# Efficient Extraction of High Centrality Vertices in Distributed Graphs

Alok Gautam Kumbhare
Computer Science Department
University of Southern California
Los Angeles, CA USA
email: kumbhare@usc.edu

Marc Frincu, Cauligi S. Raghavendra and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, CA USA
email: {frincu, raghu, prasanna}@usc.edu

***Abstract*—Betweenness centrality (BC) is an important measure for identifying high value or critical vertices in graphs, in variety of domains such as communication networks, road networks, and social graphs. However, calculating betweenness values is prohibitively expensive and, more often, domain experts are interested only in the vertices with the highest centrality values. In this paper, we first propose a partition-centric algorithm (MS-BC) to calculate BC for a large distributed graph that optimizes resource utilization and improves overall performance. Further, we extend the notion of approximate BC by pruning the graph and removing a subset of edges and vertices that contribute the least to the betweenness values of other vertices (MSL-BC), which further improves the runtime performance. We evaluate the proposed algorithms using a mix of real-world and synthetic graphs on an HPC cluster and analyze its strengths and weaknesses. The experimental results show an improvement in performance of upto 12x for large sparse graphs as compared to the state-of-the-art, and at the same time highlights the need for better partitioning methods to enable a balanced workload across partitions for unbalanced graphs such as small-world or power-law graphs.**

## I. Introduction

With the development of massive online social networks and graph structured data in various areas such as biology, transportation, and computer networks, analyzing the relationship between the various entities forming the network is increasingly important as it allows complex behavior in these large graphs to be explained. The relative location of each vertex in the network helps identify the main connectors, where the communities are, and who is at their core. Centrality indices measure the importance of a vertex in a network [1] based on their position in the network. Betweenness centrality (BC) is one such useful index that is based on the number of shortest paths that pass through each vertex. Formally, BC for a vertex $v$ is defined as the ratio of the sum of all shortest paths that pass through the node $v$ and the total number of shortest paths in the network [2]:

$$BC(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \qquad (1)$$

A simple approach to compute BC is in two steps: first, the shortest paths between all pairs is computed, and second, the length and number of pair-dependencies (i.e. $\delta_{st}(v)$) are computed and hence the betweenness centrality. This has $\mathcal{O}(n^3)$ time, and $\mathcal{O}(n^2)$ space complexity.

Brandes [2] proposed a faster algorithm with $\mathcal{O}(n+m)$ space complexity and $\mathcal{O}(nm)$ time complexity for unweighted graphs (and $\mathcal{O}(nm+n^2log(n))$ for weighted graphs). The main idea behind Brandes' algorithm is to perform $n$ shortest path computations and to aggregate all pairwise dependencies without explicitly iterating through $n^2$ shortest paths to calculate BC for each vertex. To achieve this Brandes noticed that the dependency value $\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$ of a source vertex $s$ on a vertex $v$ satisfies the following recursive relation:

$$\delta_s(v) = \sum_{w, v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(1 + \delta_s\left(w\right)\right) \qquad (2)$$

where $pred(s, w)$ represents the set of predecessors of $w$ in the shortest path from $s$ to $w$. By using Equation 2 the algorithm revisits the nodes starting with the farthest one from $s$, and accumulates the dependency values.

It can be easily seen that the algorithm exposes parallelism at multiple levels. Mainly the shortest path exploration from different source vertices as well as the computation of the individual shortest paths from a given source can be done in parallel. The first shared memory parallel algorithm that exploits the former for exact BC evaluation was proposed by Bader, et al. [3] and later improved [4] by reducing the synchronization overhead by eliminating the predecessor multisets. While this approach has been proven to be much faster, it is architecture specific and assumes a shared memory architecture. On the other hand we explore BC algorithms for large graphs partitioned on commodity clusters as opposed to tightly coupled high performance computing systems since commodity clusters are more accessible especially due to emergence of cloud computing paradigm.

In this paper we propose a partition centric bulk synchronous parallel (BSP) algorithm (§III) that exhibits a more efficient utilization of distributed resources and at the same time shows a significant reduction in the number of BSP supersteps as compared to the vertex centric model. To achieve this we rely on a partition centric approach that we and others have proposed earlier [5], [6] and perform an initial partitioning of the graph which allows us to compute shortest paths locally before communicating with other nodes, thus reducing

978–1–4799–6233–4/14/$31.00 

the communication overhead. Following the conclusions of Kourtellis et. al.[7] we focus on approximate computations and extract the top $k$ vertices with highest BC values in order to speed-up the algorithm. We also propose a light weight pre-processing step that prunes the graph by removing a subset vertices and edges (§IV) that have minimum contribution to the centrality indices of other vertices thus further improving the algorithm runtime. We demonstrate the scalability and tradeoffs for the proposed algorithm as well as the augmented leaf compression version using both real-world and large synthetic data sets (§V). Finally, We also study the impact of partitioning imbalance on different types of graphs and highlight the need for better partitioning techniques.

