A Shared-Memory Algorithm for Updating Single-Source Shortest Paths in Large Weighted Dynamic Networks

Sriram Srinivasan
University of Nebraska Omaha, Omaha, NE 68106, USA
Email: sriramsrinivas@unomaha.edu

Sara Riazi
University of Oregon, Eugene, OR 97403
Email: riazi@uoregon.edu

Boyana Norris
University of Oregon, Eugene, Eugene,OR 97403
Email: bnorris2@uoregon.edu

Sajal K. Das
Missouri University of Science and Technology, Rolla, Rolla,MO 65409,USA
Email: sdas@mst.edu

Sanjukta Bhowmick
University of North Texas, Denton,Texas,USA
Email: sanjukta.bhowmick@unt.edu

***Abstract*—Computing the single-source shortest path (SSSP) is one of the fundamental graph algorithms, and is used in many applications. Here, we focus on computing SSSP on large dynamic graphs, i.e. graphs whose structure evolves with time. We posit that instead of recomputing the SSSP for each set of changes on the dynamic graphs, it is more efficient to update the results based only on the region of change. To this end, we present a novel two step shared-memory algorithm for updating SSSP on weighted large-scale graphs. The key idea of our algorithm is to identify changes, such as vertex/edge addition and deletion, that affect the shortest path computations and update only the parts of the graphs affected by the change.**

We provide the proof of correctness of our proposed algorithm. Our experiments on real and synthetic networks demonstrate that our algorithm is as much as 4X faster compared to computing SSSP with Galois, a state-of-the-art parallel graph analysis software for shared memory architectures. We also demonstrate how increasing the asynchrony can lead to even faster updates. To the best of our knowledge, this is one of the first practical parallel algorithms for updating networks on shared-memory systems, that is also scalable to large networks.

***Keywords*-dynamic networks, parallel graph algorithms, single-source shortest path**

I. Introduction

Computing single-source shortest paths (SSSP), that is, the distance of all vertices from a given source , is one of the fundamental problems of graph theory. This problem has applications in many disciplines from Internet routing, to finding routes in GPS systems, to path planning for robots and computing centrality values that indicate the importance of a vertex in social and biological networks.

The structure of graphs that occur in real world systems often changes with time, and therefore the results of SSSP also change accordingly. Depending on the type of change, it is often faster to update the existing results rather than recompute the entire problem on the changed graph [1]. Since many big data problems require the analysis of extremely large graphs, e.g. of over a million vertices, it is imperative to use parallel algorithms for analyzing these graphs.

While there exists *sequential* algorithms for updating SSSP and *parallel* algorithms for recomputing SSSP from scratch on the changed graph, there are almost no parallel algorithms that can also update SSSP without recomputation. To the best of our knowledge, ours is the first paralllel algorithm for updating SSSP on shared memory systems. The few existing SSSP updating algorithms for clusters based systems and GPUs, do not scale to large graphs or are very memory intensive (Section VI provides more details).

Key Contributions. Our main contribution is to present *a shared-memory parallel algorithm for updating SSSP on massive weighted dynamic networks.* Our key strategy is to recognize that a change (addition/deletion of a vertex/edge) only affects certain parts of the graph. Therefore if we can limit the new set of computations to only the subgraph affected by the set of changes, then we can significantly reduce the updating time.

Our algorithm consists of a novel two step process of **first**, identifying the initial set of vertices affected by the changes and **then** updating the SSSP tree by setting each of the affected vertices as the root. These steps are implemented in parallel, and breaking the operation into two steps, improves the scalability while reducing redundant computations. Our main contributions are as follows:

- Present a two-step parallel algorithm for updating SSSP on large-scale dynamic weighted graphs on shared memory architectures. (Section III).
- Provide proofs of the correctness of our algorithm. Discuss how increasing the level of asynchrony can improve the execution time (Section IV).
- Empirically demonstrate that our algorithm is faster than recomputing SSSP using Galois, a state-of-the-art shared memory graph analysis software [1] (Section V).

[1]In this paper we use the terms network and graph synonymously.

II. Background: Updating SSSP Sequentially

In this section we present the foundational concepts of this paper including a brief description of relevant graph terminology, the Dijkstra's methods for computing SSSP and sequential algorithm for updating the SSSP.

A. Graph Terminology

A graph is defined by $G = (V, E)$, where V is the set of vertices and E is the set of edges. In weighted graphs, each edge is associated with a real number known as its weight. The vertices u and v are called the *endpoints* of an edge $e = (u, v)$. A *path* of length l is an alternating sequence $v_0, e_1, v_1, e_2, \ldots, e_l, v_l$ of vertices and edges, such that for $j = 1, \ldots, l$, the vertices v_{j-1} and v_j are the endpoints of edge e_j, with no edges or vertices repeated. A *cycle* is a path with the same starting and ending vertices, i.e. v_0=v_l. A *tree* is a connected graph with no cycles. A *spanning tree* is a tree that includes all vertices in the graph.

