators in D-Galois, and the updates between host machines in the CONGEST model are mapped to a local update of the proxy label along the proxy's edges in the operators in shared-memory: no communication occurs among hosts.

The labels on each proxy v in D-Galois are the fields used in Algorithms 3 and 5: (1) the sorted list of (distance, source) pairs (d_{sv}, s), L_v, (2) shortest path counts from each source, σ_{sv}, and (3) dependency values, $\delta_{s\bullet}(v)$. These labels are updated by the computation operators as specified in the Algorithms. They are synchronized by calling the Gluon API at the beginning of each BSP round before computation in (1) Algorithm 3: with the reduction for L_v and σ_{sv} in lines 12-17 of Algorithm 3 and (2) Algorithm 5: with the addition for $\delta_{s\bullet}(v)$. Gluon transparently handles the communication required to reconcile the proxy labels.

4.3 Optimizations

Data Structures The CONGEST model does not account for local computation cost, but Min-Rounds BC in D-Galois must be efficient in its local computation. We leverage efficient data structures suited to MRBC's computations to improve its runtime. Instead of the labels described in Section 4.2, the labels on each proxy are (1) an array A_v of size k where k is the number of sources used and (2) a map M_v.

A_v is an unsorted dense array of structures containing d_{sv}, σ_{sv}, and $\delta_{s\bullet}(v)$ for every source s being used. These fields are grouped into a single structure for spatial locality: when one of these fields is accessed for v and s, another field will usually be accessed as well. Access to the structure of a particular source does not require any search ($O(1)$ access time).

To avoid searching this unsorted array for the d_{sv}, s, and σ_{sv} to be sent in a particular round (Algorithm 3), we maintain

a Boost flat map[1] M_v, that maps from current distances d_{sv} to a dense bitvector of size k that indicates which sources currently have that distance. The map allows iterating through lexicographically sorted pairs d_{sv} and s (like L_v). Moreover, it allows searching a pair in logarithmic time (dense bitvector access is $O(1)$ time). Additionally, for Algorithm 5, we do not need to explicitly maintain the round in which a message in Algorithm 3 is sent: we can derive the round in which the σ_{sv} is ready to be sent using d_{sv} in the map, the current round number, and the number of already sent dependencies.

Delayed Synchronization In the Gluon API, we can specify the vertices whose labels have been updated in the current round so that Gluon avoids sending non-updated data. This matches the $2 \cdot m \cdot k$ bound on the number of messages sent in the CONGEST model. Nevertheless, a label like d_{sv} of a vertex could be synchronized even if it is not the final value since it can be updated several times during the algorithm by different edges. To avoid this redundancy, we exploit proxies in D-Galois (unlike in CONGEST) to store updates locally until they are finalized. With the Gluon API, we specify the vertices whose labels must be synchronized in this round by using the properties of Algorithms 3 and 5 that dictate the round in which a label is finalized. This delayed synchronization reduces the number of messages and communication volume significantly.

Proxy Synchronization Rule for Min-Rounds BC Let vertex v have a proxy on host h in D-Galois. In round r of Algorithm 3, if $r = d_{sv} + \ell_v^r(d_{sv}, s)$ (for some source s) on h, then we synchronize only d_{sv} and σ_{sv} with the other proxies of v. Otherwise, we do not need to synchronize anything for v.

The correctness of this rule is readily established by induction on round number r. The base case trivially holds at each source s for d_{ss} (for all proxies for s). Assume inductively that this result holds until round $r-1$, and let Algorithm 3 send d_{sv} in Step 9 in round r. This means that v must have received the correct $d_{sv}-1$ value from a predecessor x in a shortest path from s to v by round $r-1$ in Algorithm 3. By the inductive hypothesis, this means that the correct shortest path distance d_{sx} would have been synchronized among all proxies for x. Hence, the host h that contains edge (x, v) will contain the value d_{sx} for source s at the proxy x_h, and this value will have been propagated to v_h in a local computation by round $r-1$. The correct d_{sv} value will be ready to be synchronized with other proxies in round r.

We optimize Algorithm 5 in a similar manner: every proxy knows exactly when it needs to synchronize its dependency value, σ_{sv}, due to Algorithm 5 dictating the round in which a dependency value is required to update neighbors.

[1]We have observed Boost flat map, which uses a sorted vector, to perform better than the C++ standard map (which uses a red-black tree) even with $O(k)$ insertion complexity due to improved locality of a sorted vector.

5 Experimental Evaluation

We evaluate four different BC algorithms. Min-Rounds BC (MRBC) is the algorithm introduced in this paper. Synchronous-Brandes BC (SBBC) is the Brandes BC algorithm [13] that uses level-by-level breadth first search to compute shortest paths. MRBC and SBBC are implemented in D-Galois [16], the state-of-the-art distributed-memory graph analytics system (source code is publicly available [1]). Asynchronous-Brandes BC (ABBC) [52] is an asynchronous BC implementation in the Lonestar benchmark suite [2] which uses shared-memory Galois [46]. Maximal-Frontier BC (MFBC) [53] is a sparse-matrix based BC algorithm implemented in Cyclops Tensor Framework (CTF) [54]. ABBC is asynchronous but does not have a distributed-memory implementation[2], whereas the others are bulk-synchronous distributed-memory implementations. We evaluate all algorithms using only unweighted graphs (note that ABBC and MFBC can also handle weighted graphs).

