Computing Betweenness Centrality Of Massive-Scale Unweighted, Directed Graphs

Prabhanjan Kambadur
I.B.M. T.J. Watson Research Center
Yorktown Heights, NY 10592,
pkambadu@us.ibm.com

***Abstract*—Betweenness centrality (BC) is an important metric that can be used to determine the power of a vertex in a graph [16], [3]. Computing BC for an unweighted, directed graph $G(n, m)$, where n is the number of vertices and m is the number of edges, is both computationally and spatially demanding; the best known serial algorithm requires $O(nm)$ time and $O(n+m)$ space [8]. Furthermore, given that the sizes of graphs that need analysis is ever-increasing, parallelization of the BC computation is a must. To compute BC for massive-scale graphs on modern super computers, which by nature offer hierarchical parallelism (i.e., support shared- and distributed-memory parallelism simultaneously), it is important to develop a parallel algorithm that is also hierarchical for optimal performance. There have been several attempts at parallelizing BC computations [17], [19], [13]; however, these attempts are specialized for shared- or distributed-memory machines and have also failed to show significant scaling. Owing to the apparent infeasibility in computing *exact* BC, approximations have also been implemented [5], [10]. TODO: In this paper, we summarize the state-of-the-art algorithms for computing BC (both serial and parallel). Furthermore, we develop a new hierarchical algorithm for computing BC, which improves on existing solutions.**

I. Introduction

TODO: Write introduction

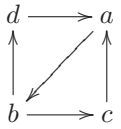
II. Background

Betweenness Centrality (BC) is a graph theoretic metric of a node's importance in a graph. [1] Informally, a node's BC gives an indication of the ratio of the total number of shortest paths between all other nodes taken pair-wise. That is, a node with a high centrality score can reach other nodes with relatively fewer hops than another node with a lower centrality score. More formally, let $G(V, E)$ be a graph with the vertex set V and edge set E; let the number of vertices ($|V|$) be n and the number of edges ($|E|$) be m. The BC of a node (vertex) $v \in V$) is given by the equation:

$$BC_v = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

Where σ_{st} is the number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those shortest paths that go through node v. There are multiple ways of computing the BC scores of nodes in a graph; in the following subsections, we highlight a few of these techniques. Please note that regardless of the approach that is used in computing BC, the central kernel is enumerating all the shortest paths.

[1] In this paper, we focus exclusively on the unweighted, directed graphs.



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 1. A sample directed, unweighted graph and its resulting adjacency matrix. A 1 in position a_{ij} indicates an edge from node i to node j.

$$A^2 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad A^3 = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Fig. 2. The 2^{nd} and 3^{rd} powers of the adjacency matrix, A, shown in Figure 1. A^2 shows the paths of path length 2, and A^3 shows the paths in the sample graph of path length 3. The entries in A, A^2, and A^3 indicate the number of paths. For example, $A^2_{2,1} = 2$ as there are two paths of length 2 between b and a in Figure 1.

A. Algebraic Approach

Every graph $G(V, E)$ can be represented as an adjacency matrix A, where an entry a_{ij} is marked as 1 *iff* $(i, j) \in E$. Figure 1 shows a sample four node graph and its related adjacency matrix; we will be using this graph as a running example throughout this section. By inspection, we can tell that the diameter of the graph is 3; therefore, all the possible paths in the matrix (including the number of paths) can be enumerated by simply taking the powers of the adjacency matrix A; in other words, we find the transitive closure of the graph G. However, computing the transitive closure is an over-kill; what we want in order to compute BC are just the shortest paths between all pair of vertices, not all paths of length diameter or less.

Another possible solution was proposed by Batagelj [7] and involved modifying Floyd/Warshall's algorithm [15], [20] for all pairs shortest paths. Briefly, Floyd/Warshall's algorithm computes all pairs shortest paths given an adjacency matrix representation of a weighted graph in $O(n^3)$ time; the algorithm is outlined in Algorithm 1. [2] In his solution, Batagelj

[2] Floyd/Warshall's does not handle negative cycles, although it can be used to detect such cycles in a graph.

Algorithm 1: FloydWarshall

Input: A: Adjacency matrix of graph $G(V, E)$

```
for i = 1 : n do
    for j = 1 : n do
        path_ij = edgeCost(i,j);
for k = 1 : n do
    for i = 1 : n do
        for j = 1 : n do
            path_ij = min(path_ij, path_ik + path_kj);
```

avoided unnecessary work by using *geodetic semiring*, an instance of the closed semiring generalization for shortest paths [2]. We briefly sketch the solution here for the sample graph shown in Figure 1. First, from the adjacency matrix A, we create a new relation matrix $R = [(d_{u,v}, n_{u,v})]$, where d is the geodesic between $(u, v) \in E$ and $n_{u,v}$ is the number of geodesics between u and v; initially

$$(d, n)_{u,v} = \begin{cases} (1,1) & \text{for} \quad (u,v) \in A \\ (\infty, 0) & \text{for} \quad (u,v) \notin A \end{cases} \tag{2}$$

