Revisiting Edge and Node Parallelism for Dynamic GPU Graph Analytics

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—Betweenness Centrality is a widely used graph analytic that has applications such as finding influential people in social networks, analyzing power grids, and studying protein interactions. However, its complexity makes its exact computation infeasible for large graphs of interest. Furthermore, networks tend to change over time, invalidating previously calculated results and encouraging new analyses regarding how centrality metrics vary with time.

While GPUs have dominated regular, structured application domains, their high memory throughput and massive parallelism has made them a suitable target architecture for irregular, unstructured applications as well. In this paper we compare and contrast two GPU implementations of an algorithm for dynamic betweenness centrality. We show that typical network updates affect the centrality scores of a surprisingly small subset of the total number of vertices in the graph. By efficiently mapping threads to units of work we achieve up to a 110x speedup over a CPU implementation of the algorithm and can update the analytic 45x faster on average than a static recomputation on the GPU.

I. INTRODUCTION

The exploding popularity of online social networking has created a profound demand for high performance, scalable graph analytics. A particularly popular set of analytics attempt to measure *centrality*, or the importance of a given degree in its network. Such analyses can be used for contingency analysis for power grid component failures [1] or to find the best locations for stores within cities [2].

Architectural improvements haven't been fast enough to keep up with the demand for faster calculation of these analytics. In addition, many networks of practical interest are rapidly changing with time, exacerbating this issue. Hence, it is crucial for algorithms to be able to update analytics rather than recompute them. A lack of dynamic graph analytics in the literature leads to frameworks that perform static computations for graphs at different points in time. Repetitive static computations are wasteful since updates to the analytic typically only require computation on a small subset of the graph. The tremendous volume of updates to social networks and the web demands a high throughput solution that can process many updates in a given unit time. Thus the construction of dynamic graph analytics on the GPU is particularly useful for these applications. Although there has been recent work regarding irregular GPU computations [3], to our knowledge this paper is the first attempt to implement a dynamic graph computation on the GPU.

The key contributions of this paper are summarized below:

- We present several GPU implementations of dynamic betweenness centrality, the best of which can get significant speedup over 1) a CPU version of dynamic betweenness centrality and 2) the full recomputation of betweenness centrality on the GPU for a diverse set of graphs.

- We provide an analysis of the frequency of the various scenarios that can occur when updating the graph and how each of these scenarios affect performance. Furthermore, we analyze the portion of the graph affected by each update and show that efficiently mapping threads to units of work is paramount to obtaining high performance.

- We analyze node and edge-based parallelism for our algorithm, noting that although edge-based parallelism has greater memory throughput, node-based parallelism has less contention over shared resources. Since the typical number of vertices to be processed at a particular instant is much less than the total number of edges in the graph, the node-based approach scales better and sees better performance characteristics overall.

The rest of the paper is organized as follows. Section 2 focuses on a large amount of related work involving static and dynamic methods for calculating betweenness centrality. Section 3 presents our algorithms for dynamic betweenness centrality on the GPU and discusses performance optimization. Section 4 presents the benchmark graphs and target architecture used for this study. Section 5 presents our experimental results and analyses. Finally, section 6 concludes and discusses future work.

II. RELATED WORK

Betweenness centrality (BC) ranks the importance of a given vertex based on the number of shortest paths on which this vertex lies. Contemporary applications of betweenness centrality involve the study of AIDS within sexual networks [4], lethality in biological networks [5], community detection [6], and the analysis of the human brain [7]. The following subsections review various algorithms for computing static and dynamic betweenness centrality for large graphs.

A. Definitions

Given a graph $G = (V, E)$ with a set V of $n = |V|$ vertices and a set $E \subseteq V \times V$ of $m = |E|$ edges we define the following metrics. Let $d_s(t)$ be the distance of the shortest path from the source vertex s to vertex t as is found by a Breadth-First Search (BFS). By definition $d_s(s) = 0$. Let σ_{st} represent the number of shortest paths starting at vertex s and ending at vertex t. Next, let $\sigma_{st}(v)$ denote the number of such shortest paths that pass through a particular vertex v. The betweenness centrality of a given vertex v can now be defined as:

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

Note that this metric is computationally intensive, as it requires the solution of the all-pairs shortest paths problem. Intuitively, the BC score of a vertex implies how often it is used as a connection on the shortest path of pairs of other vertices. Typically the vertices with the highest BC scores are of particular interest and the relative ranking of the vertices tends to be more informative than the magnitude of their scores [8].

B. Brandes's Algorithm

The fastest known sequential algorithm for computing betweenness centrality was presented by Brandes in 2001 [9]. He defines the *dependency* of a vertex v for a given source node s as:

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

In other words, the dependency of v is a function of the dependency of its immediate successors w. This recursive relationship allows for an efficient computation that avoids unnecessary additions from vertices that are not on shortest paths [10]. Using this definition, the BC score of a given vertex v can be computed as:

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

Algorithm 1 shows Brandes's approach to computing exact BC scores. For each node s in the graph, three stages are processed:

1) Initialization
2) Shortest Path Calculation
3) Dependency Accumulation

The first of these stages initializes local data structures. These data structures include a queue and stack for keeping progress during the later stages, a list of immediate predecessors for each element, the distance (d) of each element from the current root, the number of shortest paths (σ) from the root to each element, and the dependency (δ) of each element. The second stage is a Breadth-First Search (BFS) traversal that starts at the current root and finds the distance and the number of shortest paths from the current root to all other roots. Finally,

the third stage visits nodes in the reverse order of the BFS traversal and finds the fraction of shortest paths that pass through each particular vertex out of all shortest paths. For undirected graphs, the algorithm has $O(mn)$ time complexity and $O(m + n)$ space complexity.