5.1 Experimental Setup

The experimental platform is the Stampede2 Skylake cluster [55] at the Texas Advanced Computing Center [3]. Each Skylake host has 48 2.1 GHz cores on 2 sockets (24 cores per socket) and 192GB RAM. The hosts are connected through Intel Omni-Path Architecture (peak bandwidth of 100 Gbps). MFBC uses 1 process per core (48 processes) on each host while the rest use 1 48-threaded process per host. We use up to 256 hosts. All code was compiled using gcc/g++ 7.1.0.

We use the unweighted graphs listed in Table 1. livejournal and friendster [40] are social networks; indochina04, gsh15 [9–11], and clueweb12 [51] are web-crawls; road-europe [24] is a road network; rmat24 and kron30 are random power-law graphs generated in the RMAT [14] and Kronecker [39] style, respectively. We classify the input graphs into *small* (livejournal, friendster, indochina04, rmat24, and road-europe) and *large* (kron30, gsh15, and clueweb12) based on their size (number of vertices and edges listed in Table 1). We evaluate all algorithms for the small graphs on 1 and 32 hosts. Since MFBC does not perform well as graphs increase in size and ABBC is limited to shared-memory, we only show results for MRBC and SBBC for the large graphs on 64, 128, and 256 hosts.

The BC of a vertex can be *approximated* [6] by summing the betweenness scores of that vertex for randomly sampled sources. ABBC, SBBC, and MRBC can compute BC using a random subset of sources. However, due to the limitation of comparing with MFBC, which only supports a sequence of contiguous sources as input, our experiments sample a random contiguous chunk of sources. Table 1 lists the number of sources sampled for each graph as well as the maximum finite

[2]It is not trivial to port ABBC to run on distributed-memory because it is not a vertex program. Moreover, we expect ABBC to be slower on distributed-memory as acquiring locks in a distributed setting is costly.

Table 1. Inputs and their properties, rounds, and load imbalance.

	livejournal	indochina04	rmat24	road-europe	friendster	kron30	gsh15	clueweb12
$\|V\|$	4.8M	7.4M	17M	174M	66M	1,073M	988M	978M
$\|E\|$	69M	194M	268M	348M	3,612M	17,091M	33,877M	42,574M
Max Out-degree	20,293	6,985	236,460	15	5,214	3.2M	32,114	7,447
Max In-degree	13,906	256,425	236,386	12	5,214	3.2M	59M	75M
# of Sources	4096	4096	4096	32	4096	4096	2048	256
Estimated Diameter	17	45	9	22541	25	9	103	501
SBBC Rounds	25.0	40.6	6.8	42,345.7	44.2	6.0	127.1	661.0
MRBC Rounds	2.7	3.3	1.4	1,410.8	3.5	1.0	4.4	17.0
SBBC Load Imbalance at Scale	3.33	3.52	1.95	1.29	1.60	1.12	1.46	3.70
MRBC Load Imbalance at Scale	2.25	2.18	1.06	1.30	1.39	1.17	1.74	3.05

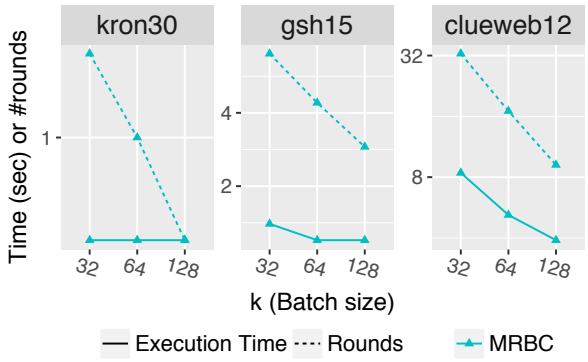


Figure 1. Execution time and number of rounds of MRBC for large graphs on 256 hosts with different k (batch sizes).

shortest path distance observed for those sources as the estimated diameter. We consider livejournal, rmat24, friendster, and kron30 to be *low-diameter* graphs (estimated diameter ≤ 25). Since the sources sampled are the same for all algorithms, the approximated BC values are the same. We present the mean execution time of three runs excluding graph loading, partitioning, and construction time. *All results are presented as an average per source*.

5.2 Configuration of different algorithms

We use double-precision floating point values for shortest path counts (otherwise, the results may be incorrect due to overflow); we modified MFBC to use double-precision as it uses single-precision by default. Configuration parameters of all algorithm implementations were tuned to optimize performance. We configured the chunk-size of the work-list in ABBC based on the input (64 for road-europe and 8 for the rest). To partition input graphs across hosts in SBBC and MRBC, we used the Cartesian vertex-cut [12, 16] partitioning policy, which performs well at scale [27].

ABBC and SBBC compute the betweenness scores of all vertices for one source at a time. MRBC and MFBC can compute the betweenness scores of all vertices for k sources simultaneously. MFBC performs best when k is the highest power-of-2 for which the graph fits in memory [53], and we choose k accordingly for each graph and number of hosts. On the other hand, it is not clear what k performs best for MRBC.