Using this transformation on the adjacency matrix shown in Figure 1, we get the following matrix:

$$R = \begin{bmatrix} (\infty,0) & (1,1) & (\infty,0) & (\infty,0) \\ (\infty,0) & (\infty,0) & (1,1) & (1,1) \\ (1,1) & (\infty,0) & (\infty,0) & (\infty,0) \\ (1,1) & (\infty,0) & (\infty,0) & (\infty,0) \end{bmatrix}$$

From this matrix R, we compute the geodesic closure R^+ using the semiring $(R, \oplus, \odot, (\infty, 0), (0, 1))$, where:

$$(a,i) \oplus (b,j) = (min(a,b), \begin{cases} i & a < b \\ i+j & a = b \\ j & a > b \end{cases}) \tag{3}$$

$$(a,i) \odot (b,j) = (a+b, i \times j) \tag{4}$$

The key intuition here is that knowing the distance ($d_{u,v}$), and the number of shortest paths ($n_{u,v}$), it is easy to compute the number of shortest paths between (u, v) using the following equation:

$$n_{u,v}(t) = \begin{cases} n_{u,t} \times n_{t,v} & d_{u,t} + d_{t,v} = d_{u,v} \\ 0 & otherwise \end{cases} \tag{5}$$

For a complete proof of $(R, \oplus, \odot, (\infty, 0), (0, 1))$ being a geodesic semiring, please refer to Batagelj [7]. [3]

The modified Floyd-Warshall's algorithm that computes R^+ is given in Algorithm 2; after application of this algorithm to the matrix R, we get:

$$R^+ = \begin{bmatrix} (3,2) & (1,1) & (2,1) & (2,1) \\ (2,2) & (3,2) & (1,1) & (1,1) \\ (1,1) & (2,1) & (3,1) & (3,1) \\ (1,1) & (2,1) & (3,1) & (3,1) \end{bmatrix}$$

[3] $(R, \oplus, \odot, (\infty, 0), (0, 1))$ is a semiring *iff* all distances ($d_{i,j}$) are positive.

Algorithm 2: computeGeodeticSemiRing

Input: R: Relational matrix of graph $G(V, E)$

```
for k = 1 : n do
    for i = 1 : n do
        for j = 1 : n do
            distance = min(∞, d_ik + d_kj);
            if d_ij ≥ distance then
                count = n_ik × n_kj;
                if d_ij == distance then
                    n_ij = n_ij + count;
                else
                    n_ij = count;
                    d_ij = distance;
```

From R^+, it is simple to compute the BC of any node using equations 5 and 1. For example, BC(d) and BC(c) in G (from Figure 1) are both $1/2$ as there are exactly two shortest paths between b and a, and c and d are on these paths. In this method, computing R^+ takes $O(n^3)$ operations, and then computing BC of any node requires considering all pair-wise entries, which takes a further $O(n^2)$ computations. Therefore, the overall computational complexity of the algorithm is $O(n^3)$; space complexity is $O(n^2)$. This is too steep a price to pay in real-world graphs, which are sparse and large.

B. Graph Traversal Approach

The key to computing efficiently is to exploit the sparsity of the graph structure (as opposed to its adjacency matrix). This was the central theme on which Brandes [8] algorithm is based. Brandes recognized the recursive nature of BC computations that were exploited in equation 5. For unweighted graphs, the basic idea is to perform breadth-first searches (BFS) from all nodes. At each step, the closest set of vertices are added, and during this step, cumulative betweenness scores for all vertices are computed by using the predecessor relationship. Formally, let us define the *predecessors* of a vertex v on shortest paths from s to be

$$P_v(s) = \{u \in V : u, v \in E, d_G(s,v) = d_G(s,u) + (u,v)\} \tag{6}$$

Where $d_G(s, v)$ is the shortest path from s to v. Now, the number of shortest paths from s to v (σ_{sv}) is exactly 1 more than the sum of the number of shortest paths from s to each vertex $u \in P_v(s)$.

$$\sigma_{sv} = 1 + \sum_{u \in P_v(s)} \sigma_{su} \tag{7}$$

TODO: Check this equation again; I might have confused lemmas 3 and 5 from Brandes' algorithm. Therefore, in $O(m)$ time, we can compute all the shortest paths and number of shortest paths from a vertex s to every other vertex; that is, in $O(mn)$, we can compute all pairs and number of shortest paths. Furthermore, this solution only requires $O(m+n)$ space.

Algorithm 3: brandesBC

Input: $G(V, E)$: A graph

```
1  BC_v = 0, v ∈ V;
2  for s ∈ V do
3      S ← new(stack);
4      P_w ← ∅, w ∈ V;
5      σ_w ← 0, w ∈ V; σ_s ← 1;
6      D_w ← −1, w ∈ V; D_s ← 0;
7      Q ← new(queue);
8      enqueue(Q,s);
9      while Q ≠ ∅ do
10         v ← dequeue(Q);
11         push(S,v);
12         foreach w ← neighbor(v) do
13             if D_w < 0 then
14                 enqueue(Q,w);
15                 D_w = D_v + 1;
16             if D_w = D_v + 1 then
17                 append(P_w,v);
18                 σ_w = σ_w + σ_v;
19     δ_v ← 0, v ∈ V;
20     while S ≠ ∅ do
21         w ← pop(S);
22         foreach v ∈ P_w do
23             δ_v ← δ_v + (σ_v/σ_w) × (1 + δ_w);
24         if w ≠ s then
25             BC_w ← BC_w + δ_w;
```

