Scalable and High Performance Betweenness Centrality on the GPU

Adam McLaughlin
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Adam27X@gatech.edu

David A. Bader
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
bader@cc.gatech.edu

***Abstract*—Graphs that model social networks, numerical simulations, and the structure of the Internet are enormous and cannot be manually inspected. A popular metric used to analyze these networks is betweenness centrality, which has applications in community detection, power grid contingency analysis, and the study of the human brain. However, these analyses come with a high computational cost that prevents the examination of large graphs of interest.**

Prior GPU implementations suffer from large local data structures and inefficient graph traversals that limit scalability and performance. Here we present several hybrid GPU implementations, providing good performance on graphs of arbitrary structure rather than just scale-free graphs as was done previously. We achieve up to 13x speedup on high-diameter graphs and an average of 2.71x speedup overall over the best existing GPU algorithm. We observe near linear speedup and performance exceeding tens of GTEPS when running betweenness centrality on 192 GPUs.

Keywords*—*GPUs, Graph Algorithms, Parallel Algorithms

I. Introduction

Network analysis is a fundamental tool for domains as diverse as compilers [28], social networks [16], and computational biology [10]. Real world applications of these analyses involve tremendously large networks that cannot be inspected manually. An example of a graph analytic that has found significant attention in recent literature is Betweenness Centrality (BC). Betweenness centrality has been used for finding the best location of stores within cities [32], studying the spread of AIDS in sexual networks [25], power grid contingency analysis [24], and community detection [35]. The variety of fields and applications in which this method of analysis has been employed shows that graph analytics require algorithmic techniques that make them performance portable to as many network structures as possible. Unfortunately, the fastest known algorithm for calculating BC scores has $O(mn)$ complexity for unweighted graphs with n vertices and m edges, making the analysis of large graphs challenging. Hence there is a need for robust, high performance graph analytics that can be applied to a variety of network structures and sizes.

GPUs provide high performance for regular, dense, and computationally demanding subroutines such as matrix multiplication. However, there has been recent success in accelerating irregular, memory-bound graph algorithms on GPUs as well [13], [28], [31]. Prior implementations of betweenness centrality on the GPU have outperformed their CPU counterparts, particularly on scale-free networks; however, they are limited in scalability to larger graph instances, use asymptotically inefficient algorithms that mitigate performance on high diameter graphs, and aren't general enough to be applied to the variety of domains that can leverage their results.

This paper alleviates these problems by making the following contributions:

- We provide a work-efficient algorithm for betweenness centrality on the GPU that works especially well for networks with a large diameter.
- For generality, we propose two algorithms that alternate between leveraging either the memory bandwidth of the GPU or the asymptotic efficiency of the work being done based on the structure of the graph being processed. The first of these approaches bases its decision on how significantly the size of the working set of vertices changes across iterations. The second is an on-line approach that uses a small amount of initial work from the algorithm to suggest which method of parallelism would be best for processing the remaining work.
- We implement our approach on a single GPU system, showing an average speedup of $2.71\times$ across a variety of both real-world and synthetic graphs over the best previous GPU implementation. Additionally, our implementation attains near linear speedup on a cluster of 192 GPUs. Our single GPU approach achieves traversal rates up to 400 MTEPS (Millions of Traversed Edges per Second) while our multi-node approach achieves traversal rates exceeding 10 GTEPS (Billions of Traversed Edges per Second).

II. Background

A. Definitions

Let a graph $G = (V, E)$ consist of a set V of $n = |V|$ vertices and a set E of $m = |E|$ edges. A path from a vertex u to a vertex v is any sequence of edges originating from u and terminating at v. Such a path is a *shortest path* if its sequence contains a minimal number of edges. A Breadth-First Search (BFS) explores vertices of a graph by starting a "source" (or "root") vertex and exploring its neighbors. The neighbors of these vertices are then explored and this process repeats until there are no remaining vertices to be explored. Each set of

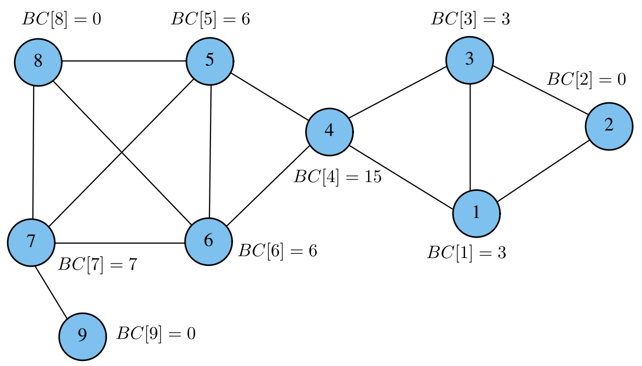


Fig. 1. Example Betweenness Centrality scores for a small graph

inspected neighbors is referred to as a *vertex frontier* and the set of outgoing edges from a vertex frontier is referred to as an *edge-frontier*. The *diameter* of a graph is the length of the longest shortest path between any pair of vertices. A *scale-free* graph has a degree distribution that follows a power law, where a small number of vertices have a large number of outgoing edges and a large number of vertices have a small number of outgoing edges [5]. Finally, a *small world* graph has a diameter that is proportional to the logarithm of the number of vertices in the graph [37]. In these networks every vertex can be reached from every other vertex by traversing a small number of edges.

B. Brandes's Algorithm

Betweenness centrality was originally developed in the social sciences for classifying people who were central to networks and could thus influence others by withholding information or altering it [18]. The metric attempts to distinguish the most influential vertices in a network by measuring the ratio of shortest paths passing through a particular vertex to the total number of shortest paths between all pairs of vertices. Intuitively, this ratio determines how well a vertex connects pairs of other vertices in the network. Formally, the Betweenness centrality of a vertex v is defined as:

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

where σ_{st} is the number of shortest paths between vertices s and t and $\sigma_{st}(v)$ is the number of those shortest paths that pass through v.

Consider Figure 1. Vertex 4 is the only vertex that lies on paths from its left (vertices 5 through 9) to its right (vertices 1 through 3). Hence vertex 4 lies on all of the shortest paths between these pairs of vertices and has a high BC score. In contrast, vertex 9 does not belong on a path between any pair of the remaining vertices in the graph and thus vertex 9 has a BC score of zero. Vertex 8 can be found on a path from vertex 5 to vertex 9; however, the shortest path from vertex 5 to vertex 9 instead goes through vertex 7. Since vertex 8 does not lie on any *shortest* paths between pairs of other vertices it also has a BC score of zero. Note that the scores reflected in Figure 1 treat a path from vertex u to vertex v as equivalent to a path from vertex v to vertex u since these paths are undirected. In other words, to avoid double counting the number of (undirected) shortest paths we divide the scores by two. One might also notice that the magnitude of BC values scales with the size of the network. For a fair comparison of BC values between vertices of two different graphs, a commonly used technique is to normalize the BC scores by their largest possible value [8]: $(n-1)(n-2)$. Such a comparison could be useful for comparing discrete slices of a network that changes over time [27].

Naïve implementations of Betweenness Centrality solve the all-pairs shortest-paths problem using the $O(n^3)$ Floyd-Warshall algorithm [17] and augment this result with path counting. Brandes improved upon this approach with an algorithm that runs in $O(mn)$ time for unweighted graphs [7]. The key concept of Brandes's approach is the *dependency* of a vertex v with respect to a given source vertex s:

$$\delta_s(v) = \sum_{w:v \in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \tag{2}$$

The recursive relationship between the dependency of a vertex and the dependency of its successors allows a more asymptotically efficient calculation of the centrality metric. Brandes's algorithm splits the betweenness centrality calculation into two major steps:

1) Find the number of shortest paths between each pair of vertices
2) Sum the dependencies for each vertex

We can redefine the calculation of BC scores in terms of dependencies as follows:

$$BC(v) = \sum_{s \neq v} \delta_s(v) \tag{3}$$

III. Prior GPU Implementations

Two well-known GPU implementations of Brandes's algorithm have been published within the last few years. Jia *et al.* [23] compare two types of fine-grained parallelism, showing that one is preferable over the other because it exhibits better memory bandwidth on the GPU. Shi and Zhang present *GPU-FAN* [34] and report a slight speedup over Jia *et al.* by avoiding data structure duplication and using a different distribution of threads to units of work. Both methods focus their optimizations on scale-free networks.