## II. Related Work

Betweenness centrality is a powerful metric used in complex graphs such as social networks as it allows us to identify the key nodes in a graph. These key nodes allow many nodes to quickly connect with each other and at the same time they are critical points in the graph, since by eliminating them we could possibly split the graph. As a result much work has been done towards efficiently identifying them since they were first introduced by Freeman [1].

One important question that needs to be answered is "what does centrality refer to?". Borgatti [8] discusses other possible definitions that look at the types of information flows and the frequent paths they take when searching for centrality nodes. While in this paper we restrict ourselves to geodesic paths we argue that our algorithm can be easily applied to any kind of centrality definition as long as the initial partitioning takes that into account.

Brandes' BC algorithm is highly parallelizable due to the way shortest paths are explored and computed. Bader et al. [3] have proposed a shared memory algorithm for exact betweenness evaluation on a dedicated multi-threaded Cray XMT system [9]. Improved versions with better memory utilization [10] and processing and communication overhead [4] have also been proposed. While these algorithms scale on dedicated systems they are not suited for large distributed environments. This is especially important since with the advent of cloud computing the graph datasets may be distributed across compute nodes with relatively higher network latency.

Distributed vertex[11] and partition centric algorithms [12] have also been proposed which exploit the parallelism in computation of the shortest paths. Houngkaew et al. [13] extends the space efficient partition centric approach proposed by Edmonds et. al. [12] by further exploiting the per compute node level parallelism to accumulate dependencies (with explicit mutual exclusion similar to Bader et. al.[3]). While both these approaches are space efficient, we observe that only the compute nodes on the fringe of the shortest paths computation are active and hence leads to under-utilization of the cluster. Our approach improves upon them by computing multiple shortest paths starting from different compute nodes in parallel, albeit, at the expense of additional space requirements.

Because computing BC is extremely costly due to the all-pair shortest path calculations approximate versions have been proposed. The main motivation is that good approximations can be an acceptable alternative to exact scores. Bader et al. [14] proposed an efficient approximate algorithm based on an adaptive sampling and more recently Kourtellis et al. [7] proposed identifying the top $k$ high betweenness nodes, arguing that the exact value is irrelevant for the majority of applications and that it is sufficient to identify categories of nodes of similar importance. In this paper we too address the aspect of vertices with high BC values and further improve the performance through graph pruning.

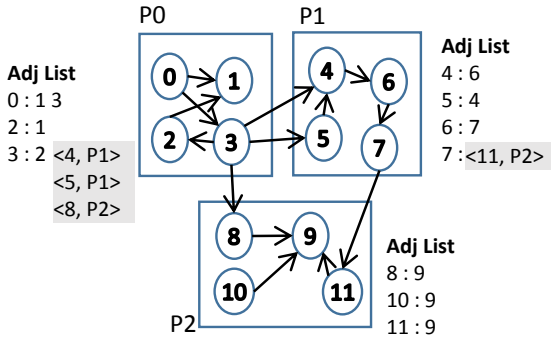
## III. Partition-Centric Betweenness Centrality

Vertex centric programming models using BSP (e.g. Pregel) [15], [16] for large distributed graph analytics have improved the programming simplicity as well as provided a simple approach for parallelizing graph algorithms. They have been shown to be useful for a large class of graph algorithms, but are prone to performance bottlenecks due to several aspects: low computation to communication ratio because of relatively small amount of work performed per vertex, synchronization required at every superstep and the fact that vertex locality and graph partitioning is ignored by the programming model. Further, the vertex centric models tend to require a large number of supersteps to propagate the information across the graph which further adds to the overall runtime.

To overcome these issues, we and others have proposed partition (or subgraph) centric programming models [17], [5] with the goal of improving the computation to communication ratio by allowing the workers to process an entire partition at a time without any explicit message passing and thus reducing the overhead. This also tends to reduce the number of supersteps required since the messages exchanged between the partitions are available to all the vertices in that partition. The number of supersteps to propagate information is reduced from $O(\text{graph diameter})$ to $O(\text{num. of partitions})$ and is reflected in several graph algorithms such as single source shortest paths [5] which forms the basis of the BC algorithm.

