ParTBC: Faster Estimation of Top-k Betweenness Centrality Vertices on GPU

SOMESH SINGH, Indian Institute of Technology Madras
TEJAS SHAH, Microsoft
RUPESH NASRE, Indian Institute of Technology Madras

Betweenness centrality (BC) is a popular centrality measure, based on shortest paths, used to quantify the importance of vertices in networks. It is used in a wide array of applications including social network analysis, community detection, clustering, biological network analysis, and several others. The state-of-the-art Brandes' algorithm for computing BC has time complexities of $O(|V||E|)$ and $O(|V||E| + |V|^2 \log |V|)$ for unweighted and weighted graphs, respectively. Brandes' algorithm has been successfully parallelized on multicore and manycore platforms. However, the computation of vertex BC continues to be time-consuming for large real-world graphs. Often, in practical applications, it suffices to identify the most important vertices in a network; that is, those having the highest BC values. Such applications demand only the top vertices in the network as per their BC values but do not demand their actual BC values. In such scenarios, not only is computing the BC of *all* the vertices unnecessary but also *exact* BC values need not be computed. In this work, we attempt to marry controlled approximations with parallelization to *estimate* the k-highest BC vertices faster, without having to compute the exact BC scores of the vertices. We present a host of techniques to determine the top-k vertices *faster*, with a *small* inaccuracy, by computing *approximate* BC scores of the vertices. Aiding our techniques is a novel vertex-renumbering scheme to make the graph layout more *structured*, which results in faster execution of parallel Brandes' algorithm on GPU. Our experimental results, on a suite of real-world and synthetic graphs, show that our best performing technique computes the top-k vertices with an average speedup of 2.5× compared to the exact parallel Brandes' algorithm on GPU, with an error of less than 6%. Our techniques also exhibit high precision and recall, both in excess of 94%.

CCS Concepts: • **Parallel algorithms**; • **Mathematics of computing** → *Graph algorithms*; **Approximation**;

Additional Key Words and Phrases: Betweenness centrality, top-k, Brandes' algorithm, graph reordering, approximate computing, GPU

ACM Reference format:
Somesh Singh, Tejas Shah, and Rupesh Nasre. 2021. ParTBC: Faster Estimation of Top-k Betweenness Centrality Vertices on GPU. *ACM Trans. Des. Autom. Electron. Syst.* 27, 2, Article 12 (November 2021), 25 pages.
https://doi.org/10.1145/3486613

Tejas Shah work done while at Indian Institute of Technology Madras, India.

Authors' addresses: S. Singh and R. Nasre, Indian Institute of Technology Madras, Sardar Patel Road, Chennai 600 036, Tamil Nadu, India; emails: {ssomesh, rupesh}@cse.iitm.ac.in; T. Shah, Microsoft Corporation India Pvt. Ltd., Microsoft Prestige Ferns Galaxy, Bellandur Gate Road, Bellandur, Bengaluru 560 103, Karnataka, India; email: tedshah@microsoft.com.

1084-4309/2021/11-ART12 $15.00
https://doi.org/10.1145/3486613

1 INTRODUCTION

Betweenness centrality (BC) is a crucial centrality metric in graphs and networks that measures the significance of a vertex. BC(n) is calculated using the number of shortest paths in the graph passing through vertex, n. It is used in a multitude of applications such as detecting communities in social and biological networks [7], targeted advertising [12], analysis of disease spreading [16], and identifying criminal networks [5], and many more. The state-of-the-art Brandes' algorithm [4] computes the exact BC values for all vertices in a graph $G = (V, E)$ in time $O(|V||E|)$ for unweighted graphs, and time $O(|V||E| + |V|^2 \log |V|)$ for graphs having positive edge-weights. As suggested by its complexity, computation of BC is quite time-consuming even on graphs of moderate sizes, having hundreds of thousands of vertices and edges. For example, a single-threaded execution of Brandes' algorithm takes several hours to terminate on an undirected graph *loc-Gowalla* (having ~196,600 vertices and ~950,300 edges).

To make BC computations scalable, Brandes' algorithm has been successfully parallelized on multi-core CPUs, many-core GPUs, and distributed systems [10, 19, 20, 25, 31]. Yet, the cost of BC computation is excessive on modern networks with millions of vertices and tens of millions of edges. For example, the exact vertex-BC computation on the undirected graph *liveJournal* (having ~4.8M vertices and ~69M edges) using a parallel implementation of Brandes' algorithm on a GPU takes several days to complete. Moreover, often applications are interested in the relative ranking of the vertices according to their BC scores, rather than their actual BC values. In addition, several applications demand identifying vertices with highest BC values. Hence, an estimate of the top-k BC vertices is sufficiently informative.

In this work, we present ParTBC, a host of novel techniques for speeding up the estimation of top-k vertices with highest BC in a graph, using approximate computing in conjunction with parallelization. We propose to compute approximate BC values of vertices, such that the relative ordering of the vertices is maintained. The contributions of this article are as follows.

- To the best of our knowledge, ParTBC is the first system that combines parallelization on GPU and approximate computing to estimate the top-k BC vertices in a graph.
- Our proposals in ParTBC restrict computation of shortest paths from only a fraction of the vertices in parallel Brandes' algorithm based on an online stopping criterion that uses *tunable knobs*. The chosen source vertices impart *sufficient* contribution to the BC of the vertices early to enable quicker identification of top-k BC vertices, while achieving the desired accuracy.
- We present a novel graph reordering scheme to make the graph layout more *structured* to enable efficient coalesced access of data in parallel Brandes' algorithm on GPU, improving performance. The modified graph-layout is also beneficial to the vertex-centric parallel implementations of other graph algorithms, such as, single-source-shortest-path computation, pagerank computation, minimum-spanning-tree, and strongly-connected-component.
- We qualitatively as well as quantitatively assess the effect of our proposals. We observe that a combination of techniques performs well consistently. Using a suite of seven graphs of varying characteristics, we illustrate the effectiveness of ParTBC. Our experiments show that on an average, ParTBC reduces the computation time by 2.5× with mean inaccuracy in the ballpark of 6%. Further, ParTBC techniques have high precision and recall in excess of 94%.

2 PROBLEM STATEMENT AND PRELIMINARIES

Problem Statement. Given an undirected, unweighted graph $G(V, E)$ and a positive integer $k \leq |V|$, find a set of k vertices, S_k, where $S_k \subseteq V$ and S_k contains vertices having the highest BC values in G. In this work, we determine the set S_k faster with a small error in set membership.

ALGORITHM 1: Brandes' Algorithm

```
Input: An undirected, unweighted graph $G(V, E)$
Output: Vertex betweenness centrality
1  $bc[v] = 0 \quad \forall v \in V$                    //initialization
2  foreach $s \in V$ do
3      //Forward Pass: form BFS DAG $D$
4      forall the $v : Node \in G$ do
5          compute $\sigma_{sv}$
6          compute $pred(s, v)$
7      Let $D$ be the DAG formed by the forward pass
8      //Backward Pass: backward traverse DAG $D$
9      forall the $v : Node \in D$ do
10         compute $\delta_s(v)$
11         $bc(v)$ += $\delta_s(v)$
12     //Reset graph attributes
```

Betweenness Centrality. Consider that communication among vertices in a graph always progresses along the shortest paths. Then, the more the number of shortest paths that go through a particular vertex, the more important is the vertex. This notion of importance is captured by *betweenness centrality*.

Brandes' algorithm [4] is the fastest known algorithm for computing the betweenness centrality scores of the vertices in a graph. The complexity of Brandes' algorithm, for an unweighted graph, is $O(nm)$, where n is the number of vertices in the graph and m is the number of edges. Brandes' algorithm is presented in Algorithm 1.

The *dependency* of a vertex v w.r.t. a given source vertex s is $\delta_s(v)$. It is computed using the following recurrence:

$$\delta_s(v) = \sum_{w | v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)). \tag{1}$$

Here, σ_{sv} is the number of shortest paths from s to v, and $pred(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w (computed using the forward pass at Line 4). A vertex's $pred$ list is bounded by its degree. $pred$ lists of all the vertices together induce a **directed acyclic graph (DAG)** D over the graph G. BC of each vertex can then be computed as a summation over all the sources (computed using the backward pass at Line 9):

$$bc(v) = \sum_{s \neq v \in V} \delta_s(v). \tag{2}$$

PROPERTY 1. *In Brandes' algorithm BC of a vertex does not change in the iteration in which it is the source.*

PROOF. For each source, s, we compute the BFS DAG rooted at s. Now, s will always lie at one end of the shortest paths to all other vertices, from s. So, s cannot lie on the shortest path between any two other vertices, as all the edges have unit weight. Thus, in the iteration in which the vertex is a source, its BC does not change. The same is captured in Equation (2). □

OBSERVATION 1. *Our investigation reveals that high BC vertices are usually either (i) the high-degree vertices, or (ii) those low-degree vertices that lie on the paths connecting two or more large well-connected clusters.*

Justification. High-degree vertices are connected to a large number of vertices and thus lie on a large number of point-to-point paths. Consequently, these lie on a large number of point-to-point shortest paths. Further, those low-degree vertices that connect large clusters in a graph, lie on the shortest paths between the vertices lying in separate clusters. Thus, such vertices too exhibit high BC.

3 RELATED WORK

We discuss the prominent relevant prior works in the realm of parallel BC computation and approximate BC computation for computing top-k BC vertices. We divide the past works into (1) exact parallel BC computation, (2) approximate BC computation, and (3) top-k BC vertex computation.

Exact Parallel BC Computation. Madduri et al. [19] propose an efficient parallel implementation for computing vertex BC on shared memory multicore architectures. They improve the algorithm to use successors instead of predecessors in the computation of the DAG, which produces a more efficient, locality-friendly algorithm. Sariyuce et al. [27], and McLaughlin and Bader [20] present efficient parallel implementations for BC computation on GPUs and heterogeneous architectures. Prountzos and Pingali [25] propose a scalable asynchronous parallel algorithm for BC that is able to extract massive parallelism. Harshvardhan et al. [9] propose k-level asynchronous paradigm for parallel graph processing. It improves the performance of traditionally level-synchronous **Breadth-first-search (BFS)** traversal by reducing synchronization, which in turn improves the performance of other algorithms employing BFS including betweenness centrality computation. Solomonik et al. [31] propose a succinct parallel BC algorithm based on novel sparse matrix multiplication routines with reduced communication. Hoang et al. [10] propose a round-efficient distributed BC algorithm. Their proposal reduces the number of rounds by 14× and achieves a mean speedup of 2.1× over Brandes' algorithm on 256 hosts.