Algorithm 1: Static Betweenness Centrality (Brandes) [9]

1   $BC[v] \leftarrow 0, \forall v \in V$
2   **for** $s \in V$ **do**
3   **Stage 1: Initialization**
4    $S \leftarrow$ empty stack; $Q \leftarrow$ empty queue
    $P[w] \leftarrow$ empty list, $\forall w \in V$
5    $d[t] \leftarrow \infty, \forall t \in V$
6    $d[s] \leftarrow 0$
7    $\sigma[t] \leftarrow 0, \forall t \in V$
8    $\sigma[s] \leftarrow 1$
9    $\delta[t] \leftarrow 0, \forall t \in V$
10   **Stage 2: Shortest Path Calculation**
11   $Q.enqueue(s)$
12   **while** $!Q.empty()$ **do**
13    $v \leftarrow Q.dequeue()$
14    $S.push(v)$
15   **for** $w \in neighbors(v)$ **do**
16   //w found for the first time?
17   **if** $d[w] = \infty$ **then**
18      $Q.enqueue(w)$
19      $d[w] \leftarrow d[v] + 1$
20   //Shortest path to w via v?
21   **if** $d[w] = d[v] + 1$ **then**
22      $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
23      $P[w].insert(v)$

24   **Stage 3: Dependency Accumulation**
25   **while** $!S.empty()$ **do**
26    $w \leftarrow S.pop()$
27   **for** $v \in P[w]$ **do**
28     $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
29   **if** $w \neq s$ **then**
30     $BC[w] \leftarrow BC[w] + \delta[w]$

Since exact centrality computation on current workstations is infeasible for large-scale graphs, a number of methods for approximating BC scores have been proposed. One of these methods is to choose a subset of vertices to process in the outermost for loop (line 2) of Algorithm 1 [11]. Since the iterations of this for loop can all be processed in parallel, this approach scales well on multiple processors and is used to calculate BC scores for large graphs in this paper. If k nodes are chosen as *source nodes* (i.e. nodes to be processed in the outermost for loop of Algorithm 1) then the time complexity for approximating BC scores reduces to $O(mk)$. Since $k \leq n$, approximating the algorithm can be significantly faster than computing it exactly, depending on the values of k and n. For a detailed analysis regarding the accuracy of this approximation, we refer the reader to [11].

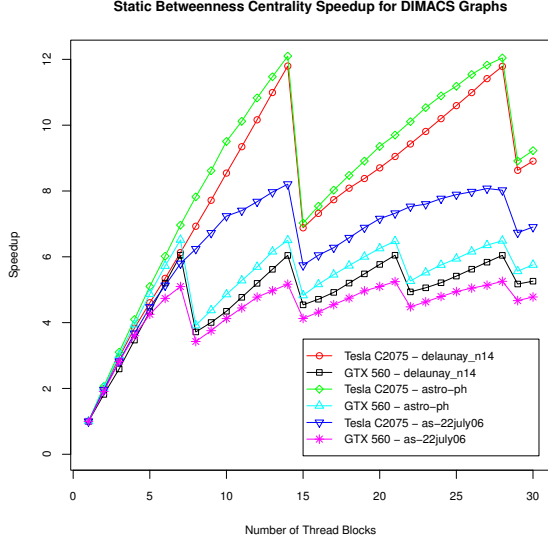**Static Betweenness Centrality Speedup for DIMACS Graphs**

Fig. 1. BC speedup relative to one thread block

C. Parallel Implementations

Parallelism is another way to reduce the high computational cost of centrality metrics. Sariyüce et al. propose a heterogeneous implementation that extracts vertices of degree 1 from the graph, showing that only minor modifications to the calculation are necessary after this removal [12]. Jia et al. propose a GPU implementation of betweenness centrality in [13]. The work of Jia et al. investigates the difference in performance between node and edge-based parallelism, concluding that edge-based parallelism gets better memory throughput and thus better performance. We revisit this comparison for our dynamic algorithms in Section III. The optimal number of CUDA thread blocks was also investigated, but in less detail. The authors concluded that the optimal number of thread blocks is the number of Streaming Multiprocessors (SMs) on the GPU. Conventional wisdom with regard to GPU programming says that each SM should have multiple active thread blocks [14]; however, the claim from [13] seems to suggest that this strategy isn't ideal for irregular algorithms since the memory bus will become saturated.

To substantiate this claim and determine the best ratio of thread blocks to SMs we run a static (and exact) betweenness centrality computation for a varying number of thread blocks and compare performance. We use three graphs from the DIMACS challenge as our input [15], using the largest graphs that are still feasible for an exact computation with contemporary hardware (i.e. graphs with tens of thousands of vertices). Figure 1 shows the speedup of static betweenness centrality relative to using one thread block for two GPUs: a GTX 560 with 7 SMs and a Tesla C2075 with 14 SMs. It is clear that the best performance is obtained by setting the number of thread blocks to be equal to the number of SMs or a multiple thereof, as concluded in [13]. For the graphs that we tested, we found that the performance of having one thread block per SM was slightly faster or about the same as the performance of having multiple thread blocks per SM.

Hence we delegate one thread block per SM for the algorithms presented in the upcoming sections.

D. Dynamic Approaches

Three different algorithms for dynamic betweenness centrality have been proposed in recent literature. Lee et al. propose QUBE, an algorithm that updates BC scores by determining which vertices have BC values that may change, thus avoiding the full all-pairs shortest paths computation [16]. Kas et al. use a Java-based graph library to improve upon this result by directly updating the auxiliary data required by the algorithm [17]. Finally, Green et al. provide a high-performance implementation along with formal proofs and algorithms for the various scenarios that can occur when inserting or removing an edge [10]. Note that all of these approaches are sequential, making our implementation the first parallelized version of dynamic betweenness centrality. Our implementation in this paper will most closely resemble the approach by Green et al. [10].

1) Update Scenarios: In this section the various scenarios for updating betweenness centrality scores are discussed in detail. Readers interested in formal proofs can find them in [10]. We restrict our focus to edge insertions since many real-world networks only experience growth and do not shrink. For example, graphs resembling co-authorship will only expand as time progresses. Furthermore, it has been shown that edge removal updates require similar algorithmic techniques to edge insertion updates [16]. Thus, the lessons learned from focusing on edge insertions are directly applicable to edge deletions. It is also noteworthy that a node insertion causes no change to existing BC scores. A newly inserted node belongs to its own connected component (equivalently, has no incoming or outgoing edges) and thus has a BC score of 0. The new node will only affect the BC scores of other nodes once edges connect the new node to other connected components in the network.

To update the BC scores, we must store supplemental global data to the scores alone. For each source vertex s, the variables $d_s(t)$, σ_{st}, and $\delta_s(t)$ are preserved $\forall t \in V$. This added storage increases the space complexity to $O(n^2)$ for exact BC computation and $O(kn)$ for approximate BC computation using k source vertices; however, as we will show in Section V the performance gain is well worth the extra space.