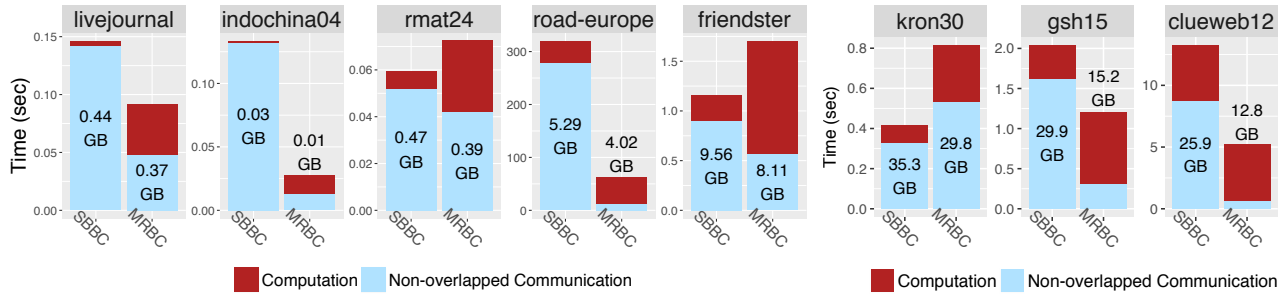
Figure 1 shows the execution time of MRBC for the large graphs on 256 hosts with different k, which we call the *batch size*. Increasing k is expected to reduce the number of rounds MRBC takes as parallelism is increased with more source in a batch; this round reduction is tied to the estimated diameter of the graph (Lemma 8). The speedup in execution time from batch size 32 to batch size 128 for kron30, gsh15, and clueweb12 is 1.0×, 1.2×, and 2.2×, respectively. For graphs with low diameter such as kron30, the reduction in rounds is minimal, and therefore, increasing batch size does not help as much and may even worsen performance due to the increased memory overhead and data structure access time. For graphs with larger diameter such as gsh15 and clueweb12, increasing batch size can improve runtime. The tradeoff between increasing parallelism and data structure access time (i.e., finding the best batch size for a graph) can be explored using a method such as autotuning; this is not the focus of this work. In the rest of this paper, we set the batch size k of MRBC to 32 and 64 for small and large graphs, respectively.

5.3 Comparison of different algorithms

Table 2 compares the execution time of all algorithms with the best-performing number of hosts. Note the execution times are averaged over the number of sources; therefore, small differences in per-source execution time can yield large differences in the execution time (in the order of hours or days) depending on the number of sources (corresponds to the approximation quality [6]). For *high-diameter* graphs like road-europe, ABBC substantially outperforms these algorithms because it is asynchronous whereas the others execute huge number of bulk-synchronous rounds with very little computation in each round. For the other graphs, however, ABBC either is slower than the others due to contention or runs out-of-memory because it is restricted to a single host. Both SBBC and MRBC outperform MFBC by significant margins,

Table 2. Execution time (sec) using the best-performing number of hosts (#hosts in parenthesis; "-" means out-of-memory).

Algorithm	Inputs					Algorithm	Large Inputs		
	livejournal	indochina04	rmat24	road-europe	friendster		kron30	gsh15	clueweb12
ABBC	0.41 (1)	-	21.72 (1)	**5.16 (1)**	-	SBBC	**0.42 (256)**	2.04 (256)	13.26 (256)
MFBC	0.10 (32)	0.29 (32)	0.20 (32)	-	4.41 (32)				
SBBC	**0.09 (1)**	0.08 (1)	**0.06 (32)**	320.46 (32)	**1.16 (32)**	MRBC	**0.82 (256)**	**1.21 (256)**	**5.18 (256)**
MRBC	**0.09 (32)**	**0.03 (32)**	0.07 (32)	62.37 (32)	1.71 (32)				



(a) Small graphs on 32 hosts. **(b)** Large graphs on 256 hosts.

Figure 2. Breakdown of execution time and communication volume (on each bar).

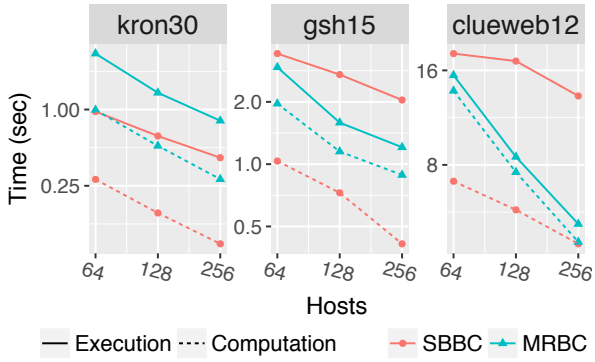


Figure 3. Strong scaling of execution time for large graphs.

which could be due to both algorithmic and implementation differences. *MRBC is 3.0× faster than MFBC on average*.

In the rest of this section, we focus on differences between SBBC and MRBC. Since SBBC and MRBC are implemented in the same system, performance differences between them are due to the algorithm. Table 2 shows that SBBC is faster for low-diameter graphs (estimated diameter ≤ 25 in Table 1) while MRBC is faster for the other graphs. Real world web-crawls like gsh15 and clueweb12 have non-trivial diameters (due to long tails), and we see that *MRBC is 1.7× and 2.6× faster than SBBC for gsh15 and clueweb12, respectively*. The larger the diameter, the better MRBC is.