Approximate BC Computation. A survey of various approximate computing strategies is presented by Mittal [21], including, *precision scaling*, *loop perforation*, *memoization*, *selective memory accesses*, *data sampling*, *voltage scaling*, *inexact reads/writes*, *lossy compression* and *using universal function approximators* in various domains for improving performance and reducing the energy requirements in exchange for acceptable loss in output quality. In one of the first works on approximating vertex BC, Bader et al. [1] propose an adaptive sampling-based approach that reduces the number of single-source shortest path computations for vertices with high BC. Geisberger et al. [6] propose a framework for unbiased approximation of the BC values and get a good approximation to the BC values of the unimportant vertices too. Mostafa [8] proposes a generic randomized framework for unbiased approximation of vertex BC to achieve high efficiency and accuracy. Singh and Nasre [28] propose techniques for approximate graph processing on GPU. Their proposed techniques: reduced execution, partial graph processing and lossy graph compression, are effective in computing approximate vertex BC faster in exchange of small inaccuracy. Singh and Nasre [29, 30] present GPU-specific techniques, aided by approximate computing, to improve memory coalescing, and reduce memory latency and thread divergence for computing approximate BC faster on GPU.

Top-k BC Vertex Computation. Lee and Chung [14] propose an efficient algorithm to determine the exact k-highest BC vertices faster by using a block-cut tree of the graph and finding the exact BC of the vertices in each *smaller* biconnected component using Brandes' algorithm. Riondato and Upfal [26] propose progressive sampling schemes based on Rademacher averages to approximate the BC values and extend their argument to compute an approximation of top-k vertices with probabilistic guarantees. Mumtaz and Wang [22] propose an approximate algorithm for BC maximization problem. They devise an estimation technique based on progressive sampling with early stopping conditions to get better accuracy.

Our approach for finding the top-k vertices entails ordering the source vertices in Brandes' algorithm to enable us in estimating the top-k BC vertices, which is different from the above approaches. Further, our work applies these approximate techniques in the context of GPUs.

ALGORITHM 2: Approximate top-k computation

```
Input: An undirected, unweighted graph $G(V, E)$
Input: $k$
Input: desired accuracy ($\leq 100\%$)
Output: top-$k$ betweenness centrality vertices
$bc[v] = 0 \quad \forall v \in V$                                  //initialization
//Phase-I
$G'(V, E)$ = graphReordering($G$)  //vertex renumbering
                                   //$G' \cong G$
//Phase-II
nextIter = $true$
while nextIter do
    nextIter = $false$
    s = getSource()                                            //pick source vertex
    //Forward Pass: form BFS DAG $D$
    forall the $v : Node \in G'$ do
        compute $\sigma_{sv}$
        compute $pred(s, v)$
    Let $D$ be the DAG formed by the forward pass
    //Backward Pass: backward traverse DAG $D$
    forall the $v : Node \in D$ do
        compute $\delta_s(v)$
        $bc(v)$ += $\delta_s(v)$
    //Reset graph attributes
    if stopping criteria not met then
        nextIter = $true$;
```

4 PARTBC'S APPROACH

In this section, we present ParTBC's overall approach towards speeding up the computation of top-k BC vertices in a graph. Section 4.1 discusses parallel Brandes' algorithm for computing vertex BC. Section 4.2 briefly describes the in-memory data layout and the parallel implementation of Brandes' algorithm, that we use in this work. In Section 4.3, we discuss a novel graph reordering scheme to improve data locality during parallel execution of Brandes' algorithm.

Algorithm 2 outlines the approach adopted in ParTBC. The computation proceeds in two phases. Phase-I performs graph reordering by renumbering the vertices of the graph to bring together in memory the data of those vertices that are likely to be accessed in tandem in parallel Brandes' algorithm on GPU. The reordered graph is the input to Phase-II. In Phase-II, BC computation happens in parallel and the source vertices are picked (Line 9) using the techniques described in Section 5. The algorithm terminates when the stopping condition is satisfied, which is calculated online based on the desired accuracy in the set of top-k vertices.

4.1 Parallelization Strategy

Brandes' algorithm has been shown to be parallelized mainly in two ways: outer parallel and inner parallel [2, 11, 25]. In outer parallel, multiple source vertices are processed in parallel (line 2 of

```
u = blockIdx.x * blockDim.x + threadIdx.x;
if (u >= G.numNodes) return;
if (level[u] == hops_from_source) { // level−synchronous
    end = G.offset[u + 1];
    for (i = G.offset[u]; i < end; ++i) {
        v = G.edges[i];
        if (level[v] == −1) {
            level[v] = hops_from_source + 1;
            takeNextIter = true;
        }
        if(level[v] == hops_from_source + 1)
            atomicAdd(&sigma[v], sigma[u]);
}}
```

(a) Topology-driven Kernel

```
tid = blockIdx.x * blockDim.x + threadIdx.x;
if(tid >= worklist_size) return;
u = in_worklist[tid]; // node processed by thread tid
int hops_from_source = level[u];
end = G.offset[u + 1];
for (i = G.offset[u]; i < end; ++i) {
  v = G.edges[i];
  if (level[v] == −1) {
    level[v] = hops_from_source + 1;
    *takeNextIter = true;
    if (atomicCAS(&flag[v], 0, 1) == 0) {
      index = atomicAdd(activeSize, 1);
      out_worklist[index] = v;
  }}
  if (level[v] == hops_from_source + 1)
    atomicAdd(&sigma[v], sigma[u]);
}
```

(b) Data-driven Kernel

Fig. 1. Topology-driven and Data-driven implementations of the forward pass of Brandes' algorithm.

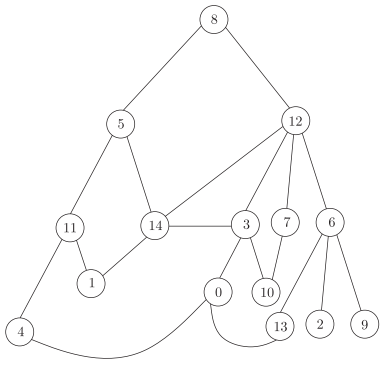
Algorithm 1), but the forward and the backward passes are executed sequentially by each thread. Thus, the contribution of each source to BC values of other vertices can be computed by the thread assigned to that source. The final computation of $bc(v)$ involves a *reduction* of the contribution of each of the sources. In this scheme, every outer loop iteration requires its own storage, leading to a substantial space overhead of $O(n^2)$ [25].

In inner parallel scheme, however, each source is processed sequentially, but each of the computation steps (lines 4, 9 in Algorithm 1) for a single source are executed in parallel. A crucial advantage of this approach is its space-efficiency, as DAG (and other transient data) corresponding to only one source need to be maintained at a time. Therefore, such an approach can be used for large graphs [25].

4.2 Graph Layout

We use the popular **Compressed Sparse Row (CSR)** storage format to represent the graph. CSR representation stores only the non-zero elements of the adjacency matrix, i.e., the edges of the graph. Figure 2 shows the CSR representation of the graph G. The CSR format uses two arrays to represent the graph: *offset* array and *edges* array. The *offset* array is sorted by vertex-ID and stores each vertex's starting offset into the *edges* array. The *edges* array stores the neighbors of the vertices contiguously, that is, the neighbors of vertex 0, followed by neighbors of vertex 1 and so on.

Further, we use the vertex-centric approach, wherein a thread is assigned to a vertex. Our parallel implementation of BC uses three kernels—one for the forward pass (Figure 1) and two as part of the backward pass. The two backward pass kernels compute δ values and accumulate BC values. We employ both *topology-driven* (Figure 1(a)) and *data-driven* implementations (Figure 1(b)) of



edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
3	4	13	11	14	6	0	10	12	14	0	11	8	11	14	2	9	12	13	10	12	5	12	6	3	7	1	4	5	3	6	7	8	14	0	6	1	3	5	12

offset

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	3	5	6	10	12	15	19	21	23	24	26	29	34	36	40

node attributes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 2. Original graph G and its CSR representation.

the forward pass kernel. In topology-driven implementations [24], all vertices are assumed to be active (i.e., required to be processed) at every step. So, in every iteration, all vertices are processed even if there is no useful work to do at some vertices. In our parallel implementation of Brandes' algorithm, this approach is useful when a large number of vertices are active at a level of the breadth-first-traversal, in the forward pass. Work inefficiency in this approach is counterbalanced by the large number of GPU threads. In contrast, in data-driven implementations [24], at a step, only those vertices are processed at which there is work to do. The data-driven approach maintains a worklist of active vertices at any time. This approach is work-efficient. This is helpful in our parallel implementation of Brandes' algorithm in cases when only a few vertices are active at a level of the breadth-first-traversal, in the forward pass. One approach may be better than the other depending on the structure and connectivity of the graph. Our implementation automatically chooses one of the two implementations (data-driven or topology driven) based on the *skewness* of the vertex degree distribution. This enables us to have a fast parallel execution of Brandes' algorithm for graphs with varying characteristics. For graphs with skewed degree distribution (|coeff. of skewness| >0.05), such as social networks, topology-driven implementation is chosen, while data-driven implementation is used for graphs with uniform degree (|coeff. of skewness| ≤0.05), such as road networks.

4.3 Improved Graph Layout

A natural way to compute top-k BC vertices faster is to improve performance of the exact parallel implementation of Brandes' algorithm. With this motivation, we propose a scheme to modify the graph layout to make it more *structured* to make it amenable for GPU-based processing. We intend to improve the memory coalescing and better utilize GPU's high memory bandwidth.