A. Vertex and Edge Parallelism

Jia *et al.* discussed two distributions of threads to graph entities: *vertex-parallel* and *edge-parallel* [23]. The vertex-parallel approach assigns a thread to each vertex of the graph and that thread traverses all of the outgoing edges from that vertex. In contrast, the edge-parallel approach assigns a thread to each edge of the graph and that thread traverses that edge only. In practice, the number of vertices and edges in a graph tend to be greater than the available number of threads so each thread sequentially processes multiple vertices or edges.

For both the shortest path calculation and the dependency accumulation stages the number of edges traversed per thread by the vertex-parallel approach depends on the out-degree of the vertex assigned to each thread. The difference in out-degrees between vertices causes a load imbalance between threads. For

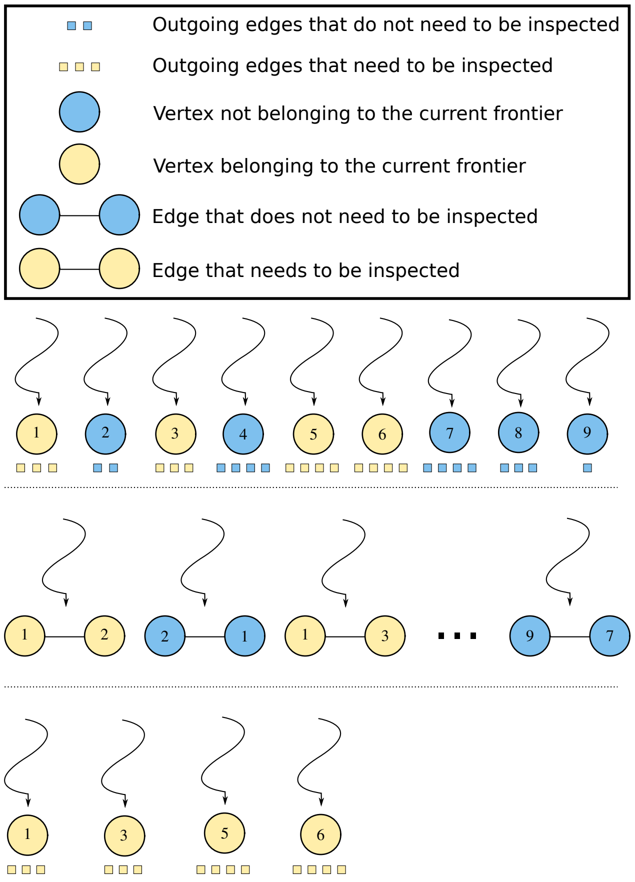

Outgoing edges that do not need to be inspected
Outgoing edges that need to be inspected
Vertex not belonging to the current frontier
Vertex belonging to the current frontier
Edge that does not need to be inspected
Edge that needs to be inspected
1
2
3
4
5
6
7
8
9
1
2
2
1
1
3
...
9
7
1
3
5
6

Fig. 2. Illustration of the distribution of threads to units of work. Top: Vertex-parallel. Middle: Edge-parallel. Bottom: Work-efficient.

scale-free networks this load imbalance can be a tremendous issue, since the distribution of out-degrees follows a power law where a small number of vertices will have a substantial number of edges to traverse [5]. The edge-parallel approach solves this problem by assigning edges to threads directly.

Both the vertex-parallel and edge-parallel approaches from Jia *et al.* use an $O(n^2 + m)$ graph traversal that checks if each vertex being processed belongs to the current depth of the search rather than keeping an explicit queue of these vertices. For graphs with large diameters this method of graph traversal produces a large amount of unnecessary work in the form of branching overhead and accesses to global memory [31].

Furthermore, even for scale-free graphs the initial and final iterations of the traversal will have a comparably small vertex frontier and possibly a small number of edges to traverse for these iterations, depending on the connectivity of those vertices. In these cases the non-linear graph traversal can also be costly in terms of execution time.

Figure 2 illustrates this concept. Using the same graph shown in Figure 1, consider a Breadth-First Search starting at vertex 4. During the second iteration of the search, vertices 1, 3, 5, and 6 are in the vertex frontier, and hence their edges need to be inspected. The vertex-parallel method, shown in the top portion of Figure 2, distributes one thread to each vertex of the graph even though the edges connecting most of the vertices in the graph do not need to be traversed, resulting in wasted work. Also note that each thread is responsible for traversing a different number of edges (denoted by the small squares beneath each vertex), leading to workload imbalances. The edge-parallel method, shown in the middle portion of Figure 2, does not have the issue of load imbalance because each thread has one edge to traverse. However, this assignment of threads also results in wasted work because the edges that do not originate from vertices in the frontier do not need to be inspected in this particular iteration (but will be unnecessarily inspected during *every* iteration). Finally, the bottom portion of Figure 2 shows a work-efficient traversal iteration where each vertex in the frontier is assigned a thread. In this case only useful work is conducted although a load imbalance can exist among threads.

B. *GPU-FAN*

The *GPU-FAN* package from Shi and Zhang was designed for the analysis of biological networks representing protein communications or genetic interactions [34]. They report speedup ranging from 11% to 19% over the implementation from Jia *et al.* on a simulated scale-free network with the number of vertices varying from 10,000 to 50,000 and a varying preferential attachment of edges to vertices. Since these results are limited in scope, it is unclear as to which of these two implementations is preferable, especially for other types of networks such as small-world networks or high-diameter networks.

Like the previous implementation from Jia *et al.*, GPU-FAN uses the edge-parallel method for load balancing across threads. The most significant difference between the two implementations is the distribution of groups of threads (thread blocks using CUDA terminology) to units of work. The GPU-FAN package focuses only on fine-grained parallelism, using all threads from all thread blocks to traverse edges in parallel for one source vertex of the BC computation at a time. In contrast, the implementation from Jia *et al.* uses both coarse-grained and fine-grained parallelism. The threads within a block traverse edges in parallel while separate thread blocks each focus on the independent roots of the BC computation. This approach requires per-block data structures for the following variables:

- d, the BFS distance from the source vertex s to each vertex
- σ, the number of shortest paths from s to each vertex
- δ, the dependency accumulation of each vertex with respect to s
- P, the predecessor lists for each vertex with respect to s

The largest of these data structures is the predecessor list, which requires $O(m)$ space. Jia *et al.* showed that the best number of thread blocks to launch is equivalent to the number of Streaming Multiprocessors (SMs) on the GPU. Since the number of SMs that currently reside on GPU architectures is small, this additional storage requirement doesn't have a significant impact on scalability.