Bulk-synchronous rounds The advantages of MRBC come mainly from the reduction in the number of rounds compared to SBBC. Table 1 shows the number of rounds executed in SBBC and MRBC for all inputs. *MRBC reduces the number of rounds executed over SBBC by 14.0× on average*.

Computation time We measure compute time on each host and denote the maximum across hosts as the computation time and the rest of the execution time as non-overlapped communication (and synchronization) time. Figure 2 shows the breakdown of execution time of SBBC and MRBC at scale into computation/communication time. For all inputs, the computation time of MRBC is higher than that of SBBC due to the overheads of maintaining additional data structures.

Communication time The non-overlapped communication time in Figure 2 includes waiting time at BSP barriers and data structure access time to (de)serialize messages. As MRBC maintains more complex data structures than SBBC does, access time and locality of MRBC can be worse. For instance, for kron30 at 256 hosts, (de)serialize time accounts for over ∼ 50% of communication time for MRBC, and (de)serialization for kron30 is ∼ 3.6× slower for MRBC than it is for SBBC. However, due to reduction in rounds, MRBC can reduce the wait time at barriers and the total communication volume across hosts (shown in Figure 2). The total number of proxies synchronized in SBBC and MRBC across all rounds are similar. The message size in MRBC is more because it identifies the source corresponding to the message (SBBC does not because it does one source at a time). However, Gluon [16] aggregates the messages of all proxies at the end of each round, compresses the metadata

that identifies the proxies, and exchanges one communication message between each pair of hosts. Therefore, the number of messages in MRBC is fewer than that of SBBC. MRBC synchronizes the same number of proxies in fewer rounds than SBBC; more proxies are synchronized in each round in MRBC, which leads to more compression of metadata and lower communication volume. Due to this, *MRBC reduces the communication time compared to SBBC by* 2.8× *on average*.

While MRBC reduces rounds for low-diameter graphs, the number of rounds executed in SBBC for these graphs is small enough that the rounds reduction may not yield net improvement due to increased computation time. For graphs with non-trivial diameter, the communication time reduction by MRBC outweighs its computation time overhead, yielding faster execution time than SBBC.

Load balance Load imbalance also affects performance in SBBC and MRBC. Graphs are irregular data structures, and nodes becomes active dynamically in each round, so it is difficult to statically partition the graph to obtain dynamic load balance. Table 1 shows estimates of load imbalance (the ratio of maximum computation time and mean computation time across hosts averaged across rounds). Reducing the number of rounds reduces the impact of load imbalance on the overall execution time as it decreases the wait time at synchronization barriers: this generally favors MRBC over SBBC since MRBC has fewer rounds. In graphs with low load imbalance (e.g., rmat24 or kron30), this effect may not be as pronounced.

Strong scaling For livejournal, indochina04, rmat24, and friendster, the speedups on 32 hosts over 1 host for MRBC are 3.1×, 6.3×, 7.8×, and 12.6×, respectively, while for SBBC, they are 0.6×, 0.6×, 2.6×, and 5.0×, respectively (performance for livejournal and indochina04 degrades due to communication overhead). Figure 3 shows the scaling of SBBC and MRBC from 64 to 256 hosts on the large graphs. MRBC scales better than SBBC as the benefits of reducing rounds grows with increase in the number of hosts: for these graphs, the mean self-relative speedup of MRBC and SBBC on 256 hosts over that on 64 hosts is 2.7× and 1.5×, respectively. Thus, *for graphs with non-trivial diameter, MRBC not only runs faster but also scales better than SBBC*.

6 Related Work

For unweighted undirected graphs, the first $O(n)$-round CONGEST APSP algorithms were given in [29, 48]. The constant factor in the number of rounds was improved to $n + O(D_u)$ in [38]. Lower bounds of $\Omega(n/\log n)$ for computing diameter and APSP were given in [23, 32].

For unweighted directed graphs, we do not know of published results that claim to compute exact APSP in $O(n)$ rounds. But as noted in Section 3.1, the $2n$-round version of APSP algorithm claimed for undirected graphs in [38] in fact works for directed graphs.

For Betweenness Centrality, an $O(n)$-round CONGEST algorithm for undirected unweighted graphs was given in [31], along with an $\Omega(\frac{n}{\log n} + D_u)$-round lower bound and a method to approximate an exponential number of shortest paths with log-size messages. An approximation algorithm for computing random walk BC in $O(n \log n)$ rounds in the CONGEST model was recently given in [30]. Distributed BC algorithms from a practical prospective are given in [59, 60]. No $O(n)$-round BC algorithm was known for directed graphs prior to our algorithm.