The forward-pass of Brandes' algorithm involves breadth-first-search (BFS) traversal of the graph from a designated source vertex in every iteration. In our parallelization strategy of the forward-pass (Figure 1), we process the vertices in a level-synchronous fashion and the thread assigned to a vertex updates the attributes of its neighbors. Reordering of vertices is shown to be effective in improving the spatial locality of vertices by assigning consecutive IDs to those that are likely to be accessed in tandem [3, 17, 23]. To improve vertex-centric processing, ParTBC proposes a novel vertex-renumbering scheme to modify the graph layout such that the connected vertices and their data are together for GPU-based processing.

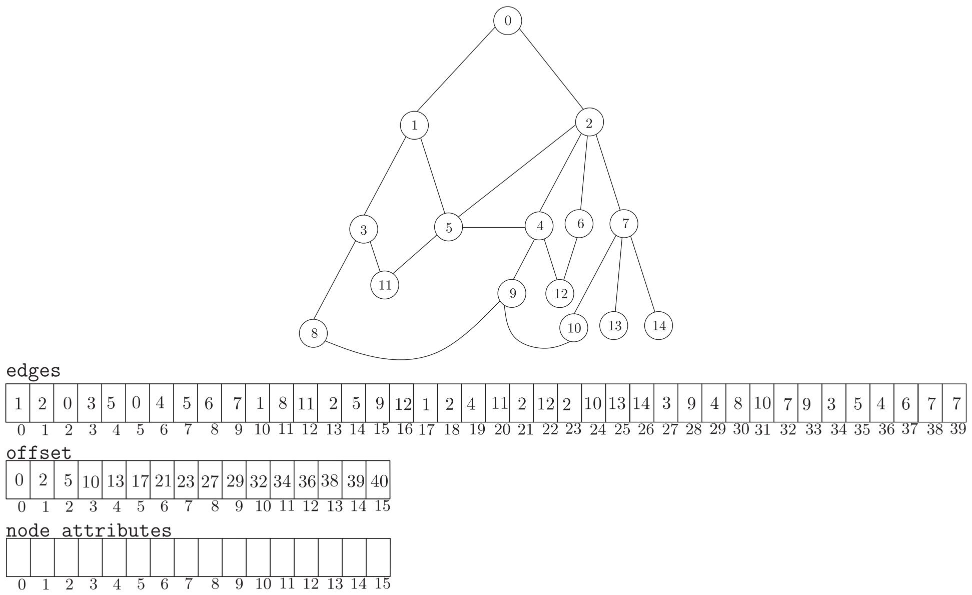
For instance, in Figure 2, assume the warp-size to be 4. The vertices 4–7 are assigned to threads having the same ID as the vertex. With vertex centric processing, the warp-threads will access the attributes of the first neighbor of the respective vertices concurrently, and so on. Hence, the warp threads will access the locations 0, 8, 2, and 10 in the *node attributes* array together. Further assume that the accesses to a chunk of 4 words can be coalesced. Clearly, the accesses to the destination vertices' {0, 8, 2, 10} data in the *node attributes* array are not coalesced, since these lie in three separate four-word chunks. We renumber the vertices such that the vertices to be accessed by the warp-threads are assigned nearby IDs; this results in improved coalescing. The vertex-renumbering is performed once at the time of loading the graph. The following is the renumbering scheme (for Line 3 in Algorithm 2):

Algorithm 3 presents the pseudocode for the vertex renumbering technique. We pick a lowest degree neighbor of a vertex having the highest degree and perform a BFS traversal on the graph, to obtain a BFS tree (line 3, Algorithm 3). The vertices at the same level in the BFS tree are assigned IDs in a round-robin fashion (lines 6–10, Algorithm 3): the first neighbor of each of the parents from the previous level is assigned a new ID followed by the renumbering of all the second-neighbors, and so on. The foregoing renumbering scheme ensures that the threads of a warp access nearby locations while accessing the attributes of the destination vertices in the *node attributes* array. Since the graph is undirected, the renumbering helps improve the coalescing in every outer iteration of Brandes' algorithm. The choice of the source of this BFS traversal helps on two accounts: (i) As we will see, we are likely to pick a vertex with low-degree as a source vertex in Brandes' algorithm (Section 5). So, picking a low-degree neighbor of the high-degree vertex as source ensures near-perfect coalesced accesses in an iteration of the Brandes' algorithm. (ii) A high-degree vertex is likely to be visited more often over all iterations in BC computation (Sections 5). So picking a neighbor of a high-degree vertex is a better choice than starting at an arbitrary vertex.

ALGORITHM 3: ParTBC technique for vertex renumbering

```
Input: An undirected, unweighted graph $G(V, E)$
Output: Reordered graph $G'(V, E)$
1  $v$.level = $\infty$     $\forall v \in G.V$
2  Node $s$ = minimum degree node
3  BREADTH_FIRST_TRAVERSAL($G$, $s$) //Assigns levels to nodes
4  gId = 0;
5  s.id = gId++;
6  for $i$ = 0 .. numLevels-2 do // numLevels is number of BFS levels
7      for $j$ = 0 .. (max node degree in $L_i$) do // $L_i$ is the list of nodes at level $i$
8          for Node $n$ : $L_i$ do
9              if (n.degree > j) && (n.neighbors[j] $\in L_{i+1}$) then
10                 n.neighbors[j].id = gId++
```

For example, in the graph G from Figure 2, vertex 12 has the highest degree. We perform BFS from vertex 8, which is a lowest degree neighbor of 12. Vertex 8 is at level zero, vertices 5 and 12 are at level 1, vertices 3, 6, 7, 11, 14 are at level 2, while other vertices are at level 3. Figure 3



edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	2	0	3	5	0	4	5	6	7	1	8	11	2	5	9	12	1	2	4

20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
11	2	12	2	10	13	14	3	9	4	8	10	7	9	3	5	4	6	7	7

offset

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	5	10	13	17	21	23	27	29	32	34	36	38	39	40

node attributes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig. 3. Graph *G* after vertex renumbering and its CSR representation.

shows the graph with vertices renumbered, with its memory layout. The attributes array for the destination vertices has more coalesced accesses in the renumbered graph.

The renumbering-scheme is applicable, in general, to several graph algorithms that are implemented using the vertex-centric approach and vertex values are propagated by updating neighbors' values through outgoing edges. Examples of such algorithms include single-source-shortest-path computation, pagerank computation, minimum-spanning-tree computation, among others.

5 TECHNIQUES FOR FAST BC ESTIMATION

We present a systematic study of the use of approximations in the computation of top-k betweenness centrality vertices. Our key observation is that not all sources in Brandes' algorithm (line 2, Algorithm 1) contribute equally to the BC values. We present a bouquet of techniques to identify vertices to be picked as sources that yield *enough* contribution to the BC of the vertices early to facilitate quicker identification of top-k BC vertices. Our strategy is to identify those vertices that would *eventually* have high BC values, in the early iterations of the Brandes' algorithm.

Based on *Property-1* and *Observation-1* from Section 2, we hypothesize that to impart a big share of their BC values early to the eventual high BC vertices, we should preferentially pick the low- and moderate-degree vertices as sources. The intuition is that such a choice of sources would increase the BC score of the high-degree vertex and not of the low and the moderate-degree neighbors of it. This would widen the gap between the eventual low and high BC vertices, thus allowing us to decide the top-k vertices in the early iterations. For our purpose, we categorize the vertices into low-order, moderate-order and high-order vertices. In the list of vertices sorted in ascending order by vertex degree, the first 25% are the low-order vertices, the next 50% are moderate-order vertices and the remaining 25% are the high-order vertices. Our experiments and analysis revealed that the number of iterations required for termination are indeed fewer when preferentially picking low- and moderate-order neighbors of high order vertices as sources. So, the high-level idea is to devise schemes that enable us to pick such vertices as sources in the early iterations. Careful filtering of the source vertices promises substantial performance gains because of reduced total work done.

Termination of execution. For each of the techniques, there are two ways to specify the termination of execution. One, we may specify the number of iterations as a percentage ($\alpha\%$) of the total number of iterations. Two, we may specify an online stopping criterion to get the desired accuracy. The second approach is preferable, since it allows us to control the performance–accuracy tradeoff.

For the latter approach, we need to define a metric that captures the quality of the top-k vertices reported. A desired metric for computing the output quality of top-k betweenness centrality vertices in a graph is the set difference between the set of exact top-k vertices and those computed using an approximate technique. This quality metric requires knowing the ground-truth (i.e., the exact top-k vertices), which would be available only upon running Brandes' algorithm to completion. Hence, we require a metric that can be computed in each iteration of Brandes' algorithm rather than at the end of the computation, to help us decide if we have reached the desired accuracy and thus terminate the execution.

A plausible proxy for the above metric is: tracking the vertex having the kth highest BC value in every iteration and checking if the vertex having the kth highest BC value has stabilized (i.e., it is unchanged for all remaining iterations).

Lemma 1. *If the kth highest BC vertex does not change across iterations, then the set of top-k BC vertices is unaltered.*

Proof. Consider two consecutive iterations of the outermost loop in Brandes' algorithm: i and $i+1$, such that $i < i+1 \leq |V|$. Note that BC of a vertex monotonically increases in every iteration. Let the set of vertices, V, be partitioned into two sets, S and S'. S contains the top-k ($k \leq |V|$) BC vertices and $S' = V \setminus S$. At each iteration, we maintain the invariant that the cardinality of S is k and that it holds the top-k BC vertices. Further, let us define a sequence on the elements of set S: $\pi_s = (v_1, v_2, v_3, \ldots, v_k)$ such that $\text{BC}(v_1) \geq \text{BC}(v_2) \geq \text{BC}(v_3) \ldots \geq \text{BC}(v_k)$. Let $v_k \in S$ be the last element in the sequence π_s, at the end of iteration i. Now, suppose at the end of iteration $i+1$, a vertex from S' moves to S (due to increase in its BC value), then a vertex from S must move to S' to maintain the invariant. Further, the vertex that moves from S' to S must have BC greater than or equal to $\text{BC}(v_k)$. So, the element to be displaced from S must have BC equal to $\text{BC}(v_k)$. Let us assume the last element in sequence π_s, i.e., v_k, will be displaced in the event of multiple vertices having the same BC value as v_k. Thus, if the element v_k is the same after iterations i and $i+1$, then it implies that the rest of the elements in the set S are also unaltered. The result holds for any two iterations i and j s.t. $i < j \leq |V|$. □

There are two issues with using the aforementioned proxy metric: (1) It requires determining the kth highest BC vertex after every iteration (which has a time complexity of $O(|V|)$), thus introducing significant time overhead. (2) The vertex with the kth highest BC needs to be tracked for all iterations to establish that the position of kth highest BC vertex is unchanged—this makes this metric unsuitable for online error estimation.