The only thing left to do is to determine the contributions to the BC of each vertex from every other vertex. This information is already embedded in $P_v(s)$; there are $|P_v(s)|$ distinct predecessors in the shortest paths from s to v, and the number of shortest paths that each predecessor $P_{u,s}(i)$ is on is $\sigma_{su}(i)$. Now, using this information, we can compute the contribution of (s, v) to the BC scores of each of the predecessors in $P_v(s)$.

$$BC(u) = BC(u) + \frac{\sigma_{su}}{\sigma_{sv}}, u \in P_v(s) \tag{8}$$

In actual computation, the total number of shortest paths that go from s to any vertex $v \in V$ through vertex $t \neq s, v$ is accumulated for every vertex s, v, t and finally, the BC scores are computed. This final algorithm is given in Algorithm 3.

Let us consider the execution of Brandes's algorithm 3 on our sample graph G from Figure 1; let the start vertex be a.

$$\begin{aligned}
&Initialize\ BCs\ (line\ 1)\\
&BC_a \leftarrow BC_b \leftarrow BC_c \leftarrow BC_d \leftarrow 0\\
&Initialize\ for\ BFS\ from\ a\ (lines\ 3\ to\ 8)\\
&Q \leftarrow a, S \leftarrow \emptyset\\
&P_a \leftarrow P_b \leftarrow P_c \leftarrow P_d \leftarrow \emptyset\\
&\sigma_b \leftarrow \sigma_c \leftarrow \sigma_d \leftarrow 0, \sigma_a \leftarrow 1\\
&D_b \leftarrow D_c \leftarrow D_d \leftarrow -1, D_a \leftarrow 0
\end{aligned}$$

$$\begin{aligned}
&BFS\ from\ a\ (lines\ 9\ to\ 18)\\
&Q \leftarrow \emptyset, S \leftarrow (c, d, b, a)\\
&P_a \leftarrow \emptyset, P_b \leftarrow a, P_c \leftarrow b, P_d \leftarrow b\\
&\sigma_b \leftarrow \sigma_c \leftarrow \sigma_d \leftarrow \sigma_a \leftarrow 1\\
&D_a \leftarrow 0, D_b \leftarrow 1, D_c \leftarrow 2, D_d \leftarrow 2
\end{aligned}$$

$$\begin{aligned}
&Compute\ BC\ contributions\ (lines\ 19\ to\ 25)\\
&S \leftarrow \emptyset\\
&\delta_a \leftarrow 1, \delta_b \leftarrow 2, \delta_c \leftarrow \delta_d \leftarrow 0\\
&BC_a \leftarrow BC_c \leftarrow BC_d \leftarrow 0, BC_b \leftarrow 2
\end{aligned}$$

This change in the BC_b denotes that b is on *all* shortest paths from a to c and a to d. Similarly, when shortest paths from b are computed, the BC's are changed to:

$$BC_a \leftarrow 0, BC_b \leftarrow 2, BC_c \leftarrow BC_d \leftarrow 1/2$$

These changes in BC_c and BC_d indicate that there are two shortest paths from b to a, one each going through c and d respectively. Similarly, when shortest paths from c are computed, the BC's are changed to:

$$BC_a \leftarrow 2, BC_b \leftarrow 3, BC_c \leftarrow BC_d \leftarrow 1/2$$

These changes in BC_a and BC_b are because a lies on two shortest paths from c (to b and d) and b on the shortest path from c to d. Finally, when shortest paths from d are computed, BC's are changed to:

$$BC_a \leftarrow 4, BC_b \leftarrow 4, BC_c \leftarrow BC_d \leftarrow 1/2$$

These changes in BC_a and BC_b are because a lies on two shortest paths from d (to b and c) and b on the shortest path from d to c.

1) Parallelization: TODO: Add information about parallelization.

C. Hybrid Formulation

Both from a computational and spatial standpoint, there are several inefficiencies in Algorithm 3. First, notice that the queue Q and the stack S can be combined into one array structure that when accessed from one end acts as a queue, and from the other end, acts as a stack. Second, the distance array D is redundant in a BFS computation as *all* the shortest

Algorithm 4: hybridBC

Input: A: Adjacency matrix of an unweighted graph G
Input: n: Dimension of A ($n \times n$)
Input: $nVerts$: Number of BFS' to perform at once