Another significant difference between these two implementations is that GPU-FAN uses $O(n^2)$ space for the predecessor list whereas Jia *et al.* use $O(m)$ space. Since each vertex besides the source vertex can have predecessors and since any of these vertices can have up to $O(n)$ predecessors based on the topology of the graph, using $O(n^2)$ space to store this information seems reasonable; however, an $O(m)$ array of boolean values can store this information more compactly. If edge number i represents an edge from vertex u to vertex v and u is the predecessor of v, then we can mark index i of an $O(m)$ predecessor array as $true$. We will show that the choice of the $O(n^2)$ data structure for the predecessor array severely limits the scalability of this algorithm in Section V. It should also be noted that GPU-FAN's approach also has the $O(n^2 + m)$ graph traversal issues mentioned in the analysis of the algorithm from Jia *et al.*

IV. Methodology

A. Work-efficient Approach

Algorithm 1: Work-efficient Betweenness Centrality Local Variable Initialization

```
1  for $v \in V$ do in parallel
2      if $v = s$ then
3          $d[v] \leftarrow 0$
4          $\sigma[v] \leftarrow 1$
5      else
6          $d[v] \leftarrow \infty$
7          $\sigma[v] \leftarrow 0$
8      $\delta[v] \leftarrow 0$
9  $Q_{curr}[0] \leftarrow s$ ; $Q_{curr\_len} \leftarrow 1$
10 $Q_{next\_len} \leftarrow 0$
11 $S[0] \leftarrow s$ ; $S_{len} \leftarrow 1$
12 $ends[0] \leftarrow 0$ ; $ends[1] \leftarrow 1$ ; $ends_{len} \leftarrow 2$
13 shared $depth \leftarrow 0$
```

Taking note of the issues mentioned in the previous section, we now present the basis for our work-efficient implementation of betweenness centrality on the GPU. Our approach leverages optimizations from the literature in addition to our own novel techniques. Algorithm 1 shows how we initialize local variables before each of the n shortest path calculations and dependency accumulations. The first way in which our implementation differs from prior GPU implementations is that we discard the predecessor array. Since all of the other local data structures require $O(n)$ memory, we reduce the space complexity of our local data structures from $O(m)$ to $O(n)$. This removal of space comes at the cost of additional computation, but does not change the overall computational complexity of the algorithm. In the dependency accumulation stage, rather than traversing the predecessors directly, all of the neighbors of a vertex are instead traversed. This technique, known as the *neighbor traversal* approach from Green and Bader [19], not only reduces storage requirements to enhance the scalability of the algorithm, but also has been shown to generate speedups on multi-core systems.

Algorithm 1 shows a second major difference between the work-efficient approach and prior GPU implementations: the use of explicit queues for graph traversal. We initialize Q_{curr}, Q_{next}, and their respective lengths for the shortest path calculation stage. Since levels of the graph are processed in parallel we use two queues to distinguish vertices that are in the current level of the search (Q_{curr}) from vertices that are to be processed during the next level of the search (Q_{next}). For the dependency accumulation stage we initialize S and its length. In this case, we need to keep track of vertices at all levels of the search and hence we only use one data structure to store these vertices. To distinguish the sections of S that correspond to each level of the search we use the $ends$ array, where $ends_{len} = max_{v \in V}\{d[v]\} + 1$ at the end of the traversal. Vertices corresponding to depth i of the traversal are located from index $ends[i]$ to index $ends[i+1] - 1$ of S. This usage of the $ends$ and S arrays is comparable to the arrays used to store the graph in CSR format.

Algorithm 2: Work-efficient Betweenness Centrality Shortest Path Calculation

```
1  Stage 1: Shortest Path Calculation
2  while $true$ do
3      for $v \in Q_{curr}$ do in parallel
4          for $w \in neighbors(v)$ do
5              if $atomicCAS(d[w], \infty, d[v] + 1) = \infty$ then
6                  $t \leftarrow atomicAdd(Q_{next\_len}, 1)$
7                  $Q_{next}[t] \leftarrow w$
8              if $d[w] = d[v] + 1$ then
9                  $atomicAdd(\sigma[w], \sigma[v])$
10     $barrier()$
11     if $Q_{next\_len} = 0$ then
12         $depth \leftarrow d[S[S_{len} - 1]]$ - 1
13         break
14     else
15         for $tid \leftarrow 0 \ldots Q_{next\_len} - 1$ do in parallel
16             $Q_{curr}[tid] \leftarrow Q_{next}[tid]$
17             $S[tid + S_{len}] \leftarrow Q_{next}[tid]$
18         $barrier()$
19         $ends[ends_{len}] \leftarrow ends[ends_{len} - 1] + Q_{next\_len}$
20         $ends_{len} \leftarrow ends_{len} + 1$
21         $Q_{curr\_len} \leftarrow Q_{next\_len}$
22         $S_{len} \leftarrow S_{len} + Q_{next\_len}$
23         $Q_{next\_len} \leftarrow 0$
24         $barrier()$
```

A work-efficient shortest path calculation stage is shown in Algorithm 2. Iterations of the while loop correspond to the traversal of depths of the graph. The parallel for loop in Line 3 assigns one thread to each element in the queue such that edges from other portions of the graph aren't unnecessarily traversed. The atomic Compare and Swap (CAS) operation on Line 5 is used to prevent multiple insertions of the same vertex into Q_{next}. This restriction allows us to safely allocate $O(n)$ memory for Q_{next} instead of $O(m)$ in the case that duplicate queue entries are allowed. Since we only require one thread for each element in Q_{curr} rather than one thread for every vertex or edge in the graph, this atomic operation experiences limited contention and thus doesn't significantly reduce performance. Merrill *et al.* show that a prefix sum can be used to have threads cooperatively add elements to the queue [31], reducing the contention for the atomic instruction that we use on Line 6. However, their work focuses explicitly on BFS and differs from ours in that they are using all SMs of the GPU to perform one high-performance graph traversal whereas we are performing many graph traversals on each SM independently. In our tests we found that the overhead of the prefix sum was too large because the number of elements to sum on each SM is Q_{curr_len}, which is $O(n)$ in the worst case. When all SMs can contribute to these sums, as is the case in Merrill's work, this overhead is significantly reduced because each SM can independently do its portion of the sum in parallel and then the local sums can be reduced into a global sum. For our work on betweenness centrality all SMs have to perform their own sums (of the same number of elements) independently.

The conditional on Line 11 checks to see if the queue containing vertices for the next depth of the search is empty; if so, the search is complete, so we break from the outermost while loop. Otherwise, we transfer vertices from Q_{next} to Q_{curr}, add these vertices to the end of S for the dependency accumulation, and do the appropriate bookkeeping to set the lengths of these arrays.

Algorithm 3: Work-efficient Betweenness Centrality Dependency Accumulation

```
1  Stage 2: Dependency Accumulation
2  while $depth > 0$ do
3      for $tid \leftarrow ends[depth] \ldots ends[depth+1]-1$ do in parallel
4          $w \leftarrow S[tid]$
5          $dsw \leftarrow 0$
6          $sw \leftarrow \sigma[w]$
7          for $v \in neighbors(w)$ do
8              if $d[v] = d[w]+1$ then
9                  $dsw \leftarrow dsw + \frac{sw}{\sigma[v]}(1+\delta[v])$
10         $\delta[w] \leftarrow dsw$
11     $barrier()$
12     $depth \leftarrow depth - 1$
```

Algorithm 3 shows a work-efficient dependency accumulation. In addition to using the neighbor traversal approach, we are also able to eliminate the use of atomics by checking *successors* rather than the predecessors of each vertex. Since vertices at the end of the BFS tree by definition have no successors, we start the dependency accumulation one level closer to the root of the tree (see Line 12 of Algorithm 2). Furthermore, since the source vertex of the BFS tree does not contribute to its own BC score, there is no need to perform any work when $depth = 0$. The technique of checking successors was developed by Madduri *et al.* for their implementation of betweenness centrality on the Cray XMT supercomputer [26]. Rather than having multiple vertices that are currently being processed in parallel update the dependency of their common ancestor atomically, the ancestor can update itself based on its successors without the need for atomic operations. Interestingly, an edge-parallel implementation the successor approach would still require atomic operations because multiple threads could be assigned to the same ancestor.

Note that the parallel for loop in Line 3 of Algorithm 3 assigns threads only to vertices that need to accumulate their

TABLE I. CORRELATION OF VERTEX AND EDGE FRONTIER SIZES WITH EXECUTION TIME FOR THREE RANDOMLY SELECTED ROOTS OF DIFFERENT TYPES OF GRAPHS. THE SIZE OF THE VERTEX FRONTIER CORRELATES POSITIVELY WITH EXECUTION TIME REGARDLESS OF THE ROOT OR STRUCTURE OF THE GRAPH.