Figure 1a shows a sample graph partitioned into three nodes showing its adjacency list as well as remote vertices that act as pointers to partitions owning those vertices, while Fig. 1b shows a generic approach and programming model for partition centric graph algorithms. The idea is to perform local-computations on the given graph partition, send messages to remote vertices if required and incrementally continue the local work when new messages are received from other partitions. The Do-Local-Compute and Process-Messages functions (fig. 1b) form a core of any partition centric algorithm.

With the development of such new programming paradigms, it is important to explore their strengths and weaknesses with respect to basic kernels in the domain. BC is one such algorithm in graph analytics and we propose a partition centric BC algorithm which improves CPU utilization by increasing parallelism and reduces the number of required supersteps. While Edmonds et. al. [12] proposed a partition centric algorithm (referred to as $\Delta$S-BC), they focus on efficient space utilization and use a $\Delta$-stepping version of the the Single Source Shortest Paths (SSSP). The $\Delta$-stepping shortest paths improves utilization by doing a look-ahead and estimating the distance to the vertices within the look-ahead window using a label correcting approach and updating the distance whenever the estimated value is found to be incorrect. While this approach improves the performance of a single run




(a) Sample graph partition with remote vertices.

```
1: procedure PARTITION-COMPUTE(Partition Graph G, messages<
   targetvertex, list < values >>)
2:     if superstep = 0 then
3:         DO-LOCAL-COMPUTE(G)
4:     else
5:         PROCESS-MESSAGES(messages)
6:         CONTINUE-LOCAL-COMPUTE(G)
7:     end if
8:     for r: remote-vertices do
9:         if condition then
10:            SEND(owner(r), r, msg)
11:        end if
12:    end for
13:    if condition then
14:        VOTE-TO-HALT
15:    end if
16: end procedure
```

(b) Programming Abstraction.

Fig. 1: Partition-centric programming model

of the shortest path algorithm, it seems to be limited because the parallization is achieved only by expanding the fringe of the shortest paths algorithm and does not exploit the fact that multiple shortest paths need to be calculated for computing the centrality values.

We sacrifice in-memory space efficiency in favor of better work distribution among the available processors by running multiple shortest paths starting in different partitions which further increases the amount of local compute and improves the compute to communication ratio. Algorithm 1 (referred to as MS-BC) shows an overview of our approach. The Algorithm is divided into four stages, viz. compute SSSP, find successors, calculate path-counts, and finally update centrality, similar to the vertex centric model BC algorithm proposed by Jonathan et. al. [1]. Each stage in the algorithm in itself follows the partition-compute model (Fig. 1b).

**Algorithm 1** Partitioned Betweenness Centrality.

```
1: v ← NEXT-LOCAL-VERTEX
2: COMPUTSSSP(v)    ▷ Each partition start a computation from different
   source vertex.
3: for Each SSSP Source s in batch do
4:     succ[s] ← COMPUTE-ALL-SUCCESORS(s)
5:     path-counts[s] ← COMPUTE-PATH-COUNTS(s)
6:     UPDATECENTRALITY(C_B, succ[s], path-counts[s]);
7: end for
```

In the SSSP stage each worker independently starts computing the shortest paths from a selected local source vertex, thus utilizing all the nodes in parallel. This runs a modified Dijkstra's algorithm as shown in Alg. 2 which computes tentative distances to the local vertices from the given source and sends a message to the corresponding partition whenever a vertex is identified as a remote vertex. Note that since a partition has no information about a remote vertex and the incoming edges, the distance calculated for the local vertices may be incorrect if there exists a shorter path to the vertex that passes through a remote partition. This entails that the *label assignment* property of the Dijkstra's algorithm no longer holds and hence requires decoupling the path count calculations from the SSSP stage similar to ΔS-BC algorithm due to its *label correcting* approach. Further, as each partition processes incoming messages (Alg. 2) belonging to different paths at the same time, it requires additional aggregate space $O(p \times (n + m))$, where $p$ is the number of partitions and consequently the number of parallel executions of the SSSP algorithm.

The compute-all-successors stage traverses the predecessor tree for the shortest paths from each SSSP source in the order of non-increasing distance values and finds successors for the vertices on the shortest paths. The details are omitted for brevity. Once the successors are identified, the paths-count stage performs a similar sweep starting at the source of each shortest path. Finally, the update-centrality stage accumulates the dependency and centrality values using Eqs. 2 and 1.

## IV. Extracting High Centrality Nodes

Given that the exact BC is highly compute intensive and that most applications (e.g., network routing, traffic monitoring etc.) are interested in finding the vertices with highest centrality, algorithms that compute approximate betweenness values [18] and that extract the high centrality vertices [19] have been proposed. The general idea behind these algorithms is to incrementally update the centrality values by computing the shortest paths from a small subset of vertices, called *pivots* until a satisfactory approximation or a stopping criterion is met. This is done in *batches* and the criterion is evaluated at the end of each batch. The batch size has been shown to have direct correlation with the result quality and the algorithm runtime. This approach achieves tremendous speed-ups with high mean precision (MP) while extracting a small set of high centrality nodes [19].