Formally, an edge insertion $e = (u, v)$ creates a new graph $G' = (V, E')$ where $E' = E \cup \{e\}$. For each source vertex, one of the following three scenarios will occur, depending on the relation between u and v *before* the edge is inserted:

- Case 1: $|d_s(u) - d_s(v)| = 0$. The nodes connected to the inserted edge are the same distance from the source node. In terms of performance, this scenario is ideal because no additional work needs to be done for this source vertex (the other source vertices may require work, however). The reason that no additional work needs to be done is that the distances of u and v from the source do not change and no additional shortest paths are created. Note that this case can actually occur for two slightly different reasons: one when u, v, and s all belong to the same connected component
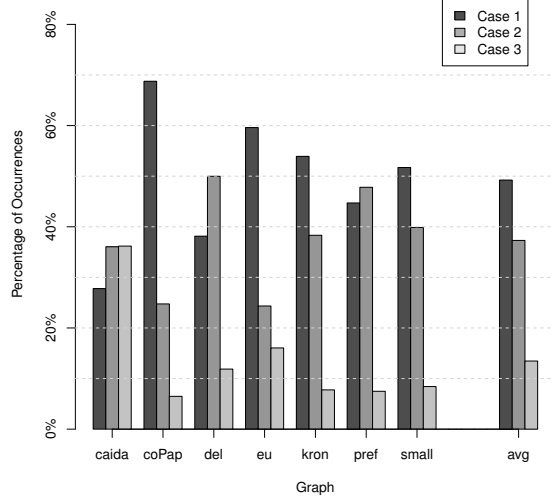
Fig. 2. Distribution of scenarios for the graphs used in this study

and another when neither u nor v belongs to the same connected component as s.

- Case 2: $|d_s(u) - d_s(v)| = 1$. The nodes connected to the inserted edge are on adjacent levels. While none of the distances from the source vertex will change, it is possible that the number of shortest paths have changed and thus centrality scores may also change.

- Case 3: $|d_s(u) - d_s(v)| > 1$. The nodes connected to the inserted edge are greater than one level apart. In this case distances from the source vertex will change and shortest paths may have changed. Hence, centrality scores will need to be updated. Note that this case can actually occur for two slightly different reasons: one when u, v, and s all belong to the same connected component and another when either u or v (but not both) belong to the same connected component as s.

Figure 2 motivates the importance of the implementation for Case 2 with regard to overall performance of the dynamic analytic. For each edge insertion, each source node will face one of the three scenarios previously described. The data in Figure 2 reflects 100 edge insertions for each input graph. For each edge insertion, every source node in the graph faces one of the three scenarios described above. Therefore if k source nodes are used to approximate the BC scores of 100 edge insertions there will be $100k$ scenarios distributed among the 3 cases described above (up to $100n$ for the exact computation). Figure 2 shows how these distributions vary for the set of graphs used in this study. Recall from above that for Case 1, no work needs to be done. We can see that Case 2 represents 37.3% of all scenarios and 73.5% of the scenarios that require actual work (Cases 2 and 3) for this set of graphs. Hence, for the rest of this paper we focus our analysis on Case 2, noting that our techniques generalize and can be applied to Case 3 and, oftentimes, parallel graph algorithms in general.

Algorithm 2 from Green et al. [10] shows how to update the intermediate variables and centrality scores for Case 2. In addition to d, σ, and δ, a few additional variables are

---

Algorithm 2: Dynamic Betweenness Centrality Case 2 (Green et al.) [10]

Input: Source node s and an inserted edge from u_{low} to u_{high}

1 **Stage 1: Initialization**
2 $Q \leftarrow$ empty queue
3 $QQ[level] \leftarrow$ empty queue, $level = 0, 1, \ldots, n-1$
4 $t[v] \leftarrow untouched, \forall v \in V \setminus \{u_{low}\}$
5 $t[u_{low}] \leftarrow down$
6 $\hat{\sigma}[v] \leftarrow \sigma[v], \forall v \in V \setminus \{u_{low}\}$
7 $\hat{\sigma}[u_{low}] \leftarrow \sigma[u_{low}] + \sigma[u_{high}]$
8 $\hat{\delta}[v] \leftarrow 0, \forall v \in V$
9 **Stage 2: Shortest Path Calculation**
10 $Q.enqueue(u_{low})$
11 $QQ[d[u_{low}]].enqueue(u_{low})$
12 **while** $!Q.empty()$ **do**
13      $v \leftarrow Q.dequeue()$
14   **for** $w \in neighbors(v)$ **do**
15   **if** $d[w] = d[v] + 1$ **then**
16   **if** $t[w] = untouched$ **then**
17                  $Q.enqueue(w)$
18                  $QQ[d[w]].enqueue(w)$
19                  $t[w] \leftarrow down$
20              $\hat{\sigma}[w] \leftarrow \hat{\sigma}[w] + (\hat{\sigma}[v] - \sigma[v])$

21 **Stage 3: Dependency Accumulation**
22 **while** $level > 0$ **do**
23   **while** $!QQ[level].empty()$ **do**
24          $w \leftarrow QQ.dequeue()$
25   **for** $v \in neighbors(w)$ **do**
26   **if** $d[w] = d[v] + 1$ **then**
27   **if** $t[v] = untouched$ **then**
28                      $QQ[level - 1].enqueue(v)$
29                      $t[v] \leftarrow up$
30                      $\hat{\delta}[v] \leftarrow \delta[v]$
31                  $\hat{\delta}[v] \leftarrow \hat{\delta}[v] + \frac{\hat{\sigma}[v]}{\hat{\sigma}[w]}(1 + \hat{\delta}[w])$
32   **if** $t[v] = up \wedge (v \neq u_{high} \vee w \neq u_{low})$ **then**
33                      $\hat{\delta}[v] \leftarrow \hat{\delta}[v] - \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
34   **if** $w \neq s$ **then**
35                      $BC[w] \leftarrow BC[w] + \hat{\delta}[w] - \delta[w]$

36      $level \leftarrow level - 1$
37 $\sigma[v] \leftarrow \hat{\sigma}[v], \forall v \in V$
38 **for** $v \in V$ **do**
39   **if** $t[v] = untouched$ **then**
40          $\delta[v] \leftarrow \hat{\delta}[v]$

introduced. Let t_v denote the stage of the update algorithm in which some vertex v was found. If $t_v = down$ then v was found in the shortest path (downward) calculation stage, if $t_v = up$ then v was found in the dependency accumulation stage, and if $t_v = untouched$ then v was not found in either stage. Also, let $\hat{\sigma}_{sv}$ and $\hat{\delta}_s(v)$ be the updated values of σ_{sv} and $\delta_s(v)$ after the insertion, respectively. Note that the algorithm takes the source node s as well as the endpoints u_{low} and u_{high} of the inserted edge. Since u_{low} and u_{high} belong to adjacent levels, one of them must be closer to s than the other. We refer to this closer node as being "higher" up in the BFS tree of s and hence call it u_{high}. Similarly, the other endpoint of the edge is "lower" in the BFS tree of s so we refer to it as u_{low}.