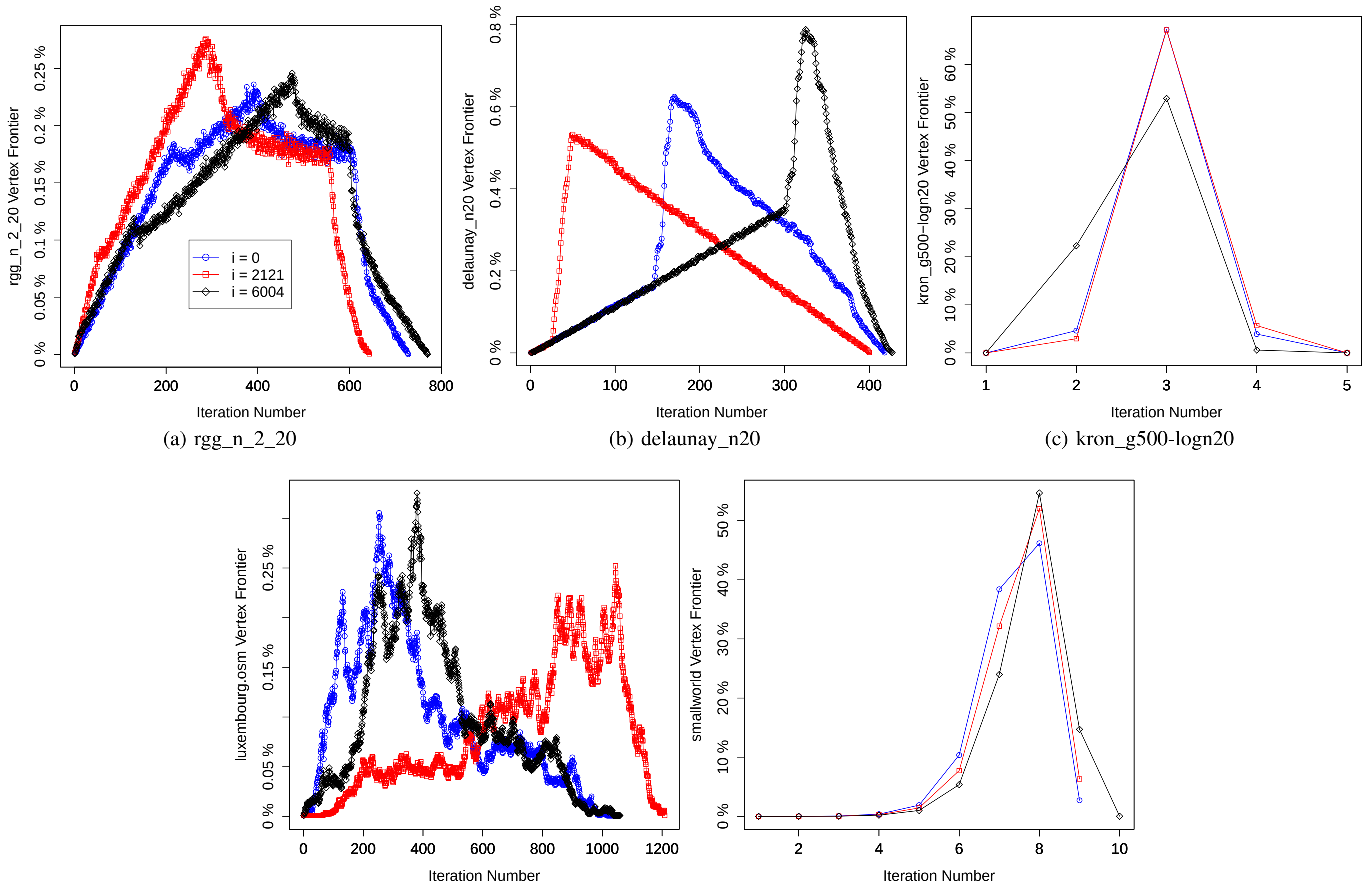
Graph	**Root**	$\rho_{v,t}$	$\rho_{e,t}$
rgg_n_2_20	0	**0.950**	0.950
	2121	**0.978**	0.976
	6004	**0.981**	0.980
delaunay_n20	0	**0.990**	0.990
	2121	**0.995**	0.995
	6004	**0.995**	0.995
kron_g500-logn20	0	**0.798**	0.093
	2121	**0.704**	0.195
	6004	**0.936**	-0.096
luxembourg.osm	0	**0.885**	0.883
	2121	**0.898**	0.892
	6004	**0.910**	0.907
smallworld	0	**0.967**	0.970
	2121	**0.989**	0.998
	6004	**0.995**	0.996

dependency values; this is where the bookkeeping done to keep track of separate levels of the graph traversal in the $ends$ array comes to fruition. Rather than naïvely assigning a thread to each vertex or edge and checking to see if that vertex or edge belongs to the current depth we instead can instantly extract vertices of that depth since they are a consecutive block of entries within S. This strategy again prevents unnecessary branch overhead and accesses to global memory that are made by previous implementations.

B. Hybrid Approach

The major drawback of the approach outlined in the previous section is the potential for significant load imbalance between threads. Although our approach efficiently assigns threads to units of useful work, the distribution of edges to threads is entirely dependent on the structure of the graph. Our approach is significantly faster than other methods on graphs with a large diameter because such graphs tend to have a more uniform distribution of outdegree. On scale-free or small world graphs, however, the algorithm outlined in the previous section does not improve performance. For these graphs there are iterations of the graph traversal that require the inspection of a high percentage of edges in the graph. For these iterations the load balance and high memory-throughput of the edge-parallel method combined with the fact that most of the edges inspected in fact need to be inspected means that the edge-parallel method is preferable to the work-efficient method. Based on this result we propose a hybrid approach that chooses between the edge-parallel and work-efficient methods based on the structure of the graph. Rather than preprocessing the graph to attempt to determine if it can be classified as a scale-free or small world graph, we implement our hybridization at a finer granularity: each iteration of the search.

Figure 3 illustrates our rationale behind this decision. Each sub-figure shows how the vertex frontier evolves for three randomly chosen source vertices within a graph. Note that the axes of the sub-figures are on different scales to appropriately show trends in the frontiers. The sub-figures represent different classifications of graphs: meshes, road networks, scale-free,



(a) rgg_n_2_20 (b) delaunay_n20 (c) kron_g500-logn20

(d) luxembourg.osm (e) smallworld

Fig. 3. Evolution of vertex frontiers (as a percentage of total vertices) for different classifications of graphs

and small-world graphs. More information on these graphs can be found in Table II. Although the position of the source vertex plays an important role in precisely how the vertex frontier changes with search iteration, we can see that the general sizes and changes in size of the vertex frontier across iterations of the search are more dependent on the overall structure of the graph. For high-diameter graphs such as *rgg_n_2_20*, *delaunay_n20*, and *luxembourg.osm* (Figures 3a, 3b, and 3d respectively), the vertex frontier grows gradually and is always a small portion of the total number of vertices in the graph. For graphs with a smaller diameter such as *kron_g500-logn20* and *smallworld* (in Figures 3c and 3e), the vertex frontier grows large after just a few iterations and contains over half of the total number of vertices in the graph at its peak. Both small-world and scale-free graphs tend to exhibit this behavior. Although the edge-frontier accurately estimates the amount of necessary work for a given iteration, we found that the size of the vertex frontier correlates quite well with the execution time of an iteration. For instance, see Table I. For the same source vertices whose vertex frontiers were plotted in Figure 3 we record the elapsed time of each iteration (using the work-efficient method) and calculate two correlation coefficients: $\rho_{v,t}$, which is the correlation between the size of the vertex frontier and elapsed time, and $\rho_{e,t}$, which is the correlation between the size of the edge frontier and elapsed time. The correlation of the vertex frontier with execution time is more robust to changes in source vertex or graph structure. This is an important result because we have the size of the vertex frontier at every iteration (since we keep an explicit queue). Obtaining the size of the edge frontier, in contrast, would require additional computation and fetches to memory.