The partition centric BC algorithm can be easily extended to incorporate this strategy by utilizing the master compute function which checks for the terminating criterion at the start of each superstep.

We further extend the notion of approximation and posit that further performance improvements can be achieved with minimal loss of accuracy by performing a pre-processing step which prunes the graph by removing vertices that satisfy the following conditions: (1) the vertices being removed do not exhibit high BC, and (2) removing such vertices has minimum (or equal) effect on the centrality values of other vertices.

[1] https://github.com/Sotera/high-betweenness-set-extraction

**Algorithm 2** Partitioned SSSP Computation.

d ← distance map; ▷ default value for each vertex is ∞.
$P$ ← predecessor map;
Q is a priority queue with vertex distance as the key

**procedure** LOCAL SSSP(SSSP Source $src$, Queue $Q$) ▷ Compute local shortest paths from the given source
  **while** Q not empty **do**
    u ← pop Q;
    **if** owner(u) = current process **then**
      **for** Neighbor v of u **do**
        **if** $d[v] > d[u] + 1$ **then**
          d[v] = d[u] + 1;
          DECREASE-KEY($Q, v$);
          CLEAR-LIST(P[v])
        **end if**
        **if** d[v] = d[u] + 1 **then**
          $P[v] \leftarrow append$ u;
        **end if**
      **end for**
    **else**
      send(owner(u), $src, < u, d[u], \sigma[u], P[u] >$);
    **end if**
  **end while**
**end procedure**

**procedure** PROC. SSSP MSG(SSSP Source $src$, message list $ms$)
  Q ← empty queue;
  $min_d[u] \leftarrow min_{m \in ms}\{m.d \text{ where}, m.u = u\}$;
  $min[u] \leftarrow \bigcup \arg\min_{m \in ms}\{m.d \text{ where}, m.u = u\}$ ▷ Get all messages with minimum distance for each target vertex u
  upd ← 0
  **for** u ∈ $min.keys$ **do**
    **if** $d[u] > min_d[u]$ **then**
      d[u] ← $min_d[u]$;
      DECREASE-KEY($Q, u$)
      CLEAR-LIST(P[v])
    **end if**
    **if** d[u] = $min_d[u]$ **then** ▷ on shortest path?
      **for** Message m ∈ $min[u]$ **do**
        $P[u] \leftarrow P[u] \cup m.P$
      **end for**
    **end if**
  **end for**
  LOCAL SSSP(src, Q)
**end procedure**

We observe that the graph's leaves (i.e. vertices with at most one edge) satisfy both these conditions since none of the shortest paths pass through a given leaf vertex (i.e. BC = 0) and it contributes at most one shortest path starting or terminating at that leaf vertex with equal contribution to all the nodes on that path. Hence we use a iterative leaf-compression algorithm (Alg. 3) as a pre-processing step before running the partition centric approximate betweenness algorithm. Although the formal proof for the error bounds is out of scope of the paper, we present empirical evidence for both performance improvements and relative error in precision in the following section.

## V. EVALUATIONS

*a) Experimental Setup:* To assess the quality and performance of the proposed algorithm (with and without leaf-compression)[2] we compare it against the high centrality node extraction algorithm by Chong et al. [19] implemented using the boost MPI version of ΔS-BC[3].

**Algorithm 3** Recursive Leaf Compression (undirected graph)

**while** No Updates **do**
  **for** v ∈ V **do**
    **if** EDGECOUNT(v) = 0 **then**
      REMOVEVERTEX(v)
    **else if** EDGECOUNT(v) = 1 **then**
      REMOVEEDGE(Edge)
      REMOVEVERTEX(v)
    **end if**
  **end for**
**end while**

| Data Set | vertices | edges | type |
|---|---|---|---|
| Enron Email | 36,692 | 108298 | Sparse, Preferential Attachment |
| Gowalla S/N | 196,591 | 1,900,654 | Small World |
| PA Road N/W | 1,088,092 | 3,083,364 | Sparse |
| Synthetic | 100,000 | 199,770 | |
| Synthetic | 200,000 | 801,268 | Erdos-Renyi sparse graphs |
| Synthetic | 500,000 | 2,499,973 | |
| Synthetic | 262,144 | 2,097,152 | Powerlaw, HPCS-GA: SCALE = 18 |
| Synthetic | 524,288 | 4,194,304 | SCALE = 19 |
| Synthetic | 1,048,576 | 8,388,608 | SCALE = 20 |