Lines 1 through 8 initialize these data structures. Note that a multi-level queue (QQ) is used in lieu of a stack because it is possible for nodes to be added to this "stack" in the dependency accumulation stage. The level order of the BFS tree from the source node s must be preserved as nodes are processed in the dependency accumulation stage. Processing in this stage begins with nodes that are the farthest away from s. If a node v at level i is pushed onto a stack rather than a multi-level queue in line 28 by a node w at level $i + 1$, the next node to be popped would be v instead of the remaining nodes at level $i + 1$ that have yet to be processed but must be processed first for correctness. Line 7 records the updated number of shortest paths for u_{low} due to the edge insertion. Since an edge is inserted from u_{high} to u_{low} all of the shortest paths that pass from s to u_{high} must also pass through u_{low} (because the new edge is the shortest path from u_{high} to u_{low}). Therefore $\hat{\sigma}[u_{low}]$ is initialized to $\sigma[u_{low}] + \sigma[u_{high}]$.

Lines 9 through 20 update the number of shortest paths from s to all other nodes due to the insertion of the new edge. Since we know that the number of shortest paths for nodes between s and u_{high} will not change, we can start the BFS traversal downward from u_{low}. Note that this approach does not explicitly store predecessors as is done in line 23 of Algorithm 1. Instead, the dependency accumulation stage looks at all neighbors of nodes popped from the multi-level queue and checks to see that a given neighbor is a predecessor before subsequent processing (line 26). Although this method generates some additional work it has been shown to save $O(E)$ memory in addition to showing speedups in practice [18].

Finally, lines 21 through 40 update the BC scores of each node. Since the preceding stage potentially changed the number of shortest paths from the root s to other nodes and since the dependency is a function of the number of shortest paths, the values of the dependency will potentially change as well. Line 31 adds the correct contribution of w to the dependency of it's predecessor v and line 33 subtracts out the prior contribution of w to the dependency of v, which is now incorrect due to the edge insertion. Lines 37 through 40 copy the updated (local) values of shortest paths and dependency to global variables to be used for the next update.

III. Dynamic Betweenness Centrality on the GPU

In this section we present several of our GPU implementations for dynamic betweenness centrality computations. Since figuring out which case each source node has to compute
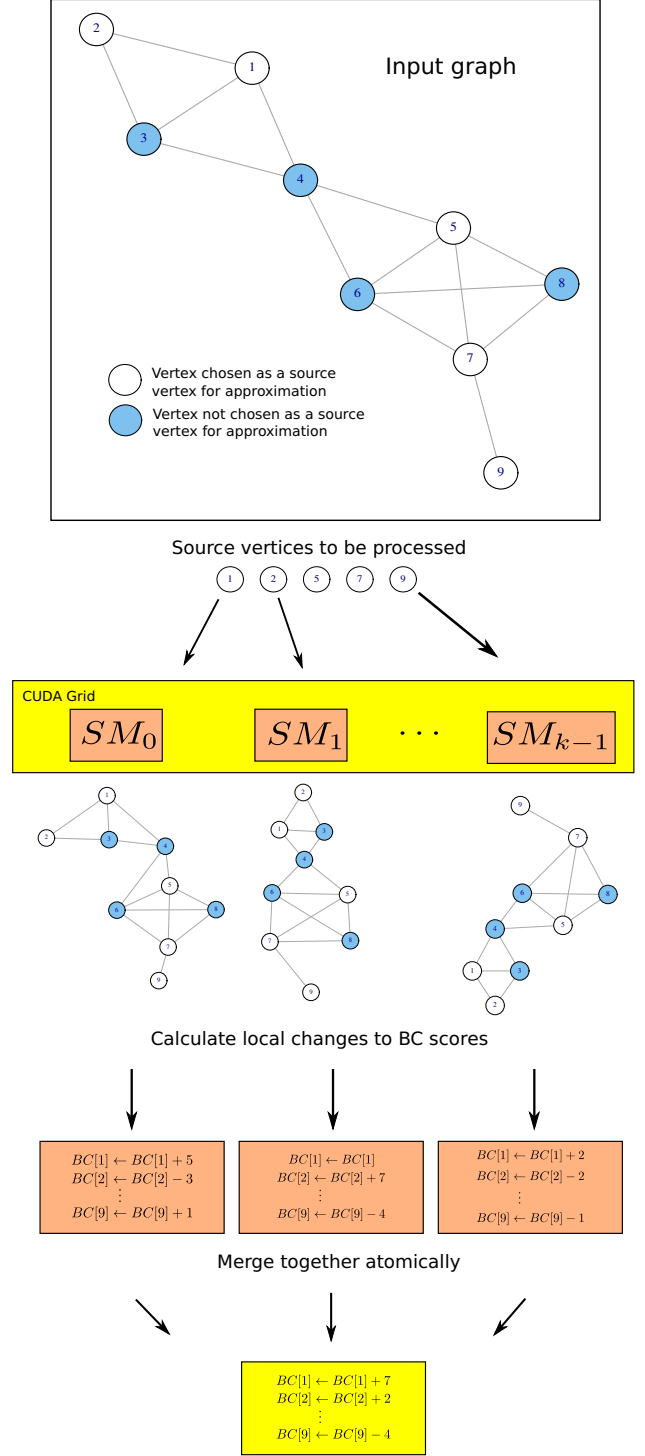

Fig. 3. Decomposition of work to parallel compute units

is trivial, we focus on the algorithmic challenges of the cases themselves. Again, our discussion focuses on Case 2 (edge insertion between nodes on adjacent levels) due to the motivation from Figure 2 and its discussion in the previous section.

Similar to previous work [13], we assign the maximum number of threads per block and set the number of thread blocks to be equal to the number of streaming multiprocessors for all kernels. Each thread block takes advantage of the available coarse-grained parallelism by handling independent source vertices while the threads within a block take advantage of fine-grained parallelism by traversing graph edges and updating state concurrently.

Algorithm 3: Kernel for initialization of local variables

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

1   **Stage 1: Initialization**
2   **for** $v \in V$ **do in parallel**
3   **if** $v = u_{low}$ **then**
4       $t[v] \leftarrow down$
5       $\hat{\sigma}[v] \leftarrow \sigma[v] + \sigma[u_{high}]$
6   **else**
7       $t[v] \leftarrow untouched$
8       $\hat{\sigma}[v] \leftarrow \sigma[v]$
9     $\hat{\delta}[v] \leftarrow 0$

Figure 3 illustrates this concept. To approximate BC, a subset of the graph's vertices are chosen at random and used as root nodes for shortest path calculations (shown as the unfilled vertices of the input graph in Figure 3). Each Streaming Multiprocessor (SM) takes one source vertex and performs a BFS to calculate the number of shortest paths from that vertex to all other vertices in the graph. These shortest path calculations are independent among SMs and can hence be performed in parallel without communication overhead. The dependency accumulation stage is also independent among SMs with the exception of the final update to the BC value itself. To update the global array holding the BC scores each SM adds its changes atomically, preventing data races. Since GPUs currently tend to have a small number of SMs (< 50) and since these additions are not necessarily performed concurrently (because one SM can finish its updates independently of the others), there is little contention for global memory resources for these atomic additions. Thus the use of atomic operations in this instance is admissible as it has negligible overhead.