// **We assume that nVerts is divisible by** n
1 $nPasses = \frac{nVerts}{n}$;
2 $BC \leftarrow$ matrix(1,n,0);
3 **foreach** $p \in (1 : nPasses)$ **do**
4 $\quad BFS \leftarrow \emptyset$;
5 $\quad batch = ((p-1) \times nVerts + 1)$:min($p \times nVerts, N$);
6 $\quad nsp \leftarrow$ matrix($nVerts$,n, 0);
7 $\quad$ **foreach** $row \in (1 : nVerts)$ **do**
8 $\quad\quad nsp(row, batch(row)) \leftarrow 1$;
9 $\quad depth \leftarrow 0$;
10 $\quad$ eltWiseAssign($fringe$,extractRows(A,$batch$));
$\quad$ // **BFS search for all vertices in current batch**
11 $\quad$ **while** nnzExists($fringe$) **do**
12 $\quad\quad depth \leftarrow depth + 1$;
13 $\quad\quad nsp \leftarrow$ eltWiseAdd(nsp,$fringe$);
14 $\quad\quad BFS(depth) \leftarrow fringe$;
15 $\quad\quad fringe \leftarrow$ matMult($fringe$,A);
$\quad\quad$ // **Reset entries for already visited vertices**
16 $\quad\quad fringe \leftarrow$ eltWiseMult($fringe$,not(nsp));
$\quad$ // **Pre-compute BC updates for all but source vertices**
17 $\quad BC_{updt} \leftarrow$ matrix($nVerts$,n,1);
18 $\quad nsp^{inv} \leftarrow$ eltWiseInvert(nsp);
$\quad$ // **Compute BC updates for all but source vertices**
19 $\quad$ **for** $d \in (depth, depth-1, ..., 2)$ **do**
$\quad\quad$ // **Compute child weights**
20 $\quad\quad weights_1 \leftarrow$ eltWiseMult($BFS(d)$,nsp^{inv});
21 $\quad\quad weights \leftarrow$ eltWiseMult($weights_1$,BC_{updt});
$\quad\quad$ // **Apply child weights**
22 $\quad\quad temp_1 \leftarrow$ matMult(A,$weights^T$)^T;
$\quad\quad$ // **Sum them up over parents**
23 $\quad\quad temp_2 \leftarrow$ eltWiseMult($BFS(d-1)$,nsp);
$\quad\quad$ // **Apply weights based on parents values**
24 $\quad\quad temp_3 \leftarrow$ eltWiseMult($temp_1$,$temp_2$);
25 $\quad\quad BC_{updt} \leftarrow$ eltWiseAdd(BC_{updt},$temp_3$);
$\quad$ // **Update BC scores from each source vertex's BFS**
26 $\quad$ **for** $row \in (1 : nVerts)$ **do**
27 $\quad\quad BC \leftarrow$ eltWiseAdd(BC,$BC_{updt}(row, :)$);
$\quad$ // **Subtract additional values added by precomputation**
28 $BC \leftarrow$ eltWiseAdd(BC,matrix(1,n,$-nPasses$));

paths to a node must be reached during the same *BFS fringe's* expansion. Any paths that reach a node during a later fringe are *not* shortest paths; hence, we can replace D with a bit-array with one bit per node to denote that its shortest path was reached during a particular fringe expansion. Third, consider the set of predecessors P_v of a particular vertex v; since G is unweighted, these predecessors must have themselves been discovered during the previous fringe expansion. That is, to compute the set of predecessors for a particular vertex v, we can look at all its incoming *discovered* edges; we need not store P explicitly. These three optimizations decrease the amount of space needed to execute Brandes' algorithm by a significant constant factor. The final optimization tries to perform BFS fringe expansions in terms of sparse matrix multiplication with the *fringe* vector (BLAS-2 kernel); this allows exploration of one full fringe in one operation rather than looping over all the neighbors of a particular node (lines 12 to 18 in Algorithm 3). The initial fringe vector contains just one entry, which corresponds to the start/source vertex. Furthermore, shortest paths from multiple sources can be determined together by replacing the sparse matrix vector product operation with a sparse matrix matrix operation (BLAS-3 kernel); this is a standard trick in linear algebra called *blocking*. However, note that the space requirements increase linearly with the block size. After performing all these optimizations, we end up with algorithm 4, which is given as the sample MATLAB implementation for the SSCA benchmarks [4]. For example, in our sample graph G from Figure 1, the first fringe expansion starting out from vertices a and c can be expressed as follows:

$$\begin{bmatrix} a^T & b^T & c^T & d^T \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} a^T & c^T \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} b^T & a^T \\ 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

For formatting purposes, we have shown right multiplication by the starting vectors, which requires transposing the original adjacency matrix. As can be seen, by starting out with BFS expansion from a and c, we end up with the new fringe b for a, and a for c, respectively. In fact, since this is always the case, we can forgo this computation and simply consider the first fringe to be the starting vertex's adjacency row. Please note that in an actual, high performance implementation, all matrices are sparsely represented to save space. Let us now work through Algorithm 4 for the sample graph G from Figure 1; for simplicity, we process only one vertex at a time. As before, let us start processing from vertex a.

$$(Initialization: \ lines \ 1 \ to \ 10)$$
$$nPasses \leftarrow 4, BC \leftarrow \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$
$$BFS \leftarrow \emptyset, batch \leftarrow 1:1, depth \leftarrow 0$$
$$nsp \leftarrow \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}, fringe \leftarrow \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$$

Here, we have initialized $nsp(a)$ to be 1 and set the fringe to

be a's adjacency, which is b.

$$(BFS\ search\ from\ a:\ lines\ 11\ to\ 16,\ first\ pass)$$
$$depth \leftarrow 1, BFS(1) \leftarrow \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$$
$$nsp \leftarrow \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}, fringe \leftarrow \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$