TABLE I: Data Sets

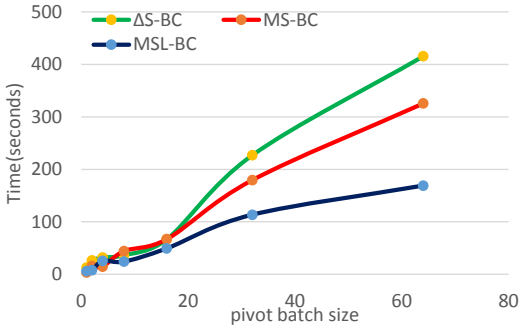
The experiments were run on a HPC cluster consisting of nodes connected with high speed interconnect each with two Quad-Core AMD Opteron 2376 processors and 16GB memory. The experiments were run on a subset of these nodes ranging from 4 to 64 for different graphs with two workers per node (i.e. 8 to 128 workers) each with upto 8GB memory allocated to it. The graphs were partitioned and distributed among the processors using a shared network file system. For the experimental results we do not include the data partition, distribution or loading times since these are consistent across the different studied algorithms and focus on end-to-end algorithm execution time as a performance metric. We use Mean Precision (MP) as the quality metrics as defined by chong et al. [19].

As the quality as well as the performance (due to different convergence rate) of a given algorithm depends on the selection and ordering of pivots, we run each experiment three times with random pivot ordering and report the median MP values and mean runtime values across the three runs.

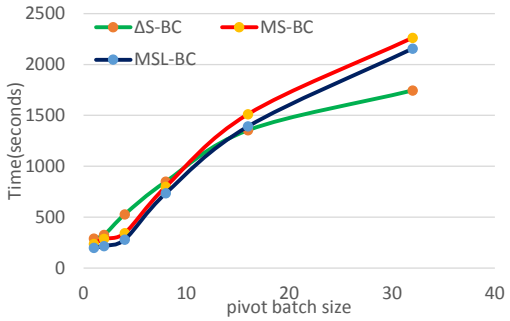
*b) Data Sets:* We evaluate the proposed algorithms over a mix of different graph types, both real-world as well as synthetic. The real world data sets includes the Enron email data set, the Gowalla social network graph, and the Pennsylvania road network data set (Tab. I). Random weights between 1 and 10 are generated for each of the edges. We use the real world graphs to analyze the quality of the algorithms, especially with leaf-compression as well as their relative performance. Further, we use several synthetic sparse as well as power-law graphs to study the performance characteristics and scalability of the proposed algorithms both with respect to increasing graph size and processing cores; and also to study the shortcomings of the proposed algorithms.

[2]available for download at: https://github.com/usc-cloud/parallel-high-betweenness-centrality

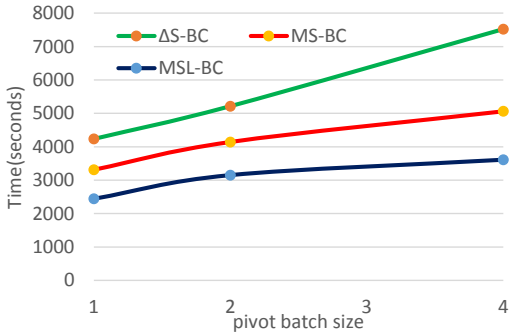
[3]http://www.boost.org/doc/libs/1_55_0/libs/graph_parallel/doc/html/betweenness_centrality.html




(a) Enron




(b) Gowalla




(c) PA Road Network

Fig. 2: Execution times for real world Graphs

*c) Results:* We first compare the quality of the proposed algorithms using real-world graphs with known ground truth. Table II shows the mean precision values for different real-world data set with the pivot batch size varying from 1 to 64 to extract top 32 high centrality nodes. While MS-BC algorithm should give exactly the same results as ΔS-BC ($\Delta = maxedgeweight = 10$) [20] given the same set of pivots, the observed minor variations are due to random pivot selections. Further, we observe that the effect of leaf-compression pre-processing step (MSL-BC) on the algorithm quality is minimal and the average error is less than 10% for the Enron data set for higher batch sizes while it performs as well as the others for smaller batch sizes. We see similar trends for the Gowalla s/n and PA road n/w graphs thus empirically validating our hypothesis.