Throughout this section we compare two approaches of fine-grained parallelism: *edge-based* and *node-based*. *Edge-based* parallelism assigns one thread to each edge in the graph, which results with a greater number of work units that consist of a small, roughly equivalent amount of work. *Node-based* parallelism, on the other hand, assigns one thread to each vertex in the graph, which results in fewer work units that have varying size. Note that both methods use the same number of threads, but map threads to work differently. Since there are typically more vertices and edges in a graph than available threads, each thread will process multiple units of

work. For example, if there are 1000 available threads and 4000 edges in the graph, the edge-based method will provide each thread with 4 edges to process. The edge-based approach has better load balancing and has been shown to generate greater memory throughput for static betweenness centrality on the GPU [13] whereas the node-based approach has less contention over shared resources. Both of these approaches initialize local variables in parallel in the same way, as shown in Algorithm 3.

A. Updating the Number of Shortest Paths

Algorithm 4: Edge-based Parallel Shortest Path Calculation Kernel

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

1   **Stage 2: Shortest Path Calculation**
2   **shared** $current\_depth \leftarrow d[u_{low}]$
3   **shared** $done \leftarrow false$
4   **while** $!done$ **do**
5     $done \leftarrow true$
6   **for** $(v,w) \in E$ **do in parallel**
7   **if** $d[v] = current\_depth$ **then**
8   **if** $d[w] = current\_depth + 1$ **then**
9   **if** $t[w] = untouched$ **then**
10            $t[w] \leftarrow down$
11            $done \leftarrow false$
12          $atomicAdd(\&\hat{\sigma}[w], \hat{\sigma}[v] - \sigma[v])$

13     $barrier()$
14     $current\_depth \leftarrow current\_depth + 1$

Algorithm 4 shows GPU pseudocode to update the number of shortest paths from the source node s to all other nodes in the graph using edge-based parallelism. The **shared** keyword is used to denote variables that are explicitly stored in the GPU's fast scratchpad (or shared) memory. Threads within an SM will see the same value of shared variables while threads belonging to different SMs will not. Note that explicit queues aren't necessary as shared memory and synchronization are used to ensure that vertex frontiers (depths) are processed in the correct order. It is possible for multiple threads to successfully execute Line 10 and set $t[w]$ to $down$, leading to a data race; however, this data race is considered benign as it has no effect on program output. Line 12 requires an atomic (serialized) write to $\hat{\sigma}[w]$ to prevent a data race. Otherwise, this calculation is exactly the same as the one in Line 20 of Algorithm 2.

Alternatively, Algorithm 5 shows GPU pseudocode that achieves the same result using node-based parallelism. We introduce three different arrays that act as queues in lines 2-6. The Q array holds nodes that are being processed in the current level of the BFS traversal. The $Q2$ array is used to hold vertices found in the current level of the BFS traversal. These vertices are transferred to Q (line 26) and are explored in the next level of the BFS traversal. Separate queues are necessary because all nodes at the current level must be processed before any nodes at the following level are to be processed to ensure correctness. Finally, the QQ array holds nodes that are to be processed during the dependency accumulation, analogous to the multi-level queue used in Algorithm 2.

Algorithm 5: Node-based parallel Shortest Path Calculation Kernel

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

1 Stage 2: Shortest Path Calculation
2 **shared** $current\_depth \leftarrow d[u_{low}]$
3 $Q[0] \leftarrow u_{low}$
4 $Q_{len} \leftarrow 1$
5 $Q2_{len} \leftarrow 0$
6 $QQ[0] \leftarrow u_{low}$
7 $QQ_{len} \leftarrow 1$
8 **while** $true$ **do**
9   **for** $tid \leftarrow 0 \dots Q_{len} - 1$ **do in parallel**
10 $v \leftarrow Q[tid]$
11   **for** $w \in neighbors(v)$ **do**
12   **if** $d[w] = d[v] + 1$ **then**
13   **if** $t[w] = untouched$ **then**
14 $t[w] \leftarrow down$
15 $i \leftarrow atomicAdd(\&Q2_{len}, 1)$
16 $Q2[i] \leftarrow w$
17 $atomicAdd(\&\hat{\sigma}[w], \hat{\sigma}[v] - \sigma[v])$
18 $barrier()$
19   **if** $Q2_{len} = 0$ **then**
20 $break$
21   **else**
22 $remove\_duplicates(Q2, Q2_{len})$
23 $Q_{len} \leftarrow Q2_{len}$
24 $Q2_{len} \leftarrow 0$
25   **for** $tid \leftarrow 0 \dots Q_{len} - 1$ **do in parallel**
26 $Q[tid] \leftarrow Q2[tid]$
27 $i \leftarrow atomicAdd(\&QQ_{len}, 1)$
28 $QQ[i] \leftarrow Q2[tid]$
29 $barrier()$
30 **for** $v \in V$ **do**
31 $atomicMax(\&current\_depth, d[v])$

Using this approach, the number of threads needed to process an iteration is simply the number of nodes that currently reside in Q (which is stored in the variable Q_{len}). In contrast, the edge-based parallel approach spawns $|E|$ threads for every level of the search, regardless of the amount of work needed to be done. Hence the edge-based approach, despite being conceptually simpler and more convenient to program, generates significantly more accesses to memory, most of which are futile.

It is important to recognize that $Q2$ may have duplicate entries whereas Q and QQ will not. An atomic operation could be used to test and set $t[w]$ on line 13, ensuring that only one thread places w into $Q2$ on line 16. We avoid this atomic operation by allowing multiple threads to insert the same node into $Q2$ and removing duplicate entries from $Q2$ (line 22) before transferring $Q2$ to Q for the next iteration of the search. Note that we pass $Q2_{len}$ to the $remove\_duplicates()$ subroutine because the removal of duplicates reduces the size of the queue. Duplicate entries are removed from $Q2$ by the following procedure (similar to Merrill et. al [19]):

1) Sort the elements of $Q2$. In our implementation we use bitonic sort, though we consider this choice to have a negligible impact on performance because $Q2_{len}$ is typically much smaller than n.