Intuitively, for large vertex frontiers, the edge-parallel approach is favorable because of its memory throughput whereas for small vertex frontiers the work-efficient approach is favorable because the number of edges that will be traversed is significantly smaller than the total number of edges in the graph. While it is clear that the work-efficient approach is the best choice for all search iterations for graphs with a high diameter, the hybrid approach is especially useful for scale-free and small world graphs. These graphs contain search iterations that can have either a small or large vertex frontier (compared to the total number of vertices), which means that the work-efficient approach could be best for some iterations while the edge-parallel approach could be best for others. Graph structures that prefer the work-efficient method tend to have consistent vertex frontiers, each of which contain a small percentage of vertices in the graph. Using the edge-parallel approach for any iteration would be wasteful for these classes of input. In contast, when the edge-parallel approach is favored, there will be some search iterations that have a comparably small amount of required work. For example, the initial iteration of the search simply expands the root vertex itself. If the root vertex is not a high degree vertex the work required (and available parallelism) for this iteration will be limited. Ideally, we would use the work-efficient method to process this iteration.

Initially, we measured the size of the vertex frontier at a given iteration and compared it to the total number of vertices in the graph. If this ratio exceeded a threshold, then the edge-parallel approach would be used. Otherwise, the work-efficient approach was used. The problem with this approach is that the percentage of vertices found in the queue can have tremendously different implications for different scales of graphs, even if those graphs have a similar classification or structure. For instance, a road network consisting of 1,000,000 vertices would be much more likely to use the edge-parallel approach than a road network consisting of 1,000 vertices, even though neither network would benefit from this approach.

Similarly, if we were to instead use the absolute size of the frontier as a threshold we would also observe undesired outcomes. The fact that the size of the vertex frontier has crossed a certain threshold gives no information about *how* it crossed that threshold. If the threshold is close to the size of the graph, then the edge-parallel approach would be performing mostly useful edge traversals and is likely to be the better choice; however in the case that the threshold is much smaller than the size of the graph the edge-parallel approach would perform many unnecessary edge traversals and the work-efficient approach would instead be the better choice.

We instead would like to detect when the frontier is significantly changing from one iteration to another as this information tells us when our strategy should change. If the frontier is large and has not significantly changed (i.e. is still large), then we should continue to use the edge-parallel method to leverage its memory throughput. Conversely, if the frontier is small and has not significantly changed then we should continue to use the work-efficient method. Hence, we only need to reconsider our parallelization strategy when the size of the frontier changes from small to large or vice versa. Once we have decided that we the frontier has changed in this way, we can select the proper strategy based on the size of the frontier that is to be processed during the next iteration of the traversal.

Algorithm 4: Hybrid method for selecting parallelization strategy

```
1 $Q_{change} = abs(Q_{next\_len} - Q_{curr\_len})$
2 if $Q_{change} > \alpha$ then
3     if $Q_{next\_len} > \beta$ then
4         //Choose edge-parallel method
5     else
6         //Choose work-efficient method
```

Algorithm 4 describes this process. The variable Q_{change} represents the change in size of the vertex frontier. If this value is less than or equal to the parameter α, then we continue to use the same strategy. Otherwise, we noticed that the size of the vertex frontier has substantially changed and that we should reconsider our strategy. If the number of vertices to be processed during the next iteration (Q_{next_len}) is larger than the parameter β, then we choose the edge-parallel method. Otherwise, we opt for the work-efficient method. In our experiments we found the values of 768 and 512 were the best choices for α and β, respectively. Although we were able to obtain favorable results in comparison to prior methods using these choices of α and β for all of the graphs tested, poor selection of these parameters can, in general, have a significant impact on performance. We initially start our calculations with the work-efficient method for two reasons. Firstly, the initial vertex frontier is simply the root itself, and for sparse graphs this vertex is unlikely to be heavily connected to the rest of the graph. Secondly, in the case that the edge-parallel method is preferred, using the work-efficient method shows a 2.2x slowdown in worst case for the input sets that we tested. However, in the case that the work-efficient method is preferred, using the edge-parallel method can show greater than a 10x reduction in speed. Thus, incorrectly choosing the edge-parallel method is more costly than incorrectly choosing the work-efficient method.

C. Sampling

The exact computation of betweenness centrality computes a BFS for each vertex in the graph. Since all of these searches are independent, they can be executed in parallel. For large graphs of interest, there often are fewer available parallel resources than vertices in the graph. For example, the Titan supercomputer at Oak Ridge National Laboratory has 18,688 GPUs and it is ranked second on the June 2014 TOP500 list [1], [15]. For graphs whose vertices mostly belong to one large connected component, the amount of time to process each root is roughly equivalent, as the same number of edges need to be traversed for each root. Therefore the amount of time required to process k vertices is roughly k times the time required to process one vertex [33].

Algorithm 5: Sampling method for selecting parallelization strategy

```
  Input: Set of $n_{samps}$ connected component sizes ($keys$)
1 $sort(keys)$
2 $barrier()$
3 if $keys[n_{samps}/2] < \gamma * \log_2(n)$ then
4     //Choose edge-parallel method
```

Using the above analysis, an estimate of the average size of the connected components within the graph (and thus the preferred method of parallelism) is obtained by processing a small subset of the vertices and storing the maximum distance of the BFS from these vertices. Essentially this method willingly computes a small number of source vertices using a potentially unsatisfactory method of parallelism and uses this result to ensure that the desired method of parallelism is used to compute the remaining source vertices. Algorithm 5 shows how this method is implemented. For our implementation we set n_{samps} to 512 and process these vertices using the work-efficient method. For these vertices, we record the maximum depth of each of their BFS traversals and take the median of this set as our estimate. We prefer the median because it is an unbiased estimator and less sensitive to outliers. If this median is sufficiently small then it is likely that our graph is a small-world or scale-free graph and thus we should switch to using the edge-parallel approach. Again, the work-efficient method is chosen by default; Algorithm 5 helps us determine whether

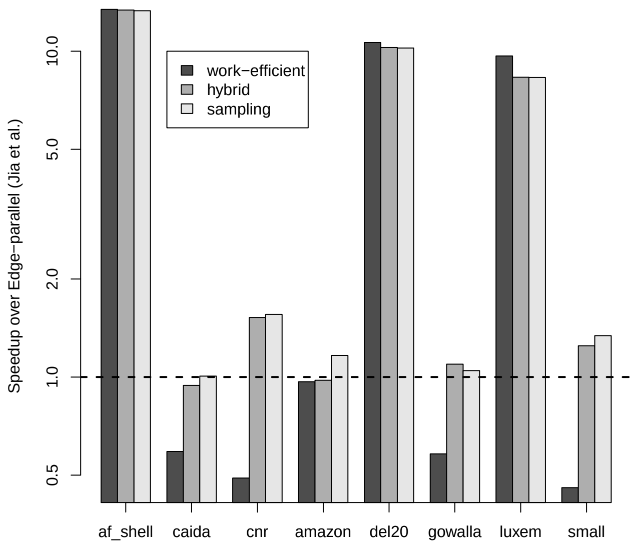


Fig. 4. Comparison of Work-Efficient, Hybrid, and Sampling methods

or not to deviate from that initial strategy. In our experiments we found the value of $\gamma = 4$ to be best.