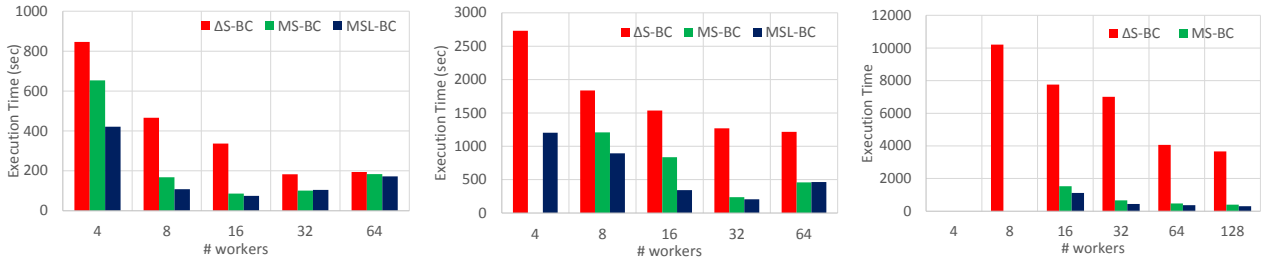
Figure 2 shows the average runtime for different real world graphs over a range of batch sizes. We observe that for the Enron and the road n/w data sets, MS-BC algorithm shows significant improvements over ΔS-BC algoritm ($\approx$ 1.5) due to the better distribution of work among the nodes. Further, the improvement in performance due to leaf compression (MSL-BC) is more pronounced ($\approx$ 2x - $\approx$ 3x) with higher improvements observed for larger batch sizes. This is primarily because of the improvement in per iteration (one batch of pivots) time due to leaf compression as the number of edges are reduced by 21.19% and 13.74% for the Enron and road n/w data sets respectively. This leads to corresponding reduction in the amount of local work done by each worker which gets accumulated due to the fact that higher batch sizes leads to more pivots being selected.

However, we observe that the performance of the algorithm suffers high synchronization penalty for the Gowalla dataset as the batch size (and hence the parallelism) is increased for a fixed set of resources. While the Enron and road network data sets are sparse and balanced (fairly uniform edge distribution), the Gowalla dataset is a dense graph that exhibits power-law edge distribution and is unbalanced. The goal of the default Metis partitioner [21] is to minimize edge cuts while balancing the number of vertices per node. However, like BC, most graph algorithms' runtime is a function of both the number of vertices and the edges, which is not accounted for by the Metis partitioner and hence leads to imbalance in the runtime of the local components, and in turn causes the synchronization delay during barrier synchronization. For the Gowalla data set we observe that the Metis partitioner produces partitions with 1:30 ratio of edge count between the smallest and the largest partition, where as 1:200 ratio for the local compute (including message processing) function. Further, as we increase the batch size (i.e. parallelism), this effect gets exaggerated as the number of pivots to be processed per partition increases. This implies that the existing partitioning schemes fall short for such graph analytics and techniques such as LALP and dynamic repartitioning [16] have been proposed. However the former is limited in its applicability [16] and the later incurs run-time performance overhead and hence requires further study.

Second, we study the performance and scalability of the proposed algorithms using a set of synthetic sparse graphs generated using the NetworkX graph generator. Figure 3 shows the execution runtime for sparse graphs of different graph sizes over a range of workers. We observe that MS-BC and MSL-BC perform consistently better than ΔS-BC for different graphs sizes and number of workers (as much as 12x performance improvement for large graphs) except for the case of small graphs with large number of workers (e.g. 100K graph with >64 workers). This is because with the increase in number of

| | Enron | | | Gowalla | | | CA Road Network | | |
|---|---|---|---|---|---|---|---|---|---|
| | MS-BC | ΔS-BC | MSL-BC | MS-BC | ΔS-BC | MSL-BC | MS-BC | ΔS-BC | MSL-BC |
| 1 | **59.375** | 56.25 | 56.25 | **31.25** | 28.125 | 25 | 25 | **28.12** | 21.8 |
| 2 | 50 | 28.125 | **56.25** | 28.125 | **31.25** | 25 | **37.5** | 34.37 | 31.25 |
| 4 | 43.75 | 46.875 | **56.25** | 28.125 | **31.25** | 28.125 | **53.125** | 50 | 34.375 |
| 8 | **68.75** | 59.375 | 53.125 | 59.375 | **62.5** | 46.875 | | | |
| 16 | 71.875 | **75** | 71.875 | **90.625** | 84.375 | 75 | | | |
| 32 | 71.875 | **81.25** | 68.75 | **87.5** | 81.25 | 71.87 | | | |
| 64 | **90.625** | 84.375 | 75 | | | | | | |

TABLE II: Mean Precision values for real-world datasets for different batch size (first column refers to the batch size).