2) Compare the value at index $i - 1$ from the value at index i of the sorted array. Using an additional array, mark index i with the value *true* if the compared values are equivalent. Else, mark *false*. This output represents which indices of $Q2$ correspond to unique elements.

3) Perform a prefix sum on the above result to determine which indices into Q each corresponding unique element of $Q2$ should be placed and to find the number of unique entries in the queue (i.e. Q_{len} for the next search iteration).

After the above procedure, the unique entries in $Q2$ are transferred to Q for the next BFS iteration (line 26). These entries are also added to QQ (line 28) for the dependency accumulation stage. Lines 30 and 31 set the appropriate distance of the furthest processed vertex from s as the starting point of the dependency accumulation method discussed in the next section.

Algorithm 6: Edge-based Parallel Dependency Accumulation Kernel

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

1 Stage 3: Dependency Accumulation
2 **while** $current\_depth > 1$ **do**
3   **for** $(v, w) \in E$ **do in parallel**
4   **if** $d[v] = current\_depth$ **then**
5   **if** $d[w] = current\_depth - 1$ **then**
6 $dsv \leftarrow 0$
7   **if** $atomicCAS(\&t[v], untouched, up) = untouched$ **then**
8 $dsv \leftarrow dsv + \delta[v]$
9 $dsv \leftarrow dsv + \frac{\hat{\sigma}[v]}{\hat{\sigma}[w]} * (1 + \hat{\delta}[w])$
10   **if** $t[v] = up \wedge (v \neq u_{high} \vee w \neq u_{low})$ **then**
11 $dsv \leftarrow dsv - \frac{\sigma[v]}{\sigma[w]} * (1 + \delta[w])$
12 $atomicAdd(\&\hat{\delta}[v], dsv)$
13 $barrier()$
14 $current\_depth \leftarrow current\_depth - 1$

B. Updating the Dependency Accumulation

Once the shortest path calculation has been updated, it remains to update the dependencies and the BC scores themselves. Algorithm 6 shows an edge-based parallel implementation of the dependency accumulation. Continuing from where Algorithm 4 left off, vertices of decreasing distance from the source are processed one level at a time. Line 7 requires an atomic operation that ensures that only the first thread to attempt to successfully set $t_v = up$ executes Line 8. The $atomicCAS()$ function does an atomic compare and swap. If $t[v] = untouched$, the function sets $t[v] = up$ and returns $untouched$. Otherwise, the function doesn't change the

contents of $t[v]$ and returns the value of $t[v]$ provided to the function, causing the if statement on Line 7 to evaluate to $false$ so that Line 8 will not execute. The register dsv is used to accumulate all changes to $\hat{\delta}[v]$ brought upon by w so that only one atomic addition (Line 12) to update $\hat{\delta}[v]$ is necessary.

Algorithm 7: Node-based parallel Dependency Accumulation Kernel

---

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

1   **Stage 3: Dependency Accumulation**
2   **while** $current\_depth > 1$ **do**
3   **for** $tid \leftarrow 0 \ldots QQ_{len} - 1$ **do in parallel**
4     $w \leftarrow QQ[tid]$
5   **if** $d[w] = current\_depth$ **then**
6   **for** $v \in neighbors(w)$ **do**
7   **if** $d[v] = current\_depth - 1$ **then**
8        $dsv \leftarrow 0$
9   **if** $atomicCAS(\&t[v], untouched, up) = untouched$ **then**
10         $dsv \leftarrow dsv + \delta[v]$
11         $i \leftarrow atomicAdd(\&Q2_{len}, 1)$
12         $QQ[i + QQ_{len}] = v$
13        $dsv \leftarrow dsv + \frac{\hat{\sigma}[v]}{\hat{\sigma}[w]} * (1 + \hat{\delta}[w])$
14   **if** $t[v] = up \wedge (v \neq u_{high} \vee w \neq u_{low})$ **then**
15         $dsv \leftarrow dsv - \frac{\sigma[v]}{\sigma[w]} * (1 + \delta[w])$
16        $atomicAdd(\&\hat{\delta}[v], dsv)$
17    $barrier()$
18    $QQ_{len} \leftarrow QQ_{len} + Q2_{len}$
19    $Q2_{len} \leftarrow 0$
20    $current\_depth \leftarrow current\_depth - 1$

The corresponding node-based parallel dependency accumulation is shown in Algorithm 7. To simulate the multi-level queue seen in Algorithm 2 we place processed vertices from all levels of the BFS traversal into one array (QQ), as shown in Algorithm 5. To process this array in level synchronous order, we have threads extract vertices from the array and check if the level of the extracted vertices matches the current level that is to be processed, as shown on line 5. If we find a node that wasn't touched in the shortest path calculation stage we can safely add it to the end of QQ (line 12) and safely process it concurrently with other nodes at its level because of this check. Again, since only QQ_{len} threads are performing work whereas $|E|$ threads are performing work in the edge-based approach, the node-based approach exhibits significantly less memory traffic. Since QQ_{len} is the number of nodes to be processed at all levels and not just the current level, even the node-based approach performs some unnecessary work. However, we will show in Section V that the amount of this extra work is tremendously small in virtually all cases.

Once the dependency accumulation kernel is complete the updated values of the dependency are used to adjust the BC scores and the local variables $\hat{\sigma}$ and $\hat{\delta}$ are copied to

---

Algorithm 8: Kernel to update global variables

Input: Source node s and endpoints u_{low} and u_{high} of the inserted edge

1   **for** $v \in V$ **do in parallel**
2   **if** $v \neq s \wedge t[v] \neq untouched$ **then**
3     $atomicAdd(\&BC[v], \hat{\delta}[v] - \delta[v])$
4    $\sigma[v] \leftarrow \hat{\sigma}[v]$
5   **if** $t[v] \neq untouched$ **then**
6     $\delta[v] \leftarrow \hat{\delta}[v]$

---

their respective global variables σ and δ for the next update. Algorithm 8 shows how to perform this task in parallel. Similar to the initialization in Algorithm 3, both the edge and node-based approaches complete this task in the same way.

IV. EXPERIMENTAL SETUP

Table I shows the graph inputs used throughout this study. Again, we focus on approximate calculation of betweenness centrality since we are interested in the analysis of large graphs. The graph data was downloaded from the 10th DIMACS challenge [15]. These graphs were chosen based on size, diversity, and relevance to dynamic graph analytics. The set of graphs consists of real-world and random graphs and different classes of graphs are represented, such as small-world and scale-free graphs.