Most distributed-memory implementations [8, 16, 19, 20, 61] of BC are based on Brandes's algorithm [13]; many of these implementations do level-by-level traversals of the graph to efficiently calculate dependency values. Maximal-Frontier BC [53] is formulated using communication-efficient matrix operations. Min-Rounds BC outperforms these algorithms for many graphs in our evaluation. BC has also been implemented for shared-memory processors [7, 28, 43, 56, 57]. Dhulipala et al. [18] use a compressed data format for very large graphs to run the Brandes BC algorithm [13] on a shared-memory multicore machine with 1 TB of memory. Asynchronous-Brandes BC [52] is an *asynchronous* algorithm for BC that performs well on high-diameter graphs. Our results show that it outperforms all other BC algorithms on such graphs, but it does not perform as well on power-law graphs.

BC algorithms exist for the external memory setting where the graph is not fully loaded into memory [5] and for the streaming setting where the graph changes in structure over time [35]. These works are orthogonal to the distributed memory setting that Min-Rounds BC is designed for.

7 Conclusion

We presented Min-Rounds BC, a round- and message-efficient distributed algorithm in the CONGEST model for computing BC in an unweighted directed graph. Our algorithm translates into a highly efficient BC algorithm on the D-Galois distributed-memory graph analytics system and runs in at most $2 \cdot (k + D)$ rounds where k is the number of sources in a batch and D is the directed diameter. Experiments on a large production cluster show that MRBC is faster than other BC algorithms for non-trivial diameter graphs even though it may perform more computation than other algorithms. For real-world web-crawls on 256 hosts, MRBC is 2.1× faster than Brandes BC on average in our experiments.

Acknowledgments

This research was supported by NSF CCF grants 1320675, 1337217, 1337281, 1406355, 1618425, and 1725322 and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563. This work used XSEDE grant ACI-1548562 through allocation TG-CIE170005. We used the Stampede2 system at Texas Advanced Computing Center.

References

[1] [n. d.]. Galois System. http://iss.ices.utexas.edu/?p=projects/galois.

[2] 2018. The Lonestar Benchmark Suite. http://iss.ices.utexas.edu/?p=projects/galois/lonestar

[3] 2018. Texas Advanced Computing Center (TACC), The University of Texas at Austin. https://www.tacc.utexas.edu/

[4] U. Agarwal and V. Ramachandran. 2019. Distributed weighted all pairs shortest paths through pipelining. In *Proceedings of the 33rd IEEE International Parallel and Distributed Processing (IPDPS '19)*. To appear.

[5] L. Arge, M. T. Goodrich, and F. van Walderveen. 2013. Computing betweenness centrality in external memory. In *2013 IEEE International Conference on Big Data*. 368–375. https://doi.org/10.1109/BigData.2013.6691597

[6] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating Betweenness Centrality. In *Proceedings of the 5th International Workshop on Algorithms and Models for the Web-Graph (WAW '07)*. 124–137.

[7] D. A. Bader and K. Madduri. 2006. Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks. In *2006 International Conference on Parallel Processing (ICPP'06)*. 539–550. https://doi.org/10.1109/ICPP.2006.57

[8] Massimo Bernaschi, Giancarlo Carbone, and Flavio Vella. 2016. Scalable Betweenness Centrality on multi-GPU Systems. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 29–36. https://doi.org/10.1145/2903150.2903153

[9] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. BUbiNG: Massive Crawling for the Masses. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14 Companion)*. ACM, New York, NY, USA, 227–228. https://doi.org/10.1145/2567948.2577304

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A Multiresolution Coordinate-free Ordering for Compressing Social Networks. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*. ACM, New York, NY, USA, 587–596. https://doi.org/10.1145/1963405.1963488

[11] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[12] E. G. Boman, K. D. Devine, and S. Rajamanickam. 2013. Scalable matrix computations on large scale-free graphs using 2D graph partitioning. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2503210.2503293

[13] U. Brandes. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology* 25 (2001).

[14] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. [n. d.]. *R-MAT: A Recursive Model for Graph Mining*. 442–446. https://doi.org/10.1137/1.9781611972740.43 arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9781611972740.43

[15] T. Coffman, S. Greenblatt, and S. Marcus. 2004. Graph-based technologies for intelligence analysis. *Commun. ACM* 47 (2004). Issue 3.

[16] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A Communication-optimizing Substrate for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, New York, NY, USA, 752–768. https://doi.org/10.1145/3192366.3192404

[17] A. Del Sol, H. Fujihashi, and P. O'Meara. 2005. Topology of small-world networks of protein–protein complex structures. *Bioinformatics* (2005).

[18] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*. ACM, New York, NY, USA, 393–404. https://doi.org/10.1145/3210377.3210414

[19] Nicoletta Di Blas and Bianca Boretti. 2009. Interactive Storytelling in Pre-school: A Case-study. (2009), 44–51. https://doi.org/10.1145/1551788.1551797

[20] N. Edmonds, T. Hoefler, and A. Lumsdaine. 2010. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *HiPC*.

[21] Michael Elkin. 2006. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM J. Comput.* 36, 2 (2006), 433–456.

[22] L. C. Freeman. 1977. A set of measures of centrality based on betweenness. (1977).

[23] Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. 2012. Networks Cannot Compute Their Diameter in Sublinear Time. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '12)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1150–1162. http://dl.acm.org/citation.cfm?id=2095116.2095207

[24] Karlsruher Institut fur Technologie. 2014. OSM-Europe. https://i11www.iti.kit.edu/resources/roadgraphs.php

[25] Juan A. Garay, Shay Kutten, and David Peleg. 1998. A SubLinear Time Distributed Algorithm for Minimum-Weight Spanning Trees. *SIAM J. Comput.* 27, 1 (Feb. 1998), 302–316.