At the end of the first pass, we have expanded vertex b and discovered vertices c and d; nsp now indicates that we have found one shortest path (a, b).

$$(BFS\ search\ from\ a:\ lines\ 11\ to\ 16,\ second\ pass)$$
$$depth \leftarrow 2, BFS(2) \leftarrow \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$
$$nsp \leftarrow \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}, fringe \leftarrow \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

At the end of the second pass, we have explored c and d; at this point there are no more undiscovered vertices and the BFS search from a ceases. nsp indicates that we discovered 3 shortest paths, (a, b), (a, c), and (a, d); the path (a, a) is not useful in computing BC scores when the start vertex is a itself. Now, we move on to updating the BC scores for all vertices based on our exploration from vertex a.

$$(Initialize\ for\ BC\ updates:\ lines\ 17\ and\ 18)$$
$$BC_{updt} \leftarrow \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}, nsp^{inv} \leftarrow \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$

As in Brandes 3), we move back from the final fringe to the first; in our example, this is $depth = 2$.

$$(Compute\ and\ update\ child\ weights:\ lines\ 20\ to\ 25)$$
$$weights_1 \leftarrow \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}, weights \leftarrow \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix}$$
$$temp_1 \leftarrow \begin{bmatrix} 0 & 2 & 0 & 0 \end{bmatrix}, temp_2 \leftarrow \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$$
$$temp_3 \leftarrow \begin{bmatrix} 0 & 2 & 0 & 0 \end{bmatrix}, BC_{updt} \leftarrow \begin{bmatrix} 1 & 3 & 1 & 1 \end{bmatrix}$$

Next, we update the BC scores for each vertex with BC_{updt}.

$$(Update\ the\ BC\ scores:\ lines\ 26\ and\ 27)$$
$$BC \leftarrow \begin{bmatrix} 1 & 3 & 1 & 1 \end{bmatrix}$$

Notice that in Algorithm 4, we pre-compute BC_{updt} to be 1 for all vertices; hence, even though at the end of the BFS exploration for vertex a, BC scores of all vertices show a value > 0, the only true change is that of vertex b, which lies on the path from a to both c and d. For brevity, we will now simply list the values of BC after the end of BFS exploration for each of the remaining vertices:

$$After\ BFS\ from\ b,\ BC \leftarrow \begin{bmatrix} 2 & 4 & 2.5 & 2.5 \end{bmatrix}$$
$$After\ BFS\ from\ c,\ BC \leftarrow \begin{bmatrix} 5 & 6 & 3.5 & 3.5 \end{bmatrix}$$
$$After\ BFS\ from\ d,\ BC \leftarrow \begin{bmatrix} 8 & 8 & 4.5 & 4.5 \end{bmatrix}$$

The final trick to get the accurate BC scores is to subtract the number of passes as each pass adds 1 to each vertex's BC score (Algorithm 4, line 28). Note that without initializing BC_{updt} with ones, line 24 in Algorithm 4 would not have computed $weights$ accurately; therefore, it is critical to initialize BC_{updt} with ones.

$$Subtract\ 4\ from\ BC\ (line\ 28):\ BC \leftarrow \begin{bmatrix} 4 & 4 & 0.5 & 0.5 \end{bmatrix}$$

This gives the final and accurate BC scores for all the vertices.

1) Parallelization: Algorithm 4 is expressed mostly in terms of computations on sparse matrices with sparse or dense vectors, which makes it especially suitable for parallelization. In this segment, we discuss some of the factors that influence parallelization of Algorithm 4. First, let us recap that we are interested mostly in social network analysis and the real-world graphs in this particular domain are similar to RMAT graphs (see Section IV). Briefly, these graphs follow the power-law, have low average connectivity and low diameter. First, let us recap that Algorithm 4 implements BFS as a matrix-matrix multiplication (line 15).

a) Space Analysis: Table I lists the space requirements of each of the variables used in Algorithm 4; we will now analyze each variable separately. BC is computed for each vertex $v \in V$, and hence, the row-vector BC is represented as a dense array of double's; for simplicity, we assume that this is the case for BC_{updt} as well; The adjacency matrix of the graph A for RMAT graphs is sparse; therefore, it can be represented in either CSR, CSC, DCSR, or DCSC [10] formats. However, as we are exclusively dealing with *unweighted* graphs in this paper, we can drop the edge weights completely; this saves us m entries in A alone. For similar reasons, we can drop the edge weights from $fringe$ and BFS. Notice that $\sum_{i=1}^{diameter(G)} BFS(i) = m$; that is, the total number of non-zero entries in BFS is exactly the same as the number of edges in G. Therefore, by not storing edge weights, we save an additional m entries for a total saving of $2m$ (potentially integers). A stark contrast is detected when it comes to the variable nsp, an array that stores the number of shortest paths from each source vertex to all other vertices. In an RMAT graph, there exists a connection between any two vertices with a high probability; that is, nsp will be dense. Therefore, storing nsp in regular dense format, which requires $nVerts \times n$ integers is beneficial over storing them in compressed formats that require explicit representation of the row and the column numbers, thereby saving us This saves us $\approx 2n$ space.

b) Kernel Operations: Table II depicts the different kernels used in Algorithm 4; As can be seen, other than extractRows(), nnzExists(), and matMult(), all the other operations are completely *local*. That is, these operations do not require and communication in the case of a distributed-memory implementation. One way to parallelize on both shared- and distributed-memory machines is to parallelize the operations that are depicted in Table II; Buluc [10] parallelized these operations for distributed memory and for shared memory [11], but not for a multi-level architecture consisting of both shared- and disributed-memory machines.