To design a pragmatic scheme for accurate estimation of online error, we draw on the observation that by the iteration when the ranks of highest *few* BC vertices are settled, the other high BC vertices also get a sufficiently large share of their respective BC values. Thus, tracking only a fixed number of highest BC vertices may suffice.

In each iteration of Brandes' algorithm, we maintain the set, S_t, of top-t ($t \leq k$) vertices. We track the number of successive iterations for which the set S_t remains unchanged. We terminate the execution when this count reaches C_t. The choices of t and C_t depend on the techniques used for picking the source vertices (Sections 5.1 through 5.5.2) and the type of the input graph. This scheme enables us to accurately estimate on-the-fly when the error in the top-k becomes substantially small.

Our experiments showed that for the techniques in Sections 5.1–5.3, $40 \leq t \leq 50$ and $5 \leq C_t \leq 10$ result in desirable speedups and accuracy. However, for the techniques presented in Sections 5.4–5.5, $5 \leq t \leq 10$ and $3 \leq C_t \leq 5$ are sufficient to achieve similar accuracy as the previous techniques. This can be attributed to the fact that the choice of sources using the latter techniques imparts a larger share of the respective BC values to the vertices quicker. Hence, stabilization of the ranks of a few high BC vertices for a few iterations indicates that the ranks of the other vertices are also stabilized, with a good chance.

We discuss the proposed ParTBC techniques below.

5.1 Selection of Source Vertices (Random)

From a uniformly random permutation of the vertices, we select a subset, guided by the stopping criterion. With this technique, we pick source vertices with varied connectivity and characteristics. In real-world scale-free graphs (that have many low-degree vertices and a few high-degree vertices), the probability of picking the high-degree vertices as source early is low due to their number. Hence, random selection of vertices naturally leads to selection of low- and moderate-degree sources. The vertices so picked include a fair number of non-high-degree neighbors of high-degree vertices. Hence, the BC scores of the vertices acquired in the early iterations causes the relative BC scores of the graph vertices to be representative of their relative exact BC values, leading to a high accuracy in top-k computation with a small number of iterations. However, we need a large value of t for getting high accuracy in less outerloop iterations. High value of t warrants computing the tth-largest BC vertex (*Lemma 1*) in each iteration, adding up to a high overhead.

5.2 Vertex Selection in Ascending Degree Order (Ascending)

A natural order is based on vertex degree: ascending and descending. We pick the vertices as sources in Brandes' algorithm in that order. The overhead of sorting of vertices ($O(|V| \log |V|)$), which is a one time operation, is a tiny fraction of the overall computation time of exact BC scores. We observe that in several real-world graphs, the low-degree vertices are connected to other low-degree vertices. So, the DAG formed by selection of a low-degree vertex as source has more levels as compared to the one resulting from picking a high-degree vertex as source; the DAG formed also has on an average few vertices at each level. However, when the sources are selected in ascending order of degrees, the vertices with high BC may not get enough contribution in the early iterations. This happens because the high-degree vertices appear towards the bottom of the DAG and thus not many vertices are reachable through them; this results in reducing the dependency contribution. Further, in real world graphs, low-degree vertices are connected to other low-degree vertices, so few low-order vertices are neighbors of high-order vertices. Hence, larger number of iterations are required to determine the top-k accurately. Similar to the *Random* technique, we need a large value of t, adding a significant execution overhead.

5.3 Vertex Selection in Descending Degree Order (Descending)

In this technique, we arrange the source vertices in decreasing order by degree. Generally, the high-degree vertices are found to be connected to other high-degree vertices. Hence, the DAG formed from such a vertex will have fewer levels, and on an average high number of vertices at each level. Since the sources selected are in descending order, all the vertices with high BC receive large BC contributions in the early iterations, because the high-degree vertices tend to appear at the top of the DAG, which results in more vertices being reachable via them. The moderate- and

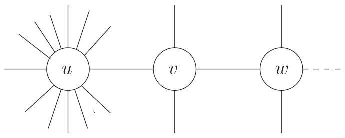


Fig. 4. A high-degree vertex having a low-degree neighbor.

low-order neighbors of high-order vertices are picked sooner than when following ascending order for selecting sources. So, fewer iterations are required to estimate the top-k with high accuracy. However, similar to *Ascending*, computing the stopping criteria has a high overhead.

5.4 Selecting Low-Degree Neighbors of High Degree Vertices

From *Observation-1*, high-degree vertices and the cut-vertices connecting large clusters are more likely to have large BC values. We also note that in Brandes' algorithm, when we pick a vertex as a source, the BC values increase more for those vertices that are at the initial levels of the DAG and have more vertices reachable from them. Thus, based on *Property-1*, to reduce the *polluting* of the vertices (increase in BC value of an unimportant vertex by as much as an important vertex, in an iteration), it is beneficial to pick the immediate neighbors of high-degree vertices as sources (Figure 4). Further, among the immediate neighbors we pick the low-degree neighbors as sources first, with the assumption that the low-degree vertices are likely to have lower BC than the high-degree vertices, in general. Consider the scenario in Figure 4. Suppose v and w are low-degree 1-hop and 2-hop neighbors, respectively, of a high-degree vertex, u. In this case, we prefer to pick the vertex v over w, as source. Picking v (1-hop neighbor of u) as source would increase the BC of u more than that of w, in that iteration, and hence widen the gap between their BC values, enabling computation of top-k vertices in fewer iterations. With this technique, the vertices connecting large clusters also get a high BC value, as desired. When a source is selected, the DAG formed has as the dominator of many other vertices, those vertices that connect large clusters; thus, all the vertices receive enough BC contribution.

The technique can be combined with *Ascending* and *Descending*. We consider only the high-order vertices and sort them in descending order by degree. Further, the neighbors of each of these are sorted in ascending order by degree. Additionally, vertices having equal degrees are arranged in ascending order by vertex id. We then select the neighbors in a round-robin fashion. Round-robin means that we select the unpicked lowest-degree neighbor of the highest degree vertex, followed by the lowest-degree neighbor of the second-highest vertex, that is not already selected, and so on. For example, in Figure 5, the vertices selected as sources are $v1a$, $v2a$, . . ., vta, $v1b$, $v2b$, and so on, in that sequence.

A caveat in the round-robin selection of source vertices is that in real-world graphs, high-degree vertices are often neighbors of other high-degree vertices, and this scheme may end up selecting the high-degree neighbors (instead of the low-degree neighbors) of high-degree vertices. To address this issue, we select only the low-order neighbors.

Interestingly, setting the threshold on the neighbors' degrees prohibits the selection of those high-order vertices as sources that are the neighbors of other high-order vertices. Since a vertex is chosen as source only once, this also prevents selection of a vertex with high-degree over a vertex with low-degree.

The threshold for high-order, moderate-order and low-order vertices can be tuned to control the accuracy in the top-k vertices, and the resulting speedup. We call this technique **Restricted-Round-Robin (RRR)**. Additionally, the number of low- and moderate-order neighbors of high-order vertices picked as source using this technique are substantially high by design. So,

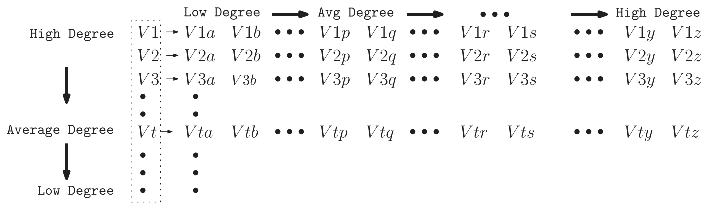

Low Degree
Avg Degree
High Degree
High Degree
Average Degree
Low Degree
V1 → V1a V1b ••• V1p V1q ••• V1r V1s ••• V1y V1z
V2 → V2a V2b ••• V2p V2q ••• V2r V2s ••• V2y V2z
V3 → V3a V3b ••• V3p V3q ••• V3r V3s ••• V3y V3z
Vt → Vta Vtb ••• Vtp Vtq ••• Vtr Vts ••• Vty Vtz

Fig. 5. Arrangement of vertices for the Round-Robin technique.

this method provides improved accuracy compared to earlier techniques, but *Random*, for similar speedups. Further, unlike in the prior techniques, the stopping condition here can be faithfully computed by tracking only up to 5 highest BC vertices across iterations, which is computable in $O(1)$.

5.5 Dynamic Selection of Source

In this technique, we also take into account the BC scores of the vertices up to that iteration to determine the source vertex for the next iteration. We observe that once the vertices get a *healthy* share of their respective final BC scores, the vertices with low BC are likely to continue having a low relative BC score in the subsequent iterations. We exploit this observation to select that so far unpicked vertex as source for the next iteration that has the least BC value up to that iteration. Again, based on *Property-1*, we further try to minimize the BC value of that particular vertex by selecting it as source. By minimizing BC value of the lowest BC vertex, we increase all the other remaining vertices' BC values and not just of a select few. So by selecting different sources dynamically, all the vertices having moderately high or high BC values get contributions relative to the vertices' eventual BC values by selecting the low BC vertices as sources and this also widens the gap between the eventual high BC and low BC vertices.

Now, to provide the vertices sufficient representative share of their BC values before going for the dynamic scheme, we execute the initial few iterations of the Brandes' algorithm. The choice of the source vertices for these initial iterations is crucial, since we want the vertices to have sufficient share of the BC values as quickly as possible. In the entire execution, no vertex is picked more than once. We discuss the heuristics for selecting these initial sources. We empirically found that selecting 5% source vertices is reasonable for contributing a good share of BC values to each vertex (and also improves execution time).

5.5.1 Descending (Dyn). We sort the vertices in descending order of their degrees. We then pick the top 5% vertices from this sorted list as the initial set of source vertices. After the 5% outer loop iterations, in the subsequent iterations, the vertex picked as source is a vertex with the least BC score up to that iteration that has not been picked already.