[26] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Andrew Lenharth, and Keshav Pingali. 2018. Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 249–264.

[27] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. 2018. A Study of Partitioning Policies for Graph Analytics on Large-scale Distributed Platforms *(PVLDB)*, Vol. 12. https://doi.org/10.14778/3297753.3297754

[28] Oded Green and David A. Bader. 2013. Faster Betweenness Centrality Based on Data Structure Experimentation. *Procedia Computer Science* 18 (2013), 399 – 408. https://doi.org/10.1016/j.procs.2013.05.203 2013 International Conference on Computational Science.

[29] Stephan Holzer and Roger Wattenhofer. 2012. Optimal Distributed All Pairs Shortest Paths and Applications. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing (PODC '12)*. ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/2332432.2332504

[30] Q. S. Hua, M. Ai, H. Jin, D. Yu, and X. Shi. 2017. Distributively Computing Random Walk Betweenness Centrality in Linear Time. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 764–774.

[31] Q. S. Hua, H. Fan, M. Ai, L. Qian, Y. Li, X. Shi, and H. Jin. 2016. Nearly Optimal Distributed Algorithm for Computing Betweenness Centrality. In *36th ICDCS*. 271–280.

[32] Qiang-Sheng Hua, Haoqiang Fan, Lixiang Qian, Ming Ai, Yangyang Li, Xuanhua Shi, and Hai Jin. 2016. Brief Announcement: A Tight Distributed Algorithm for All Pairs Shortest Paths and Applications. In *SPAA '16*. 439–441.

[33] H. Jeong, S. P. Mason, A. L. Barabási, and Z. N. Oltvai. 2001. Lethality and centrality in protein networks. *Nature* 411 (May 2001). http://dx.doi.org/10.1038/35075138

[34] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. 2010. A novel application of parallel betweenness centrality to

power grid contingency analysis. In *IPDPS*.

[35] N. Kourtellis, G. De Francisci Morales, and F. Bonchi. 2016. Scalable online betweenness centrality in evolving graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1580–1581. https://doi.org/10.1109/ICDE.2016.7498421

[36] V. Krebs. 2002. Mapping Networks of Terrorist Cells. *Connections* (2002).

[37] Christoph Lenzen and Boaz Patt-Shamir. 2013. Fast Routing Table Construction Using Small Messages: Extended Abstract. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing (STOC '13)*. ACM, New York, NY, USA, 381–390. https://doi.org/10.1145/2488608.2488656

[38] Christoph Lenzen and David Peleg. 2013. Efficient Distributed Source Detection with Limited Bandwidth. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 375–382. https://doi.org/10.1145/2484239.2484262

[39] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.* 11 (March 2010), 985–1042. http://dl.acm.org/citation.cfm?id=1756006.1756039

[40] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[41] F. Liljeros, C.R. Edling, L. Amaral, H.E. Stanley, and Y. Åberg. 2001. The web of human sexual contacts. *Nature* 411 (2001). https://doi.org/10.1038/35082140

[42] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[43] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel G. Chavarría-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS*. IEEE, 1–8.

[44] Fragkiskos D. Malliaros and Michalis Vazirgiannis. 2013. Clustering and community detection in directed networks: A survey. *Physics Reports* 533, 4 (2013), 95 – 142.

[45] Danupon Nanongkai. 2014. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing (STOC '14)*. ACM, New York, NY, USA, 565–573. https://doi.org/10.1145/2591796.2591850

[46] Donald Nguyen and Keshav Pingali. 2011. Synthesizing concurrent schedulers for irregular algorithms. In *Proc. Intl Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. 333–344. https://doi.org/10.1145/1950365.1950404

[47] David Peleg. 2000. *Distributed Computing: A Locality-sensitive Approach*. SIAM, Philadelphia, PA, USA.

[48] David Peleg, Liam Roditty, and Elad Tal. 2012. Distributed Algorithms for Network Diameter and Girth. In *ICALP'12*. 660–672.

[49] David Peleg and Vitaly Rubinovich. 2000. A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction. *SIAM J. Comput.* 30, 5 (May 2000), 1427–1442.

[50] Matteo Pontecorvi and Vijaya Ramachandran. 2018. Distributed Algorithms for Directed Betweenness Centrality and All Pairs Shortest Paths. (2018). http://arxiv.org/abs/1805.08124

[51] The Lemur Project. 2013. The ClueWeb12 Dataset. http://lemurproject.org/clueweb12/

[52] Dimitrios Prountzos and Keshav Pingali. 2013. Betweenness Centrality: Algorithms and Implementations. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/2442516.2442521

[53] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling Betweenness Centrality Using Communication-efficient Sparse Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 47, 14 pages. https://doi.org/10.1145/3126908.3126971

[54] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 813–824. https://doi.org/10.1109/IPDPS.2013.112

[55] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S. Mehringer, Eric Wernert, H. Tufo, D. Panda, and P. Teller. 2017. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC17)*. ACM, New York, NY, USA, Article 15, 8 pages. https://doi.org/10.1145/3093338.3093385

[56] G. Tan, V. C. Sreedhar, and G. R. Gao. 2011. Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *J. Supercomput.* 56 (2011). Issue 1.