(a) #vertices = 100k, #edges = 200k (b) #vertices = 200k, #edges = 800K (c) #vertices = 500k, #edges = 2M

Fig. 3: Execution times for Random Sparse Graphs

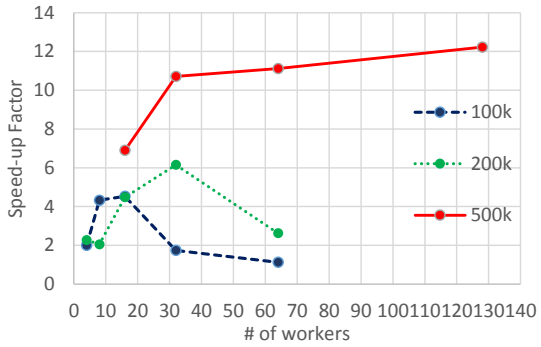



Fig. 4: MSL-BC's speedup relative to ΔS-BC for sparse graphs.

workers, the partition size decreases and in turn the amount of local work performed. At the same time there is an increase in the number of edge cuts, thus increasing the overall communication. These two factors lead to less than ideal scaling for MS-BC and MSL-BC and hence there is a drop in relative speed-up as shown in Fig. 4. Further, we observe that MS-BC and MSL-BC fail to completion for large graphs with small number of partitions due to increased memory pressure (e.g. 500k with 4 workers).

This shows both MS-BC and MSL-BC scale well with both the graph size and number of workers for sparse graphs. However there exists a break-even point for the number of workers given a graph size beyond which the performance starts to degrade.

Finally, we extend the scalability experiments over to the power-law graphs generated using R-MAT [22] power-law graph generator proposed as part of the HPC Scalable Graph Analysis (HPCS-GA) Benchmark v1.0 [23] (a precursor to Graph500 benchmark). The generator takes two parameters, the $SCALE$ factor and the edge multiplier $(\mathcal{M})$, and generates a powerlaw graph with $|V| = 2^{SCALE}$ and $|E| \approx \mathcal{M} \times |V|$ with edge weights ranging from 1 to 100. Typically an edge multiplier value of $\mathcal{M} = 8$ is suggested by the benchmark.

The benchmark also proposes a number of kernels including graph construction (K1), large sets classification (K2), graph extraction (K3), and graph analysis (K4). Of these, we focus on the graph analysis kernel (K4), the intent of which is to identify a set of vertices in the graph with highest betweenness centrality score, which is inline with our proposed algorithm. However, while the stopping criterion of our proposed algorithm is based on stability of the extracted set, the HPCS-GA benchmark uses a fixed number of the vertices $(2^{K4approx})$ as pivots. Hence we follow a similar approach for the following experiments.

The benchmark further proposes a performance metric called traversed edges per second (TEPS) to measure the performance characteristics of the algorithm as well as the underlying hardware. Since the underlying hardware is fixed for our experiments, we use TEPS to measure the relative performance of MSL-BC and ΔS-BC. For an approximate implementation, TEPS is calculated as follows:

$$TEPS(|V|) = \frac{\mathcal{M} \times |V| \times 2^{K4approx}}{time_{K4}(|V|)}$$

where $time_{k4}(|V|)$ is the execution time for kernel K4.

Given the resource constraints, we use graphs of size $V = 2^{SCALE}$, where $SCALE = 18, 19, 20$ and $|E| \approx 8 \times |V|$. Figure 5 compares TEPS for ΔS-BC and MSL-BC for different graph sizes. For these experiments, we fix the lookahead value to 100 for ΔS-BC and the batch size (i.e. number of simultaneous path calculations) to 2 for MSL-BC algorithm. As noticed, the MSL-BC algorithm out performs ΔS-BC for all graph sizes. However, the performance gain observed for powerlaw graphs (2-2.7x) is less than that observed for the sparse graphs (upto 12x). This is primarily because of the imbalance in terms of edge density and number of incoming edges between different partitions of the graph, which in turn results in straggling partitions that become bottleneck for the BSP algorithm.

To further study the impact of such straggelers on MSL-BC, we ran several experiments with fixed graph size $(SCALE = 20)$, and calculated the TEPS over different batch sizes. Note that the batch size dictates the number of parallel SSSPs running per super-step. As the batch size increases, the amount of work per partition increases proportionally to the number of incoming edges and the edge density within the partition. Hence, as shown in Fig. 6, as we increase the batch size from 2-8, the performance of the algorithm first increases since the time saved due to increase in parallelism is greater than the synchronization delay caused due to the straggling partition. However, as we increase the bath size the imbalance between the super-step execution increases and the synchronization delay becomes the bottleneck causing the performance to drop for higher batch sizes (e.g. 16, 32).