5.5.2 Restricted Round Robin (DynRR). We observed that with selection of fewer source vertices, *RRR* achieves better results than Dyn. However, if large number of sources are selected, then Dyn works better and has smaller error compared to *RRR* for the same number of sources. This suggests that picking the initial set of source vertices using *RRR* is beneficial.

For the first 5% iterations, we select the vertices as sources in a round robin fashion, selecting only those that have degree less than a threshold (which is set to average vertex degree). Further, for the selection of sources in the iterations following the first 5%, we consider neighbors of vertices having degree greater than the average degree. From this subset of vertices, the one having the least BC value up to that iteration is selected as source for the next iteration. This helps limit the search space further to only those vertices that when picked as source contribute significantly to boosting

Table 1. Input Graphs

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Graph type
fb-Friendships (FB)	63,731	817,035	Facebook friendship graph
soc-Pokec (SP)	1,632,803	30,622,564	Online social network
loc-Gowalla (LG)	196,591	950,327	Location-based social network
roadnetSF (RNSF)	174,424	221,802	San Francisco road network
usroad48 (RNUS)	102,615	147,656	Continental US road network
rmat17 (RMT)	130,977	2,091,451	R-MAT using GTgraph
random17 (RNM)	131,072	2,096,902	Random graph using GTgraph

Table 2. Effect of Vertex Numbering on Exact GPU Parallel Version

	Time (s)		Speedup	Graph reordering
Graph	NVR	VR	(NVR/VR)	time (s)
fb-Friendships	972	797	1.22×	5
soc-Pokec	204,360	166,150	1.23×	32
loc-Gowalla	25,486	21,598	1.18×	9
roadnetSF	2,762	2,444	1.13×	5
usroad48	3,629	3,211	1.13×	4
rmat17	1,477	1,241	1.19×	6
random17	587	564	1.04×	7

NVR: no vertex renumbering; VR: with vertex renumbering.

the values of the important vertices. Selecting the sources based on BC values computed till then increases the BC values of their neighbors (the high-degree vertices), which improves accuracy.

DynRR is found to perform the *best* among all the techniques, that is, it either has the lowest error for the same number of source vertices compared to other techniques, or it achieves better speedup than the other techniques for similar accuracy. It works well for all types of graphs and produces uniform results for all values of k.

Additionally, the number of low- and moderate-order neighbors of high-order vertices picked as source using Dyn and DynRR technique are substantially high by design. So, these methods provide improved accuracy and speedups compared to earlier techniques. Further, the stopping condition requires us to keep track of only up to 5 highest BC vertices across iterations, which is computable in $O(1)$. Hence, the overhead of computing the stopping criteria is negligible in each iteration.

6 EXPERIMENTAL EVALUATION

We evaluate the performance and effectiveness of ParTBC's techniques for estimating top-k BC vertices.

Experimental Setup. We use input graphs (Table 1) from SNAP [15] and KONECT [13], with different characteristics, to study the efficacy of our approach. These include social networks (such as Pokec) having small-world property, road networks (such as San Francisco) having large diameters, RMAT graphs, which are synthetically generated scale-free graphs [18], and random graphs, which do not exhibit any specific structure. We perform experiments on a machine with an Intel Xeon E5-2640 v4 @ 2.4 GHz CPU having 64 GB RAM and Nvidia Tesla P100 GPU having 3,584 cores spread across 56 SMXs with 12 GB memory. The machine runs CentOS 7.5 (64-bit). We use CUDA 8.0 to compile and execute our methods on the GPU.

Baselines. We use two baselines to evaluate our techniques. First, we compare the performance of ParTBC against the exact parallel Brandes' algorithm on GPU for computing top-k vertices. This

Table 3. Performance of PARTBC w.r.t. Exact Parallel Brandes' Algorithm and ABRA

Graph	Speedup w.r.t.		Speedup breakdown (w.r.t. exact parallel)	
	Exact Parallel	ABRA	VR	DynRR
fb-Friendships	2.80×	4.28×	1.22×	2.29×
soc-Pokec	2.76×	4.31×	1.20×	2.30×
loc-Gowalla	2.48×	4.16×	1.18×	2.10×
roadnetSF	2.71×	4.72×	1.13×	2.40×
usroad48	2.68×	4.63×	1.13×	2.37×
rmat17	2.65×	4.18×	1.19×	2.22×
random17	1.67×	1.92×	1.04×	1.61×
geomean	**2.5×**	**3.88×**	**1.15×**	**2.17×**

VR: Vertex Renumbering; Error ~6%.

is Baseline-I. Second, we compare the performance of PARTBC with ABRA [26], the state-of-the-art in approximate BC top-k computation. This is Baseline-II.

Effect of graph reordering. The execution times for the exact BC computation (which dominates top-k computation) on the graphs for Baseline-I are presented in Table 2. In Table 2, we report the time taken in the exact BC computation with and without the graph reordering (Phase-I of Algorithm 2). Note that the graph reordering technique is independent of Phase-II of Algorithm 2 and depends only on the input graph. We also report the time taken for graph reordering, which is observed to be negligible (less than 1%) compared to the total execution time of the BC computation. This reaffirms our time-investment in vertex renumbering.

We observe that the performance of the exact parallel version on the modified graph layout (resulting from vertex-renumbering) is consistently better than that on the original layout, for all types of graphs, resulting in an average speedup of 1.15×. The improvement is primarily due to better global memory coalescing. We also observe that power-law graphs (such as FB), get benefited more due to coalescing. This happens due to high number of active vertices in an iteration (due to small diameter), in the level-synchronous BFS traversal in the forward-pass. This leads to more coalesced memory accesses once the graph is made more structured. For road networks (e.g., RNSF), the gains are limited as only a few vertices are active in each iteration and the number of accesses to nearby vertices (after renumbering) is limited. Due to a lack of structure, random graphs are not sensitive to vertex renumbering.

6.1 Overall Results

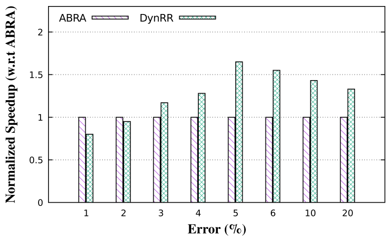
We evaluate the inaccuracy of PARTBC techniques as follows: Let bc_G be the exact set of top-k vertices, and $\widetilde{bc}_G$ be the set of top-k reported by PARTBC. The error incurred for each of the techniques is measured as $1 - \frac{|bc_G \cap \widetilde{bc}_G|}{k}$.

Each of PARTBC's techniques returns a set of top-k vertices. By design, for a technique, its *recall* and *precision* are equal, since the number of false positives (vertices incorrectly included in top-k) and the number of false negatives (the actual top-k vertices, which are missed in the reported top-k) are equal. The recall and precision are given by $\frac{|bc_G \cap \widetilde{bc}_G|}{k}$. Thus, low inaccuracy of a technique implies high recall and precision, which is desirable.

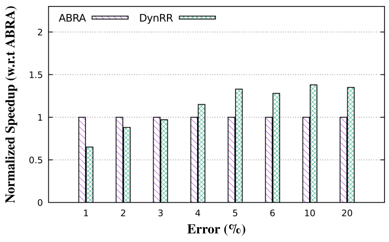
Comparison with Baseline-I and Baseline-II. Table 3 compares the performance of *DynRR* from PARTBC with the two baselines. We report the speedups averaged across $k \in \{100, 500, 1{,}000, 2{,}000, 3{,}000, 5{,}000\}$ for error ~6% for both ABRA and PARTBC.

Table 4. Comparison of Performance of CPU Version of DynRR w.r.t. ABRA for Error ~6%

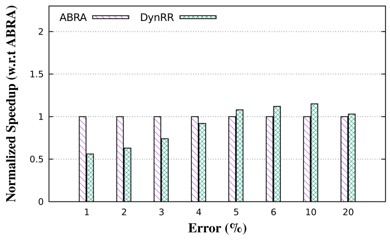
Graph	Speedup of DynRR w.r.t. ABRA on CPU
fb-Friendships	1.32×
soc-Pokec	1.55×
loc-Gowalla	1.28×
roadnetSF	1.74×
usroad48	1.86×
rmat17	1.45×
random17	1.12×
geomean	**1.45×**



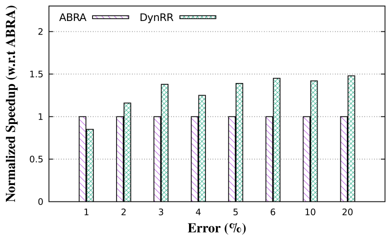
(a) soc-Pokec



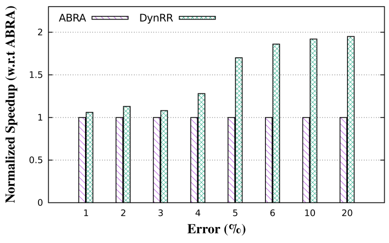
(b) loc-Gowalla



(c) random17



(d) rmat17



(e) usroad48

Fig. 6. Performance comparison of CPU version of DynRR and ABRA for different error%.