Single-threaded CPU experiments are implemented in C++ and compiled with `gcc -O3 -std=c++0x` flags and GPU experiments are implemented using CUDA and compiled with `nvcc -O3 -arch=sm_21` flags. The CPU used in this study is an Intel Core i7-2600K Processor running at 3.4GHz with an 8MB cache and 16GB of DRAM. The GPU used in this study is an Nvidia Tesla C2075 with 14 streaming multiprocessors each consisting of 32 stream processors that run at 1.15 GHz. The Tesla C2075 has 6GB of GDDR5 memory and has compute capability 2.0.

For each dynamic computation, 100 edges are chosen at random to be removed from the graph, similar to the approaches used in [16] and [17]. These edges are then reinserted into the graph one at a time and the analytic is updated. We choose $k = 256$ source nodes for approximation of BC, also at random, following the guidelines of the DARPA Scalable Synthetic Compact Applications (SSCA) benchmark suite [22]. To ensure that the proposed experiments are fair, the BC scores are approximated by all implementations: the dynamic CPU baseline from Green et al. [10], our dynamic node and edge parallel GPU algorithms, and the static BC computation on the GPU from Jia et al. [13]. For each experiment we compare the results of the baseline and our algorithms to ensure that both yield the same results. We neglect the cost of updating the graph, choosing to focus on the design and performance of the analytic itself. Several techniques for dynamically updating graph data structures at a small amortized cost are disccused in [23].

V. EXPERIMENTAL RESULTS

Table II shows the speedup of our dynamic GPU BC implementations over the dynamic sequential CPU algorithm

TABLE I.   SUITE OF BENCHMARK GRAPHS

Name	Vertices	Edges	Significance
caidaRouterLevel (caida)	192,244	609,066	Internet Router Level Graph
coPapersCiteseer (coPap)	434,102	16,036,720	Social Network
delaunay_n20 (del)	1,048,576	3,145,686	Random Triangulation
eu-2005 (eu)	862,664	16,138,468	Web Crawl
kron_g500-simple-logn19 (kron)	524,288	21,780,787	Kronecker Graph
preferentialAttachment (pref)	100,000	499,985	Scale-free [20]
smallworld (small)	100,000	499,998	Logarithmic Diameter [21]

TABLE II.   COMPARISON OF DYNAMIC CPU AND DYNAMIC GPU ALGORITHMS

Graph	CPU Time (s)	Method	GPU Time (s)	Speedup
caida	1749.98	Edge	84.79	20.64x
		Node	**15.85**	**110.41x**
coPap	1080.81	Edge	762.81	1.41x
		Node	**20.49**	**52.75x**
del	4762.75	Edge	4611.52	1.03x
		Node	**196.48**	**24.24x**
eu	3991.27	Edge	591.20	6.75x
		Node	**71.23**	**56.03x**
kron	1951.86	Edge	1668.27	1.17x
		Node	**81.54**	**23.94x**
pref	380.77	Edge	62.73	6.07x
		Node	**10.38**	**36.68x**
small	360.82	Edge	29.14	12.38x
		Node	**7.20**	**50.11x**

TABLE III.   COMPARISON OF NODE PARALLEL GPU UPDATES TO GPU RECOMPUTATION

Graph	Recomputation (s)	Update (s)	Speedup
caida	1.99	Slowest: 0.3295	6.05x
		Average: 0.1585	12.58x
		Fastest: 0.0003	6095.09x
coPap	31.35	Slowest: 0.7242	43.31x
		Average: 0.2049	153.02x
		Fastest: 0.0003	94729.29x
del	99.60	Slowest: 10.8997	9.14x
		Average: 1.9648	50.69x
		Fastest: 0.0003	296436.91x
eu	21.40	Slowest: 3.0308	7.06x
		Average: 0.7123	30.04x
		Fastest: 0.0003	64445.53x
kron	38.69	Slowest: 1.5658	24.71x
		Average: 0.8154	47.45x
		Fastest: 0.2725	141.96x
pref	1.27	Slowest: 0.5907	2.15x
		Average: 0.1038	12.24x
		Fastest: 0.0603	21.07x
small	0.68	Slowest: 0.0978	6.98x
		Average: 0.0720	9.48x
		Fastest: 0.0350	19.49x

from Green et al. [10] for both the edge and node-based parallel methods. We can see that although the edge-based parallel method can significantly outperform the CPU in some cases, the node parallel method substantially improves upon this result. It is clear that the edge-based approach does not scale well to larger graphs because the amount of unnecessary work that it performs grows with the size of the graph. Since the edge-based approach assigns one thread for every edge in the graph and since only a small subset of edges need to be traversed for a specified iteration, the edge-based approach ends up with many threads that perform an unnecessary comparison for a branch instruction along with the loads it depends on. In contrast, the node-based method assigns one thread for every element in the queue being processed. In the shortest path calculation stage each of these elements has necessary work to complete, which means that this thread mapping is perfectly work efficient. In the dependency accumulation stage only a subset of these elements have necessary work to complete although the size of the queue is $O(n)$ (and in practice typically much smaller than n), which is significantly less than the number of edges in the graph, particularly for sparse graphs. Hence the node-based approach still provides a notably better mapping of threads to units of work. The node-based method performs well even for scale-free graphs such as *preferentialAttachment* with power-law degree distributions that can lead to severe workload imbalance among threads. We can see that our node-parallel GPU approach is up to 110x faster than the sequential CPU approach for the set of graphs used in this study.

In addition to providing speedups over a single-threaded CPU implementation of the dynamic algorithm, our approach also provides high performance in comparison to a full re-

computation of the analytic on the GPU. Table III compares the execution time of a static BC computation using the implementation available from [13] to the slowest, average, and fastest updates from our optimized node-parallel dynamic algorithm. We can see that, even in the worst case for each graph a dynamic update is faster than a static recomputation. Intuitively this result makes sense because the number of edges traversed by the static computation is an upper bound for the number of edges that need to be traversed by the dynamic computation.

The fastest updates occur when all source nodes see a Case 1 scenario. Since the Case 1 scenario requires no work, if all source nodes see this scenario then no source nodes require work and the edge insertion has no effect on BC scores. This ideal scenario took place for one or more of the edge insertions for *caidaRouterLevel*, *coPapersCiteseer*, *delaunay_n20*, and *eu-2005*. We can see from Table III that these updates all took ~ 0.0003 seconds, which is simply the amount of time necessary to discover that none of the BC scores will change due to the insertion. The speedups seen for this ideal case are essentially bounded by how long a recomputation takes, which is heavily dependent on the size of the graph. In contrast, the fastest cases for *kron_g500-simple-logn19*, *preferentialAttachment*, and *smallworld* require updates to BC scores from one or more of the source nodes. For example, the fastest edge insertion for *kron_g500-simple-*

logn19 led to a Case 1 scenario for 222 of the 256 source nodes used for approximation, which means that the remaining 34 source nodes required significant amounts of work. Hence, we see the large difference between the fastest cases between the various input graphs.