The advantage to the sampling approach is that it leverages completed work to come to its decision rather than preprocessing the graph or doing calculations that do not directly advance the progress of the program. The drawback of the sampling approach is that it does not switch its strategy at the granularity of a search iteration like the hybrid method does. Given our analysis of Figure 3, we recall that when the work-efficient method is preferred, it is preferred for all iterationsof the traversal. Hence, we only need the search iteration level of granularity for choosing our method of parallelism when the edge-parallel method is best for a given network. To avoid using the edge-parallel method for iterations of the search that have trivial amounts of work we simply check the size of the queue to make sure it is sufficiently large. If it is, we proceed with the edge-parallel method; otherwise, we revert to the work-efficient method. We perform this check at every search iteration when the sampling method chooses the edge-parallel approach. Similar to the use of β in the hybrid approach, the sampling approach requires the vertex frontier to contain at least 512 elements to use the edge-parallel method. This parameter is designed to scale with the architecture rather than the size or structure of the graph.

Figure 4 provides a comparison of the various parallelization methods discussed in this paper to the edge-parallel method from Jia et. al [23]. For road networks and meshes (*af_shell, del20, luxem*) all of the methods outperform the edge-parallel method by about $10\times$. The amount of unnecessary work performed by the edge-parallel method for these graphs is severe. Note that the work-efficient method outperforms both the hybrid and sampling methods for these graphs. The latter methods require either additional computation or overhead for deciding which method of parallelism to use; this difference in performance is essentially the cost of generality. For the remaining graphs (scale-free and small-world graphs) using the work-efficient method alone performs slower than the edge-parallel method whereas the hybrid and sampling methods are either the same or slightly better. In these cases we see the advantage of choosing our method of parallelization at the granularity of a search iteration. If information about the structure of the graph is known a priori, then the value of α can be adjusted accordingly; however, given no information at all, the sampling approach slightly outperforms the hybrid approach overall. The sampling approach deduces the structure of the graph based on information extracted from completed a small portion of useful work whereas the hybrid method estimates the structure by using the parameters α and β in addition to vertex frontier sizes.

V. Results

A. Experimental Setup

Single-node GPU experiments were implemented using the CUDA 6.0 Toolkit. The CPU is an Intel Core i7-2600K processor running at 3.4 GHz with an 8 MB cache and 16 GB of DRAM. The GPU is a GeForce GTX Titan that has 14 Streaming Multiprocessors (SMs) and a base clock of 837 MHz. The Titan has 6 GB of GDDR5 memory and is a CUDA compute capability 3.5 (“Kepler”) GPU.

Multi-node experiments were run on the Keeneland Initial Delivery System (KIDS) [36]. KIDS has two Intel Xeon X5660 CPUs running at 2.8 GHz and three Tesla M2090 GPUs per node. Nodes are connected by an Infiniband QDR network. The Tesla M2090 has 16 SMs, a clock frequency of 1.3 GHz, 6 GB of GDDR5 memory, and is a CUDA compute capability 2.0 (“Fermi”) GPU.

We compare our techniques to both GPU-FAN [34] and Jia *et al.* [23] when possible, using their implementations that have been provided online[1]. The graphs used for these comparisons are shown in Table II. These graphs were taken from the 10th DIMACS Challenge [4], the University of Florida Sparse Matrix Collection [14], and the Stanford Network Analysis Platform (SNAP) [12], [38]. These benchmarks contain both real-world and randomly generated instances of graphs that correspond to a wide variety of practical applications and network structures. Although numerous approaches for approximating Betweenness Centrality have been proposed [3], [9], we focus our attention on its exact computation, noting that our techniques can be trivially adjusted for approximation.

B. Scaling

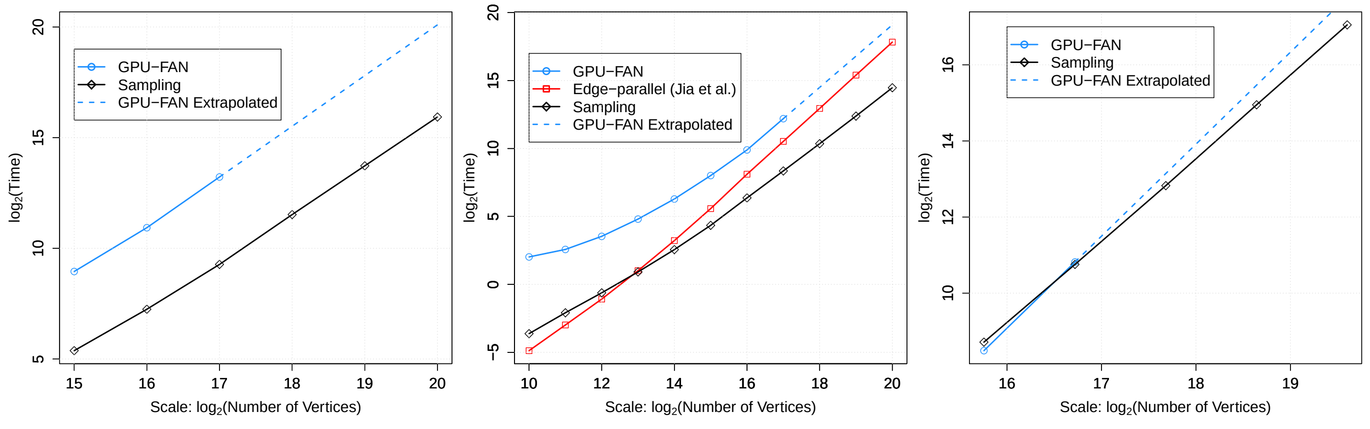
First we compare how well our algorithm scales with graph size for three different types of graphs. Since the implementation of Jia *et al.* cannot read graphs that contain isolated vertices, we were unable to obtain results using this reference implementation for the random geometric (*rgg*) and simple Kronecker (*kron*) graphs. Additionally, since the higher scales caused GPU-FAN to run out of memory, we simply extrapolated what we would expect these results to look like from the results at lower scales (denoted by dotted lines). Note that from one scale to the next the number of vertices and number of edges both double.

Noting the log-log scale on the axes, we can see from Figure 5a that the sampling approach outperforms the algorithm from GPU-FAN by over 12x for all scales of *rgg*. It is interesting

[1]Our implementation is available at https://github.com/Adam27X/hybrid_BC

TABLE II. GRAPH DATASETS USED FOR THIS STUDY

Graph	Vertices	Edges	Max degree	Diameter	Description
af_shell9 [14]	504,855	8,542,010	39	497	Sheet metal forming
caidaRouterLevel [4]	192,244	609,066	1,071	25	Internet router-level topology
cnr-2000 [4]	325,527	2,738,969	18,236	33	Web crawl
com-amazon [38]	334,863	925,872	549	46	Amazon product co-purchasing
delaunay_n20 [4]	1,048,576	3,145,686	23	444	Random triangulation
kron_g500-logn20 [11]	1,048,576	44,619,402	131,503	6	Kronecker
loc-gowalla [12]	196,591	1,900,654	29,460	15	Geosocial
luxembourg.osm [4]	114,599	119,666	6	1,336	Road map
rgg_n_2_20 [21]	1,048,576	6,891,620	36	864	Random geometric
smallworld [37]	100,000	499,998	17	9	Small world phenomenon



(a) rgg (b) delaunay (c) kron

Fig. 5. Scaling by problem size for three different types of graphs

to note that the sampling approach only takes slightly more time than GPU-FAN when the sampling approach processes a graph four times as large. For the *delaunay* mesh graphs in Figure 5b we can see that the edge-parallel method and the sampling approach both outperform GPU-FAN for all scales. The edge-parallel approach even outperforms the sampling approach for graphs containing less than 10,000 vertices; however, it should be noted that these differences in timings are trivial as they are on the order of milliseconds. As the graph size increases the sampling method clearly becomes dominant and the speedup it achieves grows with the scale of the graph. Again, the sampling approach can handle a graph with a million vertices faster than the previous approaches can handle a graph that is only half as large. Finally, we compare the sampling approach to GPU-FAN for *kron* in Figure 5c. Although GPU-FAN is marginally faster than the sampling approach for the smallest scale graph we can see that the sampling approach is best at the next scale and the trend shows the amount by which the sampling approach is best grows with scale. Furthermore, neither of the previous implementations could support this type of graph at larger scales whereas the sampling method can support even larger scales.