We observe that the geomean speedup of ParTBC w.r.t. Baseline-I is 2.5× while that w.r.t. ABRA is 3.88×. Note that ABRA has a multi-threaded CPU implementation, which we executed with 24 threads for best performance in our setup. We also present a breakdown of the contribution of each of the two phases of Algorithm 2 in the speedup achieved w.r.t. Baseline-I. The results show that for the size and type of graphs in our test suite, the ParTBC outperforms ABRA consistently without

Table 5. Performance of ParTEdge w.r.t. Exact GPU-Parallel Brandes' Algorithm on the Reordered Graph

Graph	Speedup w.r.t. exact parallel	Error
fb-Friendships	2.01×	10%
soc-Pokec	2.02×	10%
loc-Gowalla	2.01×	10%
roadnetSF	2.02×	9%
usroad48	2.02×	9%
rmat17	2.02×	10%
random17	2.00×	12%
geomean	**2.01×**	**10%**

failing the accuracy constraint. This is because ABRA uses an iterative progressive-sampling-based approximation algorithm. Its execution time is contingent on the sample size, the number of samples, and the number of iterations, all of which grow with the size of the graph for a specified value of accuracy. Additionally, the sample size is updated in each iteration. In contrast, ParTBC scales well with the size of the graph. There is very gradual increase in the overhead (which primarily includes time for graph reordering in Phase-I, time for online selection of sources, and time for computing the termination condition) with increase in graph size. In addition, ParTBC reduces the amount of work done by the same factor irrespective of the size of the graph, for obtaining the specified accuracy. Hence, even for moderate-sized graphs as in our test-suite, ABRA takes longer than ParTBC across graphs and different values of k. All techniques in ParTBC consistently outperform ABRA on moderate- and large-sized graphs. ABRA performs better than ParTBC with better accuracy on small graphs such as Enron-email ($|V| = 36{,}682$ $|E| = 183{,}831$). Since ABRA runs on CPU, we also compare ABRA with a CPU-only version of ParTBC. Table 4 presents the comparison of CPU-only version of *DynRR* w.r.t. ABRA. We observe that the geomean speedup of the CPU-version of *DynRR* w.r.t. ABRA is 1.45×. *DynRR* consistently performs better than ABRA for error ~6% even on CPU. Figure 6 shows the speedup of CPU-only version of *DynRR* w.r.t. ABRA for different error values. We observe that for most of the graphs, ABRA performs better than *DynRR* for error ≤2%. In the case of random graph (RNM), ABRA outperforms *DynRR* if the desired accuracy is up to 4%. In the case of road-network graph (RNUS), *DynRR* is consistently better than ABRA for all error values. The power-law graphs have a heavy tail and the high-degree vertices are connected to other high-degree vertices. In road-networks, however, the vertex degrees are small and largely uniform. So, *DynRR* performs well on power-law graphs and road-networks, since such graphs have sufficient number of low- and moderate-degree neighbors of high-degree vertices, which the technique attempts to prioritize during source selection. However, *DynRR* does not perform well on the random graph. This is primarily because random graphs often lack structure and there is a paucity of low- and moderate-degree neighbors of high-degree vertices, rendering the technique less effective.

As observed, ParTBC outperforms the Baseline-I in exchange for lower accuracy. This is expected due to the lower overall work done by the ParTBC techniques.

Besides the two baselines, we also explore a divergent approach to approximate top-k BC vertices, for establishing a different comparison point for ParTBC techniques. We call this approach ParTEdge. ParTEdge deletes ~50% edges of the input graph by picking the edges to remove uniformly at random while ensuring that the graph remains a single connected component. The vertices are not deleted. This modified graph is input to the exact GPU-parallel version of Brandes' algorithm. Note that with ParTEdge, the number of outerloop iterations is $|V|$, however, in every

Table 6. Performance of ParTBC w.r.t. Exact GPU-Parallel Brandes' Algorithm on the Reordered Graph

	α	Mean speedup	Mean error		α	Mean speedup	Mean error
Random	0.1	7.6×	9.7%	**Dyn**	0.1	8.3×	16.4%
	0.2	3.3×	7.3%		0.2	3.9×	11.9%
	0.3	2.6×	5.4%		0.3	2.7×	8.6%
	0.4	1.8×	4.9%		0.4	2.2×	7.1%
	0.5	1.3×	3.7%		0.5	1.8×	5.5%
	0.6	1.2×	3.0%		0.6	1.2×	4.2%
	0.7	1.1×	2.4%		0.7	1.1×	3.0%
	0.8	1.1×	2.0%		0.8	1.1×	2.0%
Ascending	0.1	7.4×	27.7%	**Descending**	0.1	7.3×	17.3%
	0.2	3.2×	20.1%		0.2	3.2×	13.0%
	0.3	2.4×	14.6%		0.3	2.4×	9.6%
	0.4	1.6×	12.0%		0.4	1.7×	8.7%
	0.5	1.3×	9.3%		0.5	1.4×	6.6%
	0.6	1.1×	4.9%		0.6	1.2×	5.0%
	0.7	1.1×	3.6%		0.7	1.1×	4.7%
	0.8	1.1×	2.2%		0.8	1.1×	3.5%
RRR	0.1	8.1×	11.9%	**DynRR**	0.1	8.7×	15.8%
	0.2	3.6×	10.0%		0.2	3.8×	10.5%
	0.3	2.4×	7.9%		0.3	2.8×	8.1%
	0.4	2.1×	7.1%		0.4	2.1×	6.9%
	0.5	1.6×	5.8%		0.5	1.7×	5.3%
	0.6	1.2×	4.2%		0.6	1.3×	4.1%
	0.7	1.1×	3.0%		0.7	1.2×	2.8%
	0.8	1.1×	2.5%		0.8	1.1×	1.6%

outerloop iteration, the edges traversed are only $\sim\frac{|E|}{2}$. Hence, the overall work done is lesser compared to the exact version. Table 5 compares the performance of ParTEdge with the exact parallel implementation of Brandes' algorithm, on the reordered graph. We observe that with ParTEdge, the mean speedup is 2.01× at the expense of 10% mean error w.r.t. the exact parallel Brandes' algorithm for computing the top-k vertices. The speedup is on account of a reduction in the total work done by 50%, since with ParTEdge we process only 50% edges in every iteration of Brandes' algorithm. The error in top-k vertex computation is contingent on which edges are removed from the original graph. Since the removal of edges impacts connectivity and structure of the graph, the effect of edge removal on error also depends on the characteristics of the original graph. As we can observe, with the removal of nearly 50% edges, the error is ∼10% for power-law graphs, while it is ∼9% for road-networks and 12% for random graph. In contrast to ParTEdge, with the *DynRR* technique, we achieve ≤6% error for a speedup ∼2×.

Evaluation of Source Selection Techniques in ParTBC. We next evaluate the effect of the approximate techniques for selection of sources (Section 5). We do so by using the parallel implementation of Brandes' algorithm (from Baseline-I) executed on the modified graph layout, post vertex-renumbering as the new baseline. We call this Baseline-III. In Table 6, we report the comparison of the ParTBC techniques with Baseline-III. This is the geomean speedup and error of the different techniques in ParTBC computed across $k \in \{100, 500, 1{,}000, 2{,}000, 3{,}000, 5{,}000\}$ for all graphs in our testbed. α denotes the fraction of vertices chosen as sources.

We observe that the techniques achieve high speedups for smaller α. This is due to fewer outerloop iterations. However, as one would expect, the approximation error is also high. As α increases,

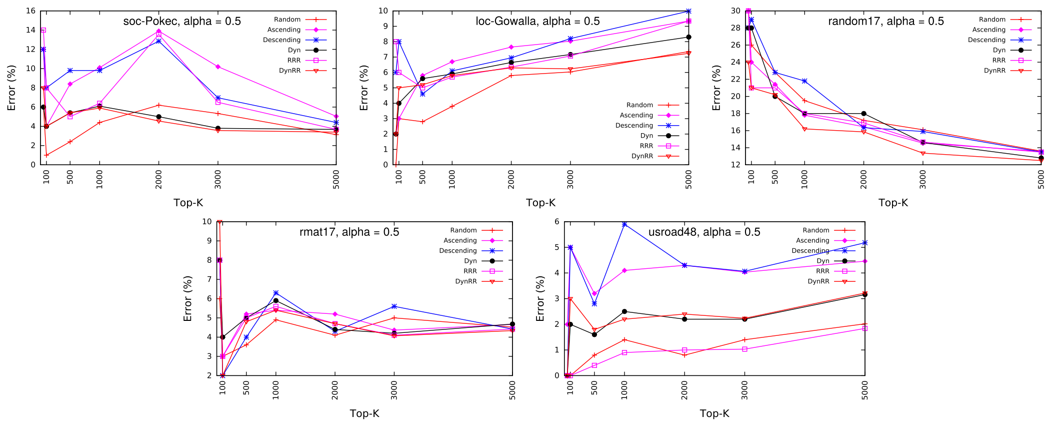


Fig. 7. Graph-wise effect of varying k on the error for $\alpha = 0.5$.

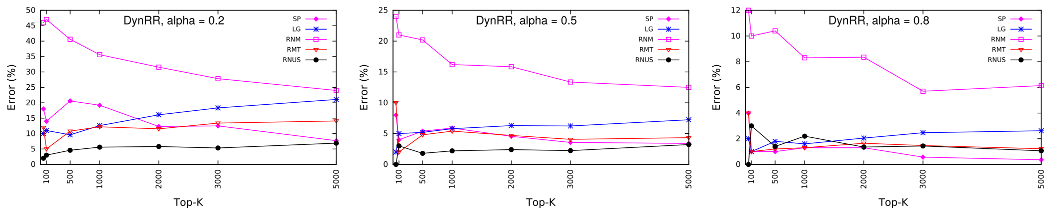


Fig. 8. Effect of varying k on the error for DynRR.

both the speedup and the error reduce. Out of the six variants, *Random* and *DynRR* achieve the least error. The effective difference across techniques, however, reduces with increasing α; for instance, beyond $\alpha = 0.8$, all the techniques achieve similar accuracy.

We observe that for $\alpha \geq 0.5$, the overall mean error is less than 6% for techniques *Random, RRR, Dyn, DynRR*, with decent speedups. Thus, these techniques bear the potential to estimate the set of top-k vertices faster with a small accuracy loss for a variety of graphs.

To examine the robustness of the proposed techniques, we study the effect of varying k, on error. Figure 7 shows the variation in error with k on different graphs for various proposed techniques for $\alpha = 0.5$. As a general trend across all graphs, the absolute error in top-k increases with k.

For $\alpha = 0.5$, the error is least for **road-network graph (RNUS)**—below 6% across all techniques, while the error is highest for **random graph (RNM)**. In the case of RNUS graph, *RRR* maintains the least error (below 2%) for all k. *Random*, *Dyn*, and *DynRR* follow in closely with error less than 3%. The increase in error with k is slow for these four techniques. *RRR* leads to the least error for road-networks, because a road-network has low, uniform vertex degrees. Thus selecting the vertices in a round robin fashion selects favorable source vertices. *Random* technique performs well on road-networks, since vertices have similar characteristics.