More typically, many or even all source nodes can require significant work. This result is more likely to occur in practice because it is unlikely for two nodes to be on the same level of a breadth-first search from the perspective of every other node in the graph. However, even the slowest graph updates are preferable to a full recomputation of the BC scores, with speedups ranging from 2x to 43x for the graphs used in this study. The key takeaway is that the updates can ignore the portion of the graph between the root node and u_{high} when counting the number of shortest paths as the insertion of an edge below these vertices cannot create new shortest paths from the root to these vertices. Depending on the distance from the root node to the nodes connecting the inserted edge, the number of shortest path computations that can be neglected in this way can be very significant. Hence, the amount of time that a given update takes in not solely dependent on how many of the source nodes require work; it is also dependent on *how much* work each of those source nodes requires. The scatterplot in Figure 4 illustrates this concept. For each occurrence of a Case 2 scenario in each graph we record the number of nodes that are "touched" (i.e. $\{i \in V \mid t[i] \neq untouched\}$). We divide these counts by the total number of nodes in the graph and sort them from least to greatest to see the distribution.

The results shown in Figure 4 are quite surprising. Of the 62,844 Case 2 scenarios encountered across the graphs used in this study, the largest percentage of nodes that were touched was approximately 35%. Perhaps even more surprising is that a vast majority of the Case 2 scenarios touched an extremely small portion of the graph, as evidence by the dense collection of points toward the bottom of the figure. These results strongly motivate the need for proper mapping of threads to units of work in high performance algorithm design for dynamic graph analytics. Through a combination of being able to detect when updates to the graph will not affect BC scores and avoiding unnecessary accesses to memory and redundant computation when graph updates do affect BC scores we achieve an average of a 45x speedup over a full recomputation of BC scores on the GPU across all graphs used in this study.

VI. Conclusions

In this paper we compared two GPU implementations of dynamic betweenness centrality, leveraging analyses from a wealth of related work. To our knowledge, this is the first time GPUs have been used for the analysis of time-varying graphs. By comparing these two different decompositions of threads to units of work, we show that keeping explicit track of the work that needs to be done is a vastly superior strategy. Although our approach uses atomic operations that serialize certain accesses to memory, we show that these memory locations are typically in low contention among threads because a surprisingly small number of nodes are affected by each update. Our approach achieves up to a 110x speedup over a CPU implementation of the algorithm and can update the analytic 45x faster on average than a static recomputation of the analytic on the GPU.
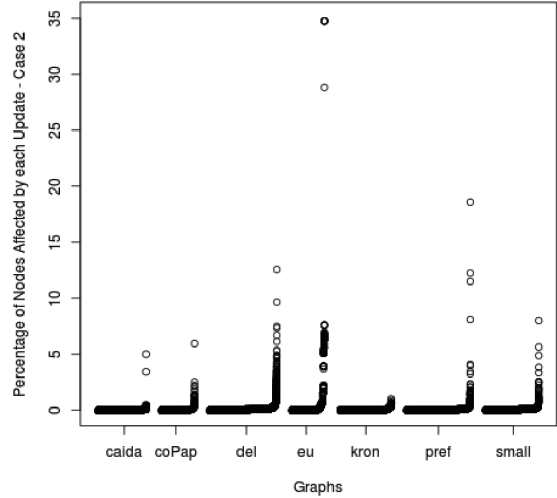

Fig. 4. Portion of the graph that is touched for each Case 2 scenario

A multitude of opportunities exist to extend this area of work. If a linear space dynamic betweenness centrality algorithm exists, it would allow scaling of the techniques in this paper to significantly larger graphs. Further performance improvements can be attained with multi-GPU, heterogeneous, or distributed implementations of this algorithm. The vast amount of coarse-grained parallelism that exists should allow for excellent strong scaling for such implementations. Finally, there are plenty of other graph algorithms that can benefit from either dynamic implementations or parallelism on multi-core CPUs and GPUs.

References

[1] 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 *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–7.

[2] S. Porta, V. Latora, F. Wang, E. Strano, and A. Cardillo, "Street centrality and densities of retail and services in bologna, italy," 2009.

[3] R. Nasre, M. Burtscher, and K. Pingali, "Atomic-free irregular computations on gpus," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 96–107.

[4] 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.

[5] H. Jeong, S. P. Mason, A. L. Barabsi, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, no. 6833, pp. 41–42, May 2001.

[6] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[7] M. Rubinov and O. Sporns, "Complex network measures of brain connectivity: Uses and interpretations," *NeuroImage*, vol. 52, no. 3, pp. 1059–1069, 2010.

[8] 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 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09, Washington, DC, USA, 2009, pp. 1–8.

[9] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[10] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for streaming betweenness centrality," in *Proceedings of the 2012 ASE/IEEE International Conference on Social Computing and 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust*, ser. SOCIALCOM-PASSAT '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 11–20.

[11] U. Brandes and C. Pich, "Centrality estimation in large networks," in *Intl. Journal of bifurcation and chaos, special issue on complex networks' structure and dynamics*, 2007.

[12] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, "Betweenness centrality on gpus and heterogeneous architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, ser. GPGPU-6. New York, NY, USA: ACM, 2013, pp. 76–85.

[13] 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.

[14] M. Harris, "Optimizing Parallel Reduction in CUDA," http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf, nVidia, Tech. Rep., 2008.

[15] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proc.*, ser. Contemporary Mathematics, vol. 588. American Mathematical Society, 2013.

[16] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung, "Qube: A quick algorithm for updating betweenness centrality," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 351–360.

[17] M. Kas, M. Wachs, K. M. Carley, and L. R. Carley, "Incremental algorithm for updating betweenness centrality in dynamically growing networks," in *Proceedings of the 5th International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '13.

[18] 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.

[19] 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.

[20] L. Li, D. Alderson, J. C. Doyle, and W. Willinger, "Towards a theory of scale-free graphs: Definition, properties, and implications," *Internet Mathematics*, vol. 2, no. 4, pp. 431–523, 2005.

[21] D. Watts and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, no. 393, pp. 440–442, 1998.

[22] D. A. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," 2006.

[23] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *The IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, Sep. 2012, best paper award.