C. Benchmarks

For graph algorithms, standard metrics such as FLoating-point Operations Per Second (FLOPs) are not accurate indicators of performance because most of their processing time is spent accessing memory [2]. One alternative metric to FLOPs used to measure the performance of data-intensive algorithms is the number of Traversed Edges per Second (TEPS). For the exact computation of betweenness centrality that is considered in this paper, the number of TEPS has been defined as [33]:

$$TEPS_{BC} = \frac{mn}{t} \tag{4}$$

In this equation, n is the number of vertices, m is the number of (undirected) edges, and t is the execution time of the BC computation.

Table III compares sampling results to Jia *et al.* only because the graphs tested are too large to be analyzed by GPU-FAN. Results are reported in Millions of Traversed Edges Per Seconds (MTEPS). In the most extreme case, the edge-parallel approach requires more than two and half days to process the *af_shell9* graph while the sampling approach cuts this time down to

TABLE III. PERFORMANCE OF EDGE-PARALLEL AND SAMPLING METHODS FOR VARIOUS GRAPHS. RESULTS ARE IN MTEPS (MILLIONS OF TRAVERSED EDGES PER SECOND).

Graph	Edge-parallel	Sampling	Speedup
af_shell9	18.00	239.66	13.31x
caidaRouterLevel	180.98	182.21	1.01x
cnr-2000	141.75	220.64	1.56x
com-amazon	109.72	127.79	1.16x
delaunay_n20	14.19	145.09	10.23x
loc-gowalla	209.56	219.31	1.05x
luxembourg.osm	4.74	39.42	8.31x
smallworld	297.48	398.63	1.34x
Average	**2.71x Geometric Mean Speedup**		

under five hours. Similarly, the edge-parallel approach takes over 48 minutes to process the *luxembourg.osm* road network whereas the sampling approach requires just 6 minutes. We can see that the sampling approach achieves approximately 40 MTEPS for all of these graphs whereas the edge-parallel method has particularly low MTEPS for high-diameter graphs. The TEPS metric described above only accounts for edges that need to be traversed by the algorithm (i.e. useful work). Since the edge-parallel approach naïvely traverses every edge for every BFS iteration of every root, the number of useful edge traversals per unit time is overcome by futile edge inspections. Overall, sampling performs 2.71x faster on average than the edge-parallel approach.

D. Multi-GPU Experiments

Although our approaches leverage both coarse and fine-grained parallelism there is still more available parallelism than can be handled by a single GPU. Our methods easily extend to multiple GPUs as well as multiple nodes. We extend the algorithm by distributing a subset of roots to each GPU. Since each root can be processed independently in parallel, we should expect close to perfect scaling if each GPU has a sufficient (and an evenly distributed) amount of work. For graphs that have one very large connected component the amount of work to perform (in terms of the number of edges to traverse) will be equivalent for each root and thus, each GPU if the number of GPUs divides evenly into the number of source vertices. For graphs that have a larger number of connected components an imbalance between GPUs is of course more probable; however, since each GPU processes hundreds of source vertices (or more) it is highly likely for each GPU to process source vertices from each connected component.

Although the local data structures for each root are independent (and thus only need to reside on one GPU), we replicate the data representing the graph itself across all GPUs to eliminate communication bottlenecks. Once each GPU has its local copy of the BC scores these local copies are accumulated for all of the GPUs on each node. Finally, the node-level scores are reduced into the global BC scores by a simple call to $MPI\_Reduce()$. Figure 6 shows how well our algorithm scales out to multiple GPUs for *delaunay*, *rgg*, and *kron* graphs. It shows that linear speedup is easily achievable if the problem size is sufficiently large (i.e. if there is sufficient work for each GPU). Looking at the *delaunay* 64 node case specifically, it appears that the graph needs at least 2^{18} vertices to achieve near linear speedups. Since each node contains 3 GPUs, this allocates at least 1350 root vertices to each GPU. Furthermore, since the *delaunay* graphs have a particularly small ratio of edges to vertices, these particular graphs need *more* work per GPU than denser graphs that typically occur in real-world problems. Hence linear speedups are achievable at even smaller scales of graphs for denser network structures. For instance, using 64 nodes provides about a $35\times$ speedup over a single node for scale 16 *delaunay* graph whereas using the same number of nodes at the same scale for *rgg* and *kron* graphs provides over $40\times$ and $50\times$ speedups respectively. The scaling behavior seen in Figure 6 is not unique to these graphs because of the vast amount of coarse-grained parallelism offered by the algorithm. For graphs of large enough size this scalability can be obtained independently of network structure.

TABLE IV. MULTI-NODE PERFORMANCE FOR VARIOUS GRAPHS. RESULTS ARE IN GTEPS (BILLIONS OF TRAVERSED EDGES PER SECOND).

Graph	**64 Nodes (GTEPS)**	**Speedup over 1 Node**
rgg_n_2_20	8.25	63.34x
delaunay_n20	9.37	63.24x
kron_g500-logn20	24.13	63.75x

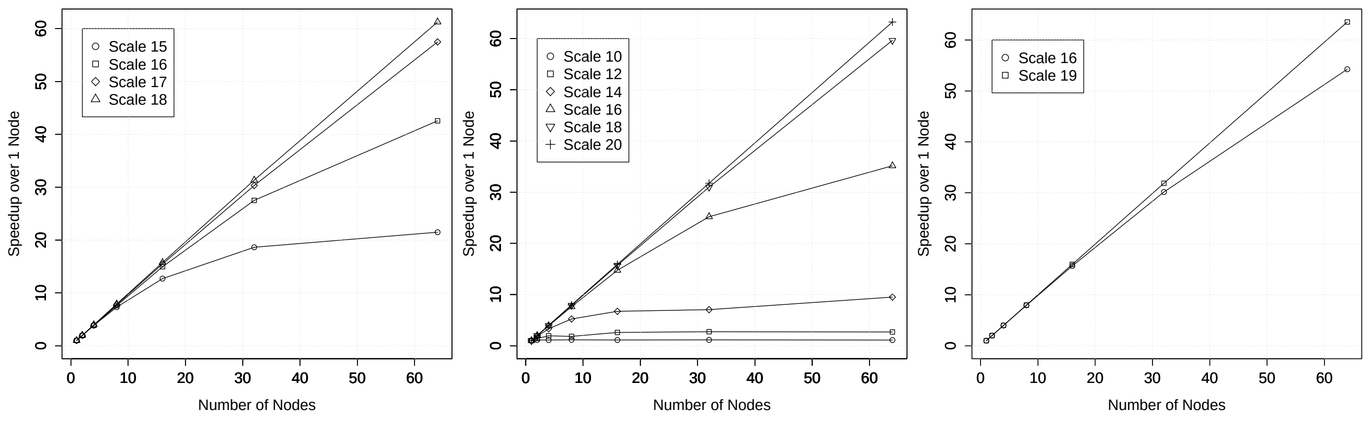
Table IV shows TEPS rates for our 64 node (192 GPU) implementation. Results are reported in Billions of Traversed Edges per Second (GTEPS). For each graph classification we see almost perfect linear speedup over 1 node (3 GPUs). The notably better TEPS rate for the Kronecker graph occurs because this graph tends to have more isolated vertices than real world graphs (or even other synthetic graphs) due to how it is randomly generated. Since the calculation for TEPS implicitly assumes that all vertices belong to one connected component, the reported TEPS value for *kron_g500-logn20* is inflated. Nevertheless, over 75% of the vertices for this graph are not isolated, and adjusting for this factor still results in approximately 18 GTEPS for this graph. The reason that this adjusted value is still greater than the TEPS values for the *delaunay* and *rgg* graphs is that the Kronecker graph is scale-free and thus utilizes the edge-parallel method for certain traversal iterations.