We note that for power-law graphs (e.g., SP, RMT), the error is small (below 6%) for *Random*, *Dyn*, and *DynRR* techniques. The error for these three techniques decreases rapidly with increase in k, and the errors are also very similar for large k. Up to $k = 2{,}000$, *Random* has the least error (below 4%), but for higher k, the *Dyn* and *DynRR* have the least errors (below 4%). In general, *DynRR* performs well for larger k for all graphs, because the initial set of seed vertices selected using round robin technique provides a fair share of the BC values to the various important vertices.

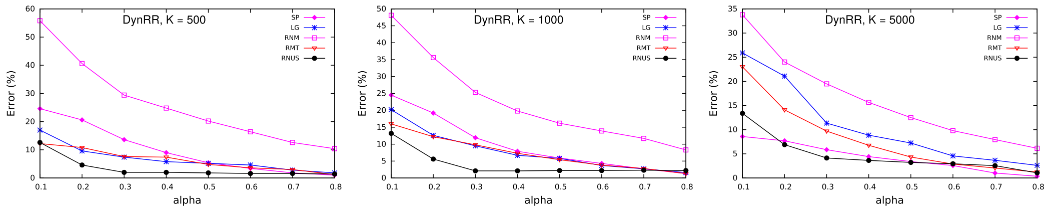


Fig. 9. Effect of varying α on the error for DynRR.

Among all the graphs, random graph (e.g., RNM) exhibits the highest error. This is because of a lack of well-defined structure. It is noteworthy that for *DynRR* (Figure 8), the error does not exceed 7% for any of the graphs for $\alpha = 0.5$ for reasonably large k in our setup. The deviation in error is also small. Overall, *DynRR* emerges as a very stable technique that works consistently well across all but random graphs, for all k. We observed similar trends in error for $\alpha = 0.8$.

6.2 Effects of the Fraction of Source Vertices

Figures 10 and 11 show the variation in error with α for various proposed techniques for $k = 500$ and $k = 5{,}000$, respectively. For each graph, for each technique, the error decreases with increase in α, since a higher value of α translates to performing more work and moving closer to the exact version ($\alpha = 1$).

For $k = 500$, we observe that for road-network graphs (e.g., RNUS), beyond $\alpha = 0.3$ the change in error is small and very gradual, for *Dyn* and *DynRR*. The error is also low. The small slope of the curve suggests that there is little change in the relative ordering of the BC scores of the vertices on choosing more source vertices. This also hints at the scalability of these techniques for road-networks. We may terminate Brandes' algorithm after 30% iterations, without incurring high error, and gain immensely in execution time.

The error tends to decrease slowly in case of SP and RMT graphs with *DynRR*, *RRR*, *Dyn* and *Random* for $\alpha \geq 0.5$. For RNM graph, it is seen that there is a steep decrease in error values with increase in α. This shows that relative ordering of vertices continues to change even for high α. Thus, for random graphs, exact BC needs to be computed for top-k. Overall, we observe that the technique *DynRR* has quite good accuracy for most of the graphs (Figure 9) for $\alpha \geq 0.5$. At $\alpha = 0.5$, the error is less than 5% for most of the graphs and as α increases the percentage error decreases, which indicates the robustness and the scalability of the technique. We observed similar trends for higher values of k.

The results suggest that the techniques *DynRR* and *Random* are comparable. In practice, both the techniques have their advantages. *DynRR* is faster than *Random* because of a lower overhead in computing the stopping condition. However, a benefit *Random* has over *DynRR* is simplicity. In contrast to *DynRR*, *Random* does not require an involved and difficult (for the average user) parameter selection based on the input graph characteristics.

6.3 Controlling the Number of Outerloop Iterations

For all the techniques in ParTBC, the number of outerloop iterations can be tuned by setting an appropriate stopping criterion. For every technique, the speedup and error are tied to the number of outerloop iterations. As discussed in Section 5, the stopping criterion is a function of C_t and t. In general t dominates C_t in governing the number of iterations as the top-t vertices typically stabilize in P ($\gg C_t$) iterations.

The trends in Figures 10 and 11 can be used as guidelines for setting C_t and t for different types of graphs. In general, to get good speedups and accuracy across all the proposed techniques, C_t

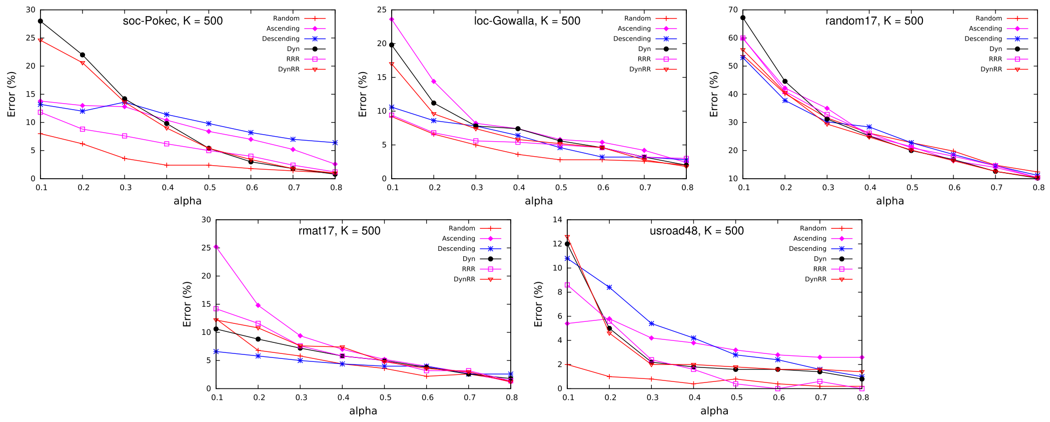


Fig. 10. Graph-wise effect of varying α on the error for $k = 500$.

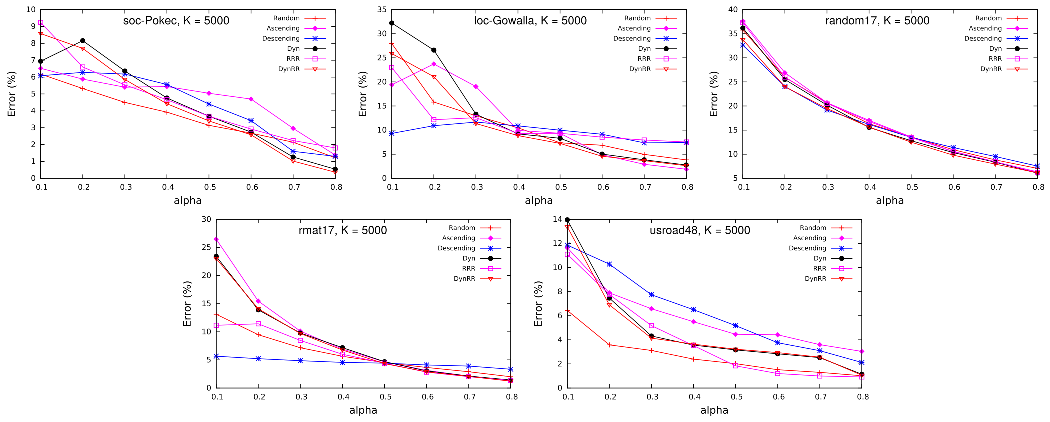


Fig. 11. Graph-wise effect of varying α on the error for $k = 5{,}000$.

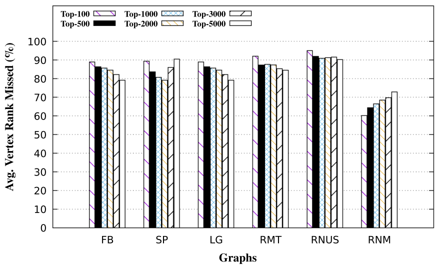
and t should both be set to a small value for road-networks. For power-law graphs, C_t and t should be higher.

In our experiments, for *DynRR*, we assign $C_t = 5$ and $t = 5$ for power-law graphs, and $C_t = 4$ and $t = 3$ for road-networks. This resulted in execution of 55% outerloop iterations for LG graph and 28% iterations for the RNUS graph using *DynRR*. However, for *Random*, we assign $C_t = 10$ and $t = 50$ for power-law graphs, and $C_t = 5$ and $t = 40$ for road-networks. This resulted in execution of 60% outerloop iterations for LG graph and 35% iterations for the RNUS graph using *Random*.

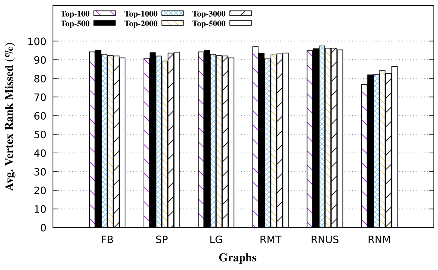
6.4 Discussion on Quality of the Reported Top-k Vertices

To assess the quality of the top-k vertices reported by the proposed techniques, we consider two measures: (i) The mean exact rank of the vertices in top-k that are not reported using the approximate techniques. (ii) The mean exact rank of the vertices that are not in the exact top-k but are erroneously included.

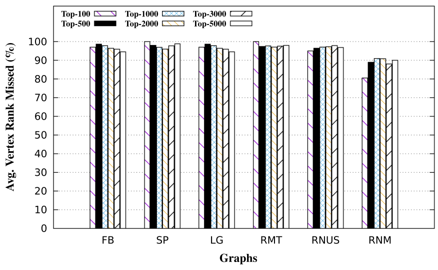
We argue, on empirical evidence, that the vertices we report as top-k are indeed *important*. We observe that the top-k vertices we fail to include in the set of top-k vertices are the ones that have their true ranks close to k, based on their exact BC scores. So, our techniques miss out on less



(a) $\alpha = 0.2$

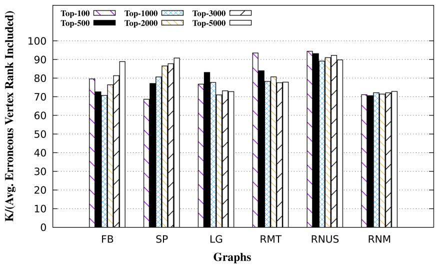


(b) $\alpha = 0.5$

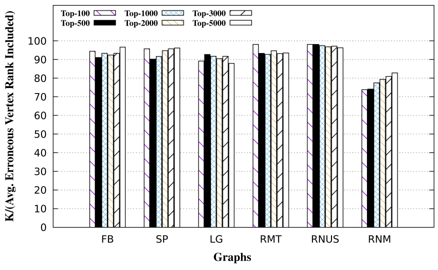


(c) $\alpha = 0.8$

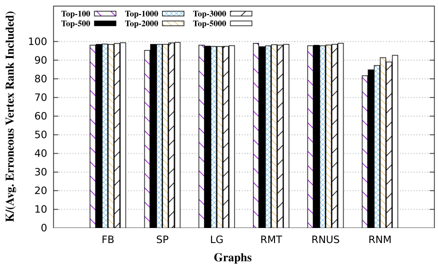
Fig. 12. Average rank of top-k vertices missed (expressed as a percentage of k).