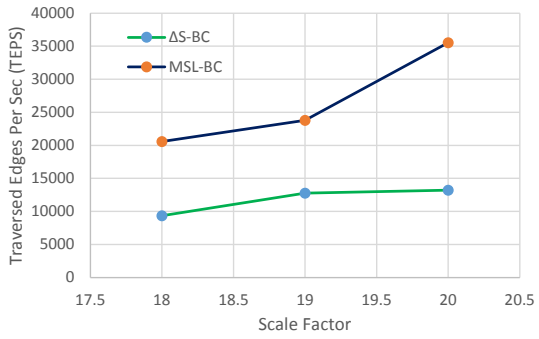



Fig. 5: TEPS for MSL-BC and ΔS-BC for power-law graphs at different scales.

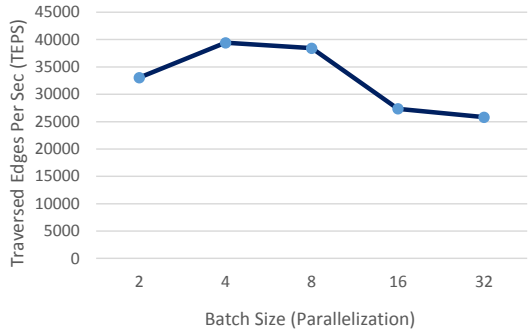



Fig. 6: TEPS for MSL-BC at SCALE=20 for differnt batch size (parallelization).

Further improvements in the execution time may be achieved by improving the partitioning scheme that not only balances the number of vertices across partitions but also balances the number of edges within and across the partitions.

## VI. Conclusion

In this paper we proposed a partition centric algorithm for efficient extraction of high centrality vertices in distributed graphs that improves the execution time by improving overall utilization of the cluster and the work distribution. We also prosed a graph pruning technique based on leaf-compression that further improves the performance and experimental results show an improvement of upto 12x for large sparse graphs. Further, we studied the performance characteristic of the proposed algorithm for large synthetic power-law graphs and observed modest performance improvements of upto 2-2.7x. We also identified the cause of the performance degradation for power-law graphs that accentuates the need for better partitioning methods.

As future work, we will analyze the effect of different partitioning schemes on the algorithm performance and propose a custom partitioner that accounts for graph structure as well as algorithm complexity while partitioning the data.

## Acknowledgment

This work was supported by a research grant from the DARPA XDATA grant no. FA8750-12-2-0319. Authors would like to thank Jonathan Larson of Sotera Defense for the data sets, ground truth results, and relevant discussions as well as Charith Wickramaararachi and Yogesh Simmhan for their critical reviews.

## References

[1] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, 1977.

[2] Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, 2001.

[3] D. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *Proc. The 35th International Conference on Parallel Processing (ICPP)*, 2006.

[4] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proceedings of the IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2009.

[5] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics," in *International Conference on Parallel Processing (EuroPar)*, 2014, p. To Appear.

[6] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[7] N. Kourtellis, T. Alahakoon, R. Simha, A. Iamnitchi, and R. Tripathi, "Identifying high betweenness centrality nodes in large social networks," *Social Network Analysis and Mining*, vol. 3, no. 4, 2013.

[8] S. P. Borgatti, "Centrality and network flow," *Social Networks*, vol. 27, no. 1, pp. 55 – 71, 2005.

[9] K. P., "Introducing the cray xmt," in *Proceedings Cray User Group meeting*, ser. CUG'07, 2007.

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

[11] M. Redekopp, Y. Simmhan, and V. K. Prasanna, "Optimizations and analysis of bsp graph processing models on public clouds," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[12] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory," in *International Conference on High Performance Computing*, 2010.

[13] C. Houngkaew and T. Suzumura, "X10-based distributed and parallel betweenness centrality and its application to social analytics," in *International Conference on High Performance Computing*, 2013.

[14] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proceedings of the 5th International Conference on Algorithms and Models for the Web-graph*, ser. WAW'07. Springer-Verlag, 2007, pp. 124–137.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010.

[16] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013.

[17] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[18] U. Brandes and C. Pich, "Centrality estimation in large networks," *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, 2007.

[19] W. H. Chong, W. S. B. Toh, and L. N. Teow, "Efficient extraction of high-betweenness vertices," in *Advances in Social Networks Analysis and Mining, 2010 International Conference on*. IEEE, 2010.

[20] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of dijkstra's shortest path algorithm," in *Mathematical Foundations of Computer Science 1998*. Springer, 1998.

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, 1998.

[22] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[23] "HPC scalable graph analysis benchmark v1.0," http://www.graphanalysis.org/benchmark/GraphAnalysisBenchmark-v1.0.pdf, accessed: 2014-07-30.