2) Shortcomings: Algorithm 4 works *only* for unweighted graphs. When the edges are weighted, the "shortest path" from one node to another is determined not only by the number of hops that need to be taken, but also the cost/weight of each hop. Consequently, many changes are needed to algorithm 4 in order to make it work for weighted graphs. For example, line 16, which ensures that each vertex is visited only once is incorrect for weighted graphs as a path with greater number of hops, but lower cost can still be found. In essence, we

Variable	Shape	Space Requirement	Representation	Sparsity (RMAT-specific)
BC	Row-vector	n	double[]	Dense
A	Square-matrix	$O(m)$	CSR/CSC(<int, int>)	Hyper-sparse
$fringe$	Rectangular-matrix	$O(nVerts \times m)$	CSR/CSC(<int, int>)	Sparse
BFS	Rectangular-matrix	$O(m)$	CSR/CSC<int, int>[]	Sparse
nsp	Rectangular-matrix	$nVerts \times n$	int[]	Dense
nsp^{inv}	Rectangular-matrix	Computed on the fly	int[]	Dense
BC_{updt}	Rectangular-matrix	$nVerts \times n$	double[]	Dense
$weights_1$	Rectangular-matrix	Computed on the fly	—	—
$weights$	Rectangular-matrix	$O(nVerts \times n)$	double[]	Sparse to Hyper-sparse
$temp_1, temp_2, temp_3$	Rectangular-matrix	Computed on the fly	—	Sparse

TABLE I
TABLE DEPICTING THE SPACE REQUIREMENTS OF EACH VARIABLE IN ALGORITHM 4. SPARSITY IS SPECIFIC TO RMAT GRAPHS (SEE SECTION IV. HYPER-SPARSE MEANS THAT THERE ARE LESS NON-ZEROS THAN THE NUMBER OF COLUMNS/ROWS. CSR REFERS TO THE COMPRESSED SPARSE ROW FORMAT AND CSC TO THE COMPRESSED SPARSE COLUMN FORMAT.

Kernel	Description	Type
matrix(m,n, init)	Create an $(m \times n)$ matrix initialized to *init*	local
extractRows(A,rowList)	Extract a submatrix formed by the rows in $rowList$	global
eltWiseAssign(A,B)	Assign the matrix A to the matrix B	local
eltWiseAdd(A,B)	Element-wise addition of two (sparse) matrices	local
eltWiseMult(A,B)	Element-wise multiplication of two (sparse) matrices	local
nnzExists(A)	Check for the presence of a non-zero in the matrix A	global
matMult(A,B)	Multiply two (sparse) matrices	global

TABLE II
A TABLE DEPICTING THE DIFFERENT MATRIX KERNELS IN ALGORITHM 4 AND THEIR FEATURES.

degenerate to Floyd-Warshall's $O(n^3)$ algorithm; therefore, Algorithm 4 is unsuitable for weighted graphs. However, for benchmarking purposes, Algorithm 4 is sufficient; for example, kernel 4 of HPC Graph Analysis Benchmarks [4] only require computation of unweighted betweenness centrality.

D. Approximating BC

Real-world graph sizes are ever increasing and BC computations are expensive; consequently, approximate measures for BC have been explored. There are three prominent works in this regard. First, Eppstein and Wang [14] describe a randomized approximation algorithm for estimation of *closeness centrality* in weighted graphs. [4] Their method (RAND) randomly chooses k vertices one by one and use these vertices as start vertices for the single source shortest paths; that is, instead of solving all sources shortest paths problem, k sources shortest paths problem is solved. They further proved that for $k = \Theta(\frac{log(n)}{\epsilon^2})$, their algorithm approximates closeness centrality with an additive error of $\epsilon\Delta_G$, where Δ_G is the diameter of the graph, and ϵ is a small constant. Brandes and Pich [9], after experimenting with various deterministic strategies for selecting source vertices for approximating BC, concluded that a random sampling strategy was superior. Bader et al. [5] presented an adaptive sampling technique that approximates the BC of *a given vertex*. They conclude that for $0 < \epsilon < \frac{1}{2}$, if the centrality of a vertex v is $\frac{n^2}{t}$ for some constant factor $t \geq 1$, then with a probability $\geq (1 - 2\epsilon)$, its centrality can be estimated within a factor of $\frac{1}{\epsilon}$ by using only ϵt samples of source vertices.

[4]Closeness centrality (CC) of a vertex is a global metric that measures the the distance of a vertex to all other vertices in the graph. Formally $CC_v = \frac{1}{\sum_{u \in V} d(v,u)}$.

III. SPARSE MATRIX OPERATIONS

A. Matrix-Matrix Multiply

TODO: Add text here about Buluc's work.