[57] Guangming Tan, Dengbiao Tu, and Ninghui Sun. 2009. A Parallel Algorithm for Computing Betweenness Centrality. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP '09)*. IEEE Computer Society, Washington, DC, USA, 340–347. https://doi.org/10.1109/ICPP.2009.53

[58] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181

[59] W. Wang and C. Y. Tang. 2013. Distributed computation of node and edge betweenness on tree graphs. In *52nd Conf. on Decis. and Contr.* 43–48.

[60] K. You, R. Tempo, and L. Qiu. 2017. Distributed Algorithms for Computation of Centrality Measures in Complex Networks. *Trans. on Autom. Contr.* 62, 5 (2017), 2080–2094.

[61] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 301–316. http://dl.acm.org/citation.cfm?id=3026877.3026901

A Artifact Appendix

A.1 Abstract

We provide source code to SBBC and MRBC in this paper and scripts to run experiments from the paper. This artifact must be run on the Stampede2 supercomputer. This artifact supports the paper by making it possible to replicate the figures and numbers in this paper, and it can be validated by comparing the figures and results that this artifact's scripts generate with the data from the paper. We also provide CSVs that can be used to generated the exact figures in this paper.

Users can reproduce the figures in this paper and the numbers of SBBC and MRBC in Table 2, except those on 1 host.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Min Rounds Betweenness Centrality, Synchronous Brandes Betweenness Centrality
- **Compilation:** cmake, g++ 7.1.0, boost 1.64
- **Data set:** Public web crawls, randomly generated power-law graphs
- **Hardware:** Stampede2 supercomputer
- **Execution:** Scheduling on Stampede2's job queue
- **Metrics:** Execution time, communication volume, breakdown of computation/communication time
- **Output:** Figures and CSV files (with runtime statistics)
- **Experiments:** Use provided scripts in the artifact to build, schedule jobs, and generate figures
- **How much time is needed to prepare workflow (approximately)?:** Assuming access to Stampede2, 10 minutes
- **How much time is needed to complete experiments (approximately)?:** Roughly a week
- **Publicly available?:** Yes [1]

A.3 Description

A.3.1 How delivered

Via Zenodo: https://doi.org/10.5281/zenodo.2399798

A.3.2 Hardware dependencies

This artifact uses the Stampede2 supercomputer.

A.3.3 Software dependencies

The R software environment is required to generate CSVs and graphs. We provide usable R executables on Stampede2 that our scripts use. All other dependencies are handled by our scripts assuming you are running on Stampede2.

A.3.4 Data sets

The data sets are made available on Stampede2. They are too large (roughly 1.5 TB total) to package with the artifact.

A.4 Installation

These instructions assume that you have Stampede2 access.

Once the artifact is downloaded, move it onto Stampede2 into your personal `$WORK` directory. This can be accessed using the command `cd $WORK`. Extract the artifact using `tar -xf`. The building of the executables will be handled by the same script that runs the experiments described in the below sections.

Before running any experiments, to set up notifications for when the scheduled experiments start and end, change the following line located in the following files in the execute_scripts directory:

```
test_rmat15_lvl.sbatch
test_rmat15_mr.sbatch
skx_run_stampede.template.sbatch
```

```
#SBATCH --mail-user=<insert email here>
```

Additionally, in the same files above, change the following line to whatever allocation you are using on Stampede2:

```
#SBATCH -A Galois
```

Replace "Galois" with your Stampede2 allocation name.

A.5 Experiment workflow

To run experiments, users run provided scripts that will compile our code and schedule runs of the executable on Stampede2 via sbatch. To run jobs on Stampede2, one must put them on a queue and wait until they are scheduled (our scripts handle this automatically). Once these experiments finish, the statistics collected by the Galois runtime will be output. We provide scripts that will compile these results into CSVs using R, and from these CSVs, we have included scripts that will use R to create the figures that we used in the paper. Details are located in Section A.6.

The experimental scripts can be modified to test different runtime parameters by changing the Stampede2 scheduling scripts in the `execute_scripts` directory; details are described in Section A.7.

If a user does not have Stampede2 access, if they have R installed on their local machine, the artifact includes CSVs that can be used to generate the exact figures used in this paper. Details are in Section A.8.

A.6 Evaluation and expected result

We provide two executables for reviewers to evaluate: bc_level (SBBC) and bc_mr (MRBC). The source code for both can be found in `GaloisCpp/dist_apps` in the artifact directory.

Users are expected to *reproduce* the results in this paper, specifically generating all figures in the paper and and similar average execution times per source (Table 2). There may be slight variation of roughly 5-10% (possibly more when the runtime numbers are in the millisecond scale) from the numbers reported in the paper.