VI. RELATED WORK

Recent work on high performance graph algorithms has focused on accelerators, hybridization, and vectorization. Davidson *et al.* provide a GPU implementation to solve the Single-Source Shortest Path (SSSP) problem and also show a tradeoff between work-efficiency and available parallelism [13]. They compare the performance of various methods that all save work compared to the traditional Bellman-Ford approach. We consider the application of hybrid approaches such as the ones presented in this paper to this problem to be an interesting direction of future work. Beamer *et al.* present a hybrid implementation of Breadth-First Search on multi-socket CPU server systems [6]. Similar to our hybrid approach, they use one approach ("top-down") when the vertex frontier is small and another ("bottom-up") when the vertex frontier is large. Their heuristic requires the size of the frontier, the number of edges to check from the frontier, and the number of edges to check from unexplored vertices. We alternatively use the change in the frontier size as the major factor in deciding how to distribute threads to units of work. Finally, Hong *et al.* provide a fast parallel detection of Strongly Connected Components in Small-World Graphs on multi-core CPUs [22]. Their work found limitations of previous approaches that were especially detrimental on large graph instances that exhibit the small-world phenomenon. The limitation of this approach is that it is not performance portable to general structures of network data, requiring users to have a priori knowledge of the topological structure of their input.

VII. CONCLUSIONS

In this paper we have discussed various methods for computing Betweenness Centrality on the GPU. Leveraging information about the structure of the graph, we present several methods that choose between two methods of parallelism: edge-parallel and work-efficient. For high-diameter graphs using



(a) rgg (b) delaunay (c) kron

Fig. 6. Multi-GPU scaling by number of nodes for various graph structures. Each node contains three GPUs.

asymptotically optimal algorithms is paramount to obtaining good performance whereas for small-diameter graphs it is preferable to maximize memory throughput, even if unnecessary work is completed. In addition our methods are more scalable and general than existing implementations. Finally, we run our algorithm on a cluster of 192 GPUs, showing that speedup scales almost linearly with the number of GPUs. Overall, our single-GPU approaches perform $2.71\times$ faster on average than the best previous GPU approach and our multi-GPU implementation is capable of exceeding 10 GTEPS.

For future work we would like to efficiently map additional graph analytics to parallel architectures. The importance of robust, high-performance primitives cannot be overstated for the implementation of more complicated parallel algorithms. Ideally, GPU kernels should be modular and reusable [30]; fortunately, packages such as Thrust [20] and CUB [29] are beginning to bridge this gap. A software environment in which users have access to a suite of high performance graph analytics on the GPU would allow for fast network analysis and serve as a building block for more complicated programs.

ACKNOWLEDGMENT

The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited." This work was also partially sponsored by NSF Grant ACI-1339745 (XScala). This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. Special thanks to Jeff Young for his assistance with running experiments on KIDS. Finally, we would also like to thank NVIDIA Corporation for their donation of a GeForce GTX Titan GPU.

REFERENCES

[1] ORNL Debuts Titan Supercomputer. 2012 (accessed April 10, 2014). [Online]. Available: https://www.olcf.ornl.gov/wp-content/themes/olcf/titan/Titan_Debuts.pdf

[2] M. Anderson, "Better Benchmarking for Supercomputers," *Spectrum, IEEE*, vol. 48, no. 1, pp. 12–14, 2011.

[3] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating Betweenness Centrality," in *Algorithms and models for the web-graph*. Springer, 2007, pp. 124–137.

[4] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge*, ser. Contemporary Mathematics, vol. 588, 2013.

[5] A.-L. Barabási and R. Albert, "Emergence of Scaling in Random Networks," *science*, vol. 286, no. 5439, pp. 509–512, 1999.

[6] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3, pp. 137–148, 2013.

[7] U. Brandes, "A Faster Algorithm for Betweenness Centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[8] U. Brandes, "On Variants of Shortest-Path Betweenness Centrality and their Generic Computation," *Social Networks*, vol. 30, no. 2, pp. 136–145, 2008.

[9] U. Brandes and C. Pich, "Centrality Estimation in Large Networks," *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, pp. 2303–2318, 2007.

[10] E. Bullmore and O. Sporns, "Complex Brain Networks: Graph Theoretical Analysis of Structural and Functional Systems," *Nature Reviews Neuroscience*, vol. 10, no. 3, pp. 186–198, 2009.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining." in *SDM*, vol. 4. SIAM, 2004, pp. 442–446.

[12] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and Mobility: User Movement in Location-Based Social Networks," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 1082–1090.

[13] A. A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *International Parallel and Distributed Processing Symposium*, vol. 28, 2014.

[14] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[15] J. J. Dongarra, H. W. Meuer, and E. Strohmaier, "Top500 Supercomputer Sites," 1994.

[16] D. Ediger, K. Jiang, J. Riedy, D. A. Bader, C. Corley, R. Farber, and W. N. Reynolds, "Massive Social Network Analysis: Mining Twitter for Social Good," 2010.

[17] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–, Jun. 1962.

[18] L. C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, pp. 35–41, 1977.

[19] O. Green and D. A. Bader, "Faster Betweenness Centrality Based on Data Structure Experimentation," *Procedia Computer Science*, vol. 18, no. 0,

pp. 399 – 408, 2013, 2013 International Conference on Computational Science.

[20] J. Hoberock and N. Bell, "Thrust: A Parallel Template Library," *Online at http://thrust. googlecode. com*, vol. 42, p. 43, 2010.

[21] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a Scalable High Quality Graph Partitioner," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

[22] S. Hong, N. C. Rodia, and K. Olukotun, "On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 92.

[23] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart, "Edge v. Node Parallelism for Graph Centrality Metrics," *GPU Computing Gems*, vol. 2, pp. 15–30, 2011.

[24] S. Jin, Z. Huang, Y. Chen, D. Chavarria-Miranda, J. Feo, and P. C. Wong, "A Novel Application of Parallel Betweenness Centrality to Power Grid Contingency Analysis," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–7.

[25] F. Liljeros, C. R. Edling, L. A. Amaral, H. E. Stanley, and Y. Aberg, "The Web of Human Sexual Contacts," *Nature*, vol. 411, no. 6840, pp. 907–908, 2001.

[26] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda, "A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, May 2009, pp. 1–8.

[27] A. McLaughlin and D. Bader, "Revisiting Edge and Node Parallelism for Dynamic GPU Graph Analytics," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2014.

[28] M. Mendez-Lojo, M. Burtscher, and K. Pingali, "A GPU Implementation of Inclusion-based Points-to Analysis," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 107–116.

[29] D. Merrill. CUDA Unbound. 2013 (accessed April 11, 2014). [Online]. Available: http://nvlabs.github.io/cub/

[30] D. Merrill, M. Garland, and A. Grimshaw, "Policy-based Tuning for Performance Portability and Library Co-Optimization," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–10.

[31] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128.

[32] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and R. Messora, "Street Centrality and Densities of Retail and Services in Bologna, Italy," *Environment and Planning B: Planning and design*, vol. 36, no. 3, pp. 450–465, 2009.

[33] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. Çatalyürek, "Regularizing Graph Centrality Computations," *Journal of Parallel and Distributed Computing (JPDC)*, 2014 (to appear).

[34] Z. Shi and B. Zhang, "Fast Network Centrality Analysis using GPUs," *BMC bioinformatics*, vol. 12, no. 1, p. 149, 2011.

[35] J. Soman and A. Narang, "Fast Community Detection Algorithm with GPUs and Multicore Architectures," in *International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 568–579.

[36] J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community," *Computing in Science & Engineering*, vol. 13, no. 5, pp. 90–95, 2011.

[37] D. J. Watts and S. H. Strogatz, "Collective Dynamics of Small-World Networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[38] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities Based on Ground-Truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. ACM, 2012, p. 3.