(a) $\alpha = 0.2$



(b) $\alpha = 0.5$



(c) $\alpha = 0.8$

Fig. 13. Average rank of vertices erroneously included in top-k (expressed as k / Avg. rank).

important vertices among the exact top-k. Figure 12 shows the average true rank of the vertices we missed for *DynRR*. The average rank is expressed as a fraction of k. The vertices having lower ranks are more important. In the plot, values close to 100% depict that the vertices we missed have an actual rank closer to k. We observe that for low α (Figure 12(a)), we miss more number of important vertices for all types of graphs. This is due to high error in the reported top-k. For

$\alpha = 0.5$ (Figure 12(b)), the values lie between 90% and 100% for all graphs except for random graphs, for which the error in top-k is high. Following a similar trend, the accuracy approaches 100% for $\alpha = 0.8$ (Figure 12(c)) for all the graphs. Among the graphs, road network is the most stable for all k for all α, while random graph is the most unstable. In general, we observe that average rank of the vertices missed (as a percentage of k) is proportional to the accuracy of the technique. This shows that we miss the vertices having true ranks in the range (accuracy% of k, k).

We further observe that the vertices we erroneously include in the top-k vertices are also among the important vertices although they are not in the exact top-k vertices. To verify this, we observed the following ratio:

$$R = \frac{k}{\text{true avg. rank of erroneous vertices}}.$$

Figure 12 shows this ratio, expressed as a percentage, for *DynRR*. The higher the R, the closer is the average rank of the erroneous vertices to k. From the plot, we can compute the range of the true ranks of the erroneously included vertices to be within $((1 - 1/R) \times k)$ away from k on an average. For $\alpha = 0.5$ (Figure 13(b)), the values are in the range 90%–100% for all the graphs except random graph. The true rank of the vertices erroneously included in the top-k are in the range $[k + 1, \frac{1}{0.9} \times k)$, that is, $[k + 1, 1.1 \times k)$. For higher α, the range shrinks. For $\alpha = 0.8$ (Figure 13(c)), the range reduces to $[k + 1, 1.02 \times k)$ on an average for all graphs except random graph. Since the true rank of the erroneous vertices is close to k, the vertices we include in the top-k are important.

7 CONCLUSIONS

In this article, we presented a systematic study of lightweight heuristics for selecting source vertices in Brandes' algorithm that enable us to determine the relative ordering among the vertices quicker. We established that preferentially picking low-degree neighbors of high-degree vertices in Brandes' algorithm ensures that all graph vertices receive a sizeable fraction of their respective eventual BC scores in the early iterations; thus, facilitating quicker estimation of the top-k vertices. We demonstrated empirically that our proposed techniques compute the top-k BC vertices 2.5× faster compared to the exact parallel Brandes' algorithm, with a mean error of less than 6% on graphs of varying characteristics. We identified that in most real-world graphs, the high-degree vertices that are connected to other high-degree vertices tend to have a high relative BC score. In addition, the vertices that are connected to well-connected clusters within the graph (e.g., a cut vertex) also have a high BC score.

Our techniques have a high precision and recall. These are robust and work well for larger values of k. Further, we proposed a novel vertex-renumbering scheme to make the graph layout more structured, having better locality, to enable efficient coalesced accesses to/from global memory in parallel Brandes' algorithm. This resulted in a mean speedup of 1.15×. The renumbering scheme is beneficial to other parallel graph algorithms on GPU that employ vertex-centric push-style implementations.

REFERENCES

[1] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating betweenness centrality. In *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph (WAW'07)*. Springer-Verlag, Berlin, 124–137. http://dl.acm.org/citation.cfm?id=1777879.1777889

[2] David A. Bader and Kamesh Madduri. 2006. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proceedings of the International Conference on Parallel Processing (ICPP'06)*. IEEE Computer Society, Washington, DC, 539–550. https://doi.org/10.1109/ICPP.2006.57

[3] Vignesh Balaji and Brandon Lucia. 2019. Combining data duplication and graph reordering to accelerate parallel graph processing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'19)*. Association for Computing Machinery, New York, NY, 133–144. https://doi.org/10.1145/3307681.3326609

[4] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *J. Math. Sociol.* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249

[5] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. 2004. Graph-based technologies for intelligence analysis. *Commun. ACM* 47, 3 (Mar. 2004), 45–47. https://doi.org/10.1145/971617.971643

[6] Robert Geisberger, Peter Sanders, and Dominik Schultes. 2008. Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering and Experiments*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 90–100.

[7] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Proc. Natl. Acad. Sci. U.S.A.* 99, 12 (2002), 7821–7826. https://doi.org/10.1073/pnas.122653799

[8] Mostafa Haghir Chehreghani. 2013. An efficient algorithm for approximate betweenness centrality computation. In *Proceedings of the Conference on Information and Knowledge Management (CIKM'13)*. ACM, New York, NY, 1489–1492. https://doi.org/10.1145/2505515.2507826

[9] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2014. KLA: A new algorithmic paradigm for parallel graph computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. Association for Computing Machinery, New York, NY, 27–38. https://doi.org/10.1145/2628071.2628091

[10] Loc Hoang, Matteo Pontecorvi, Roshan Dathathri, Gurbinder Gill, Bozhi You, Keshav Pingali, and Vijaya Ramachandran. 2019. A Round-Efficient distributed betweenness centrality algorithm. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. Association for Computing Machinery, New York, NY, 272–286. https://doi.org/10.1145/3293883.3295729

[11] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong. 2010. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'10)*. 1–7. https://doi.org/10.1109/IPDPS.2010.5470400

[12] Hyoungshick Kim, John Tang, Ross Anderson, and Cecilia Mascolo. 2012. Centrality prediction in dynamic human contact networks. *Comput. Netw.* 56, 3 (2012), 983–996. https://doi.org/10.1016/j.comnet.2011.10.022 (1) Complex Dynamic Networks (2) P2P Network Measurement.

[13] Jérôme Kunegis. 2017. KONECT: The Koblenz network collection. Retrieved from http://konect.uni-koblenz.de.

[14] Min-Joong Lee and Chin-Wan Chung. 2014. Finding K-highest betweenness centrality vertices in graphs. In *Proceedings of the 23rd International Conference on World Wide Web (WWW'14)*. ACM, New York, NY, 339–340. https://doi.org/10.1145/2567948.2577358

[15] Jure Leskovec and Rok Sosič. 2014. SNAP: A general purpose network analysis and graph mining library in C++. Retrieved from http://snap.stanford.edu/snap.

[16] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. Aberg. 2001. The web of human sexual contacts. *Nature* 411 (2001), 907–908. https://doi.org/10.1038/35082140

[17] Wai-Hung Liu and Andrew H. Sherman. 1976. Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.* 13, 2 (1976), 198–213. https://doi.org/10.1137/0713020

[18] Kamesh Madduri and David A. Bader. 2006. GTgraph: A suite of synthetic random graph generators. Retrieved from http://www.cse.psu.edu/~madduri/software/GTgraph/.

[19] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09)*. IEEE Computer Society, Washington, DC, 1–8. https://doi.org/10.1109/IPDPS.2009.5161100

[20] Adam McLaughlin and David A. Bader. 2014. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Press, Piscataway, NJ, 572–583. https://doi.org/10.1109/SC.2014.52

[21] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *ACM Comput. Surv.* 48, 4, Article 62 (Mar. 2016), 33 pages. https://doi.org/10.1145/2893356

[22] Sara Mumtaz and Xiaoyang Wang. 2017. Identifying Top-K influential nodes in networks. In *Proceedings of the ACM on Conference on Information and Knowledge Management (CIKM'17)*. ACM, New York, NY, 2219–2222. https://doi.org/10.1145/3132847.3133126

[23] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 622–636. https://doi.org/10.1145/3173162.3173180

[24] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, 12–25. https://doi.org/10.1145/1993498.1993501

[25] 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, 35–46. https://doi.org/10.1145/2442516.2442521

[26] Matteo Riondato and Eli Upfal. 2018. ABRA: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. *ACM Trans. Knowl. Discov. Data* 12, 5, Article 61 (July 2018), 38 pages. https://doi.org/10.1145/3208351

[27] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. 2013. Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU'13)*. ACM, New York, NY, 76–85. https://doi.org/10.1145/2458523.2458531

[28] Somesh Singh and Rupesh Nasre. 2018. Scalable and performant graph processing on GPUs using approximate computing. *IEEE Trans. Multi-scale Comput. Syst.* 4, 3 (2018), 190–203. https://doi.org/10.1109/TMSCS.2018.2795543

[29] Somesh Singh and Rupesh Nasre. 2019. Optimizing graph processing on GPUs using approximate computing: Poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. Association for Computing Machinery, New York, NY, 395–396. https://doi.org/10.1145/3293883.3295736

[30] Somesh Singh and Rupesh Nasre. 2020. Graffix: Efficient Graph processing with a tinge of GPU-specific approximations. In *Proceedings of the 49th International Conference on Parallel Processing (ICPP'20)*. Association for Computing Machinery, New York, NY, Article 23, 11 pages. https://doi.org/10.1145/3404397.3404406

[31] 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 (SC'17)*. ACM, New York, NY, Article 47, 14 pages. https://doi.org/10.1145/3126908.3126971

Received February 2021; revised July 2021; accepted August 2021