There are 3 types of scripts that you must run: run scripts, compile scripts, and plot scripts. For the `run` and `compile` scripts, there are 7 different kinds: `small`, `large64`, `large128`, `large256` (`lvl` and `mr`), `b32`, and `b128`. The run scripts will compile and schedule the runs on the Stampede2 queue, and the compile scripts will compile the corresponding statistics from the run (*only once they finish*) into CSVs for easier inspection. These correspond to 7 different groups of experiments. Descriptions of what each group of scripts is below:

small Runs the SBBC/MRBC experiments for the smaller graphs in the paper.

large64 Runs the SBBC/MRBC experiments for kron30, gsh15, clueweb12 at 64 nodes.

large128 Runs the SBBC/MRBC experiments for kron30, gsh15, clueweb12 at 128 nodes.

Note that for the 3 groups below, only THREE jobs may be on queue at once due to the use of 256 nodes. Each script will schedule 3 jobs, so you will not be able to run these in parallel.

large256(mr, lvl) Runs the SBBC/MRBC experiments for kron30, gsh15, clueweb12 at 256 nodes.

b32 Runs the MRBC experiments for kron30, gsh15, clueweb12 at 256 nodes with a batch size of 32.

b128 Runs the MRBC experiments for kron30, gsh15, clueweb12 at 256 nodes with a batch size of 128.

Workflow will be as follows:

```
./run_small_experiments.sh
<wait for scheduled experiments to finish>
./compile_small_experiments.sh
./run_large256_experiments.sh
<wait for scheduled experiments to finish>
./compile_large256_experiments.sh
.... and so on, for the 5 remaining groups.
./plot_fig1.sh
.... and so on for remaining figures
```

There are 4 types of output reviewers can inspect once all scripts for an experiment group and appropriate plot scripts have been run.

The first is raw Galois runtime output to be found in `skx_outputs`. These contain output from the executables printed during runtime: most notably, each output file has sanity check output used to verify correctness across runs (e.g. the maximum betweenness centrality value among all nodes, the sum of all centrality values, etc.).

The second is Galois statistics output found in `skx_results`. which detail statistics collected during runtime such as execution time, graph construction time, communication time, etc. Notably, `Timer_0` in these files contain the execution time of the programs.

The third is CSVs generated by R from the Galois statistics. This contains summarized statistics derived from the Galois file such as total communication volume in an easier to process format. Notably, it will contain the average time per source in the final column: this can be used to compare some results with Table 2. Each `compile` script will add to a particular CSV file, as listed below:

bc_results_small.csv `small_results`

bc_results_large.csv `large64_results`, `large128_results`, `large256_results`

bc_results_b32.csv `b32_results`

bc_results_b128.csv `b128_results`

The last is the figures to be found in `figs` directory. These correspond to all the figures in the paper once all plot scripts have been run. Dependencies for each figure are listed below:

Figure 1 `run` and `compile` for `large256`, `b32`, and `b128` should be completed successfully.

Figure 2a `run` and `compile` for `small_experiments` should be completed successfully.

Figure 2b `run` and `compile` for `large256` should be completed successfully.

Figure 3 `run` and `compile` for `large64`, `large128`, and `large256` should be completed successfully.

In the event of job failure (e.g., runtime crash), it is possible to schedule specific jobs and algorithms again by changing the top-level run scripts and the execute scripts: see Section A.7 for details.

A.7 Experiment customization

The top-level `run` scripts call into individual scripts in `execute_scripts`: you can change which individual scripts are called by commenting them out.

Users may experiment with a number of runtime settings by altering the files located in `execute_scripts` under the `clueweb`, `gsh`, `kron`, and `small` directories.

Algorithms to Run You can specify which algorithms to run (bc_level, bc_mr, or both) by changing the strings in `EXECS`.

Number of Hosts/Time Limit You can change the number of hosts to run the distributed algorithms on by changing the number before the comma in the `SET` variable. For example, 32,03:00:00 means to run on 32 hosts for a maximum time of 3 hours. Multiple jobs can be scheduled by specifying many host/time pairs in `SET` (see the scripts for examples).

Number of Sources You can change the number of sources to calculate betweenness centrality for by changing the number in the `NUMSOURCES` variable. As our scripts choose the sources to run from a text file, *it is not possible to change the number of sources in a script to greater than the number that the script has by default.*

Batch Size for MRBC Batch size can be changed for MRBC by changing the number in the `BATCHSIZE` variable.

Threads You can change the number of threads used on each host by changing the `threads` variable in `skx_run_stampede_all.sh`. (Default is the maximum number of threads on Skylake without hyperthreading.)

A.8 Notes

The directory `ppopp2019csv` contains CSVs that can be used to generate the same figures found in the paper. Simply copy the CSVs to the same directory as the `plot` scripts, and the plot scripts will be able to detect them. If NOT running on Stampede2, make sure to go into the `plot` scripts and change the path to Rscript to the one on your system (it is hardcoded to use a public installation on Stampede2). You may also need to install the R packages dependencies, which include the following:

`ggplot2, gtable, grid, gridExtra, plyr, reshape2`

We provide test scripts prefixed with `test` in the top-level directory that will run a very small job and output results.

We have seen non-deterministic failure of scheduled jobs periodically: this is usually resolved by rescheduling the job.

A.9 Methodology

Submission, reviewing and badging methodology:

http://cTuning.org/ae/submission-20180713.html
http://cTuning.org/ae/reviewing-20180713.html
https://www.acm.org/publications/policies/artifact-review-badging