B. Element-wise Matrix Operations

TODO: Add text here about Buluc's work.

IV. RMAT GRAPHS

As important as it is to be able to compute social networking metrics such as BC accurately and efficiently, it is also necessary to be able to generate synthetic graphs that mimic real-world graphs for testing and comparative purposes. In recent years, RMAT graphs [12] have been widely adopted for generating synthetic graphs that mimic the power-law characteristics demonstrated by several real-world graphs. The idea behind RMAT graphs is simple: a graph $G(V, E)$ is considered to be a boolean adjacency matrix A, where $A_{ij} = 1$ implies the prescence to an edge between vertices (i, j). The edge connections are determined using an appropriate psuedo-random number generator (PRNG) with range $[0, 1)$ and four carefully chosen weights a, b, c, and d ($a+b+c+d = 1$). These four weights reflect the probability that any given edge falls within one of the four equal sized partitions that the adjacency matrix is recursively divided into. Typically, $a \geq b$, $a \geq c$, $a \geq d$; in many real-world scenarios, $(a + b)/(c + d) = 3$. Partitions a and d represent separate groups of nodes, and b and c represent cross-links between these groups. Recursion ends upon reaching a 1×1 cell (at some position (i, j), which

Algorithm 5: RMAT

Input: n: Number of nodes in graph 2^n
Input: k: The total number of edges $(k \times 2^n)$
Input: a: Prob that an edge lies in first quadrant
Input: b: Prob that an edge lies in second quadrant
Input: c: Prob that an edge lies in third quadrant
Input: d: Prob that an edge lies in fourth quadrant

1 $N \leftarrow 2^n, M \leftarrow k \times N$;
2 $rowIndices \leftarrow colIndices \leftarrow$ matrix(1,M,1);
3 $ab \leftarrow (a+b), cNorm \leftarrow \frac{c}{(c+d)}, aNorm \leftarrow \frac{a}{(a+b)}$;
4 **for** $i \in 0..(n-1)$ **do**
5 $\quad lowerHalf \leftarrow$ rand(M,1)$> ab$;
6 $\quad upperHalf \leftarrow$ eltWiseNot($lowerHalf$);
7 $\quad cQuadrant \leftarrow$ eltWiseMult($cNorm$,$lowerHalf$);
8 $\quad aQuadrant \leftarrow$ eltWiseMult($aNorm$,$upperHalf$);
9 $\quad acQuadrant \leftarrow$ eltWiseAdd($aQuadrant$,$cQuadrant$);
10 $\quad rowBits \leftarrow upperHalf$;
11 $\quad colBits \leftarrow$ rand(M,1)$> acQuadrant$;
12 $\quad currentRows \leftarrow$ eltWiseMult(2^i,$rowBits$);
13 $\quad currentCols \leftarrow$ eltWiseMult(2^i,$colBits$);
14 $\quad rowIndices \leftarrow rowIndices + currentRows$;
15 $\quad colIndices \leftarrow colIndices + currentCols$;
16 $rmat \leftarrow$ sparse($rowIndices$,$colIndices$);
17 **return** $rmat$;

cannot be sub-divided any further. If, by chance, that cell is already occupied, the duplicate is either discarded. [5] Algorithm 5 lists the steps involved in generating an RMAT graph using an array syntax; we now briefly discuss the method involved. The input to the algorithm is (n, k, a, b, c, d), and it outputs an adjacency matrix of a graph G with $N = 2^n$ vertices and $M = (k \times 2^n)$ edges. We iterate over n passes determining M separate edges during each iteration; as mentioned before, duplicate edges are either discraded when creating the final adjacency matrix or are added up to represent edge weights. As seen in lines 5 and 6, we generate M random numbers and determine if they are in the lower $(c + d)$ or the upper half $(a+b)$; these form the rows. The columns are determined by another sample of M random numbers; only this time, we chose to place them in either the $(a + c)$ half or $(b + d)$ half of the adjacency matrix.

1) Parallelization: To parallelize algorithm 5, let us first understand how the edges are generated. The for loop (lines 4 to 15) is repeated n times, where 2^n is the number of vertices. In each loop iteration, M boolean values are generated for row numbers (i in a_{ij}) and M boolean values are generated for column numbers (j in a_{ij}), where $M = k \times 2^n$ is the number of edges. At iteration l, these boolean values are used to set the l^{th} bit of the M row and column numbers that correspond to the M edges. At the end of n iterations, each of the n bits required to represent 2^n vertices are set to either 1 or 0 for each of the M rows and columns. There are two strategies for parallelization that can be pursued. First, we can parallelize the execution of the n iterations, taking care to see that each parallel execution stream knows its global iteration numbers — this is necessary to ensure that edges generated connect vertices globally rather than locally. However, considering that we have only n iterations (e.g., a graph with 2^{64} nodes is an order of magnitude greater than *exascale*), this approach only gives us n-way parallelism. A better approach is to parallelize across generation of the M edges, which gives us more parallelism; that is, each processor generates a portion of the graph. However, since there might still be multiple edges, a global reduction operation is required at the very end. Note that if there is sufficient computation power, one can always parallelize both the n iterations and the generation of the M edges.

[5] Weighted RMAT graphs are generated by accumulating duplicate edges that fall in the same cell. For example, if an l edges happen to fall within the same cell (i, j), the edge weight of (i, j) is set to be l.

V. Related Work

Parallelizing BC became a key application for super computers to demonstrate ever since the publication of the HPC Graph Analysis Benchmarks [4]. In this section, we will attempt to enumerate those efforts that are relevant in that they deal with parallelism, scalability and betweenness centrality for RMAT [12] graphs. Bader et al. [6] developed shared-memory parallel algorithms for various centrality metrics, including BC. In their work, they were able to compute BC metrics on graphs with 3 million nodes and 16 million edges, and further claimed to be able to scale to billions of nodes and edges. In this work, they exploit fine-grained parallelism by involving multiple threads in finding shortest paths from a single source and coarse-grained concurrency by starting shortest path searches from multiple sources simultaneously. However, their approach does not exploit the vast amount of distributed-memory parallelism that is on offer in most modern clusters and is not space efficient. Madduri et al. [17] presented a lock-free parallel algorithm for computing BC, which significantly reduced synchronization overhead involved in merging BC scores from various threads. Their key idea was to replace predecessor set of a vertex $P_v(s)$ with a successor set; this allows more parallelism as successor sets are private, as opposed to predecessor sets, which have to be shared. Using their approach, they were able to *approximate* BC scores for a RMAT graph with billions of edges in a reasonable amount of time. Like their previous efforts, they focussed exclusively on shared-memory architectures. Yang and Leonardi [21] developed a distributed memory solution for BC, which achieves almost linear speedup (for at least 32 processors). However, their approach relied on replicating the graph on all processors, which is not space efficient and hence, not scalable. Edmonds et al. [13] devised a space-efficient parallel BC computation targeted towards distributed-memory machines. Their algorithm computes BC for both weighted and unweighted graphs, which are partitioned *a priori* with a space complexity of $O(|V| + |E|)$ building on the Δ-stepping single source shortest path solution by Meyer [18].

Although Edmonds' algorithm does not preclude multi-level parallelism, it does not demonstrate multi-level parallelism. Also, they achieved limited scaling for RMAT graphs — scaling was non-existant for over 16 processors. Furthermore, all the efforts described above try to parallelize based on the graph-traversal approach that was described in Section II-B. It is important to note that there have been several noteworthy efforts to parallelize BFS traversal of a graph itself. Notably, Agrawal et al. [1] were able to achieve impressive numbers by exploiting the inherent non-uniform nature of modern multi-core processors.

To our knowledge, there has been only one effort (Buluc [10]) that is based on the hybrid approach of using linear algebraic primitives for BFS exploration (see Section II-C). Buluc successfully used BC as a case study for the need to have combinatorial BLAS operations for graph algorithms by scaling up to 1225 processors on graphs with 32 million vertices and ≈ 256 million edges. Note that exact BC was not computed, but merely approximations were performed by randomly picking certain vertices as source vertices. This effort, however promising, still does not take advantage of the multiple levels of fine-grained parallelism available on modern multi-core processors.

VI. Open Problems

- Heirarchical approaches as all the existing approaches are for either distributed memory or are shared memory.
- Out-of-core approaches as there is currently a notion that everything has to fit in memory; this is not necessarily going to be true. We need to be thinking out of core.
- Graph partitioning is important as we cannot afford to replicate the graph at each node.
- Approximate metrics are needed for computing betweeness centrality quickly.
- Graph layout algorithms are needed to ensure that we do not incur too many cache misses.
- If the graph is replicated, can we use any pure task parallelism techniques to solve the problem efficiently?

References

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable Graph Exploration on Multicore Processors. *SC Conference*, 0:1–11, 2010.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1974.

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

[4] D. Bader, J. Gilbert, J. Kepner, and K. Madduri. HPC Graph Analysis Benchmarks. http://www.graphanalysis.org/benchmark.

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

[6] David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Intl. Conf. on Parallel Processing*, pages 539–550, Washington, DC, 2006.

[7] V. Batagelj. Semirings for socal network analysis. *Journal of Mathematical Society*, 19(1):53–68, 1994.

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

[9] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. In *International Journal Of Bifurcation and Chaos, Special Issue On Complex Networks' Structure And Dynamics*, 2007.

[10] Aydin Buluc. *Linear Algebraic Primitives for Computation on Large Graphs*. PhD thesis, University of California, Santa Barbara, 2010.

[11] Aydın Buluç, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

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

[13] Nicholas Edmonds, Torsten Hoefler, and Andrew Lumsdaine. A Space-Efficient Parallel Algorithm for Computing Betweenness Centrality in Distributed Memory. Springer, December 2010.

[14] David Eppstein and Joseph Wang. Fast Approximation of Centrality. *Journal of Graph Algorithms and Applications*, 8:228–229, 2001.

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

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

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

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

[19] Eunice E. Santos, Long Pan, Dustin Arendt, and Morgan Pittkin. An Effective Anytime Anywhere Parallel Approach for Centrality Measurements in Social Network Analysis. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 4693–4698, october 2006.

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

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