The *shortest path* between two vertices, u and v, is a path that starts from u and ends at v such that the sum of the edges is minimized. The *single-source shortest path tree* (SSSP tree), T, is a spanning graph, such that the distance of any vertex v (referred here as $Dist$) from the root node r in T is the shortest path from v to r in the original graph G. For any two vertices v and u in the SSSP tree, v is marked as the parent (in this paper referred by the variable $Parent$) of u, if *(i)* the distance of v from the root is less than the distance of u from the root, and *(ii)* there exists an edge (v, u) in T connecting v to u. Figure 1 (a) and (b) gives an example of a graph and its corresponding SSSP tree.

B. Computing the Single-Source Shortest Paths

Dijkstra's method is most commonly used for computing the single-source shortest path. When implemented using a priority queue, the complexity of the algorithm is $O(|E| + |V|log|V|)$. Algorithm 1 gives the pseudocode of Dijkstra's algorithm on a weighted undirected graph. This version uses priority queue and maintains the structure of the SSSP tree in the form of a parent-child relation.

C. Updating the Single-Source Shortest Paths

The changes in the graph are in the form of addition/deletion of vertices/edges. The process for updating the SSSP tree for a single changed edge is as follows. Identify the endpoint vertex that is affected by the change, treat it as a root and push it to the priority queue. Then continue to update the SSSP tree, as per the Dijkstra's method, until the priority queue is empty (see Algorithm 2).

The information about the parent-child relation among the vertices in the SSSP tree is not necessary for updating due to edge insertion, since all the neighboring vertices of v are checked to see if their distance decreases. However, when an edge is deleted, we need to know which are its children as per the tree, since we have to disconnect the children and reconnect using neighbors who are not children. In this case, the structure of the SSSP tree is needed.

Algorithm 1 Dijkstra's Algorithm for Finding SSSP

Input: Weighted Graph $G(V, E)$, Source Vertex s. Weight of edge (v, u) is $W(v, u)$.

Output: Distance from all vertices v to s. The SSSP Tree is stored in the form of Parent of each vertex

```
1:  procedure DIJKSTRA'S METHOD(G, s)
        ▷ Initialize Distance , Dist, Parent Parent and Priority Queue, PQ.
2:      for v ∈ V do
3:          Dist[v] = INF
4:          Parent[v] = v
5:      PQ ← s
6:      Dist[s] ← 0
7:      while (PQ not empty) do
8:          v=PQ.top()
9:          PQ.dequeue()
10:         for n where n is neighbor of v do
11:             if Dist[n] > Dist[v]+W(v,n) then
                                              ▷ Update Dist
12:                 Dist[n] ← Dist[v] + W(v,n)
13:                 PQ.enqueue(n)     ▷ Add n to Priority Queue
14:                 Parent[n] ← v ▷ Mark v as parent of n
```

Batched Updates: The addition of a vertex is equivalent to adding a new vertex, and then inserting edges. Similarly vertex deletion can be accomplished by deleting the edges connecting it to the graph. Moreover, multiple edges can be added/deleted. Thus the changes to a graph occur in batches, i.e., there is a set of changed edges. As per the algorithm in [2], the distances of the affected endpoints of all the changed edges are updated, and these vertices are added to the priority queue. Then their neighbors are processed in a manner similar to that given in Algorithm 2. This is in effect processing the changed edges sequentially based on the position of the affected vertices in the priority queue.

III. Our Contribution: Parallel Dynamic SSSP

We present our proposed shared memory algorithm for updating SSSP on a graph. Since change in vertices, in effect, equals to addition/deletion of edges, we use changed edges in our examples for simplicity.

A. Challenges to Scalability

There are two ways that a batch of changed edges can be processed in parallel. The **first** is to distribute the work across the graph. Here each thread is assigned a subgraph. Each subgraph is updated in parallel, as per the set of changes that affect that subgraph. This approach is prone to load imbalance, since the set of changed edges may not

Algorithm 2 Updating SSSP for a Single Change

Input: Weighted Graph $G(V, E)$, $Dist$, $Parent$, SSSP Tree T, Changed edge $E = (a, b)$ with weight $W(v, u)$.

Output: Updated $Dist_u$, $Parent_u$ and SSSP Tree T_u based on structure of $Parent_u$

1: **procedure** UPDATING_PER_CHANGE(E, G, T, $Dist$, $Parent$)
2: ▷ Initialize Updated Distance and Parent
3: **for** $v \in V$ **do**
4: $Dist_u[v] \leftarrow Dist[v]$
5: $Parent_u[v] \leftarrow Parent[v]$
▷ Find the affected vertex, x
6: **if** $Dist_u[a] > Dist_u[b]$ **then** $x \leftarrow a$, $y \leftarrow b$
7: **else** $x \leftarrow b$, $y \leftarrow a$
▷ Initialize Priority Queue PQ and update $Dist_u[x]$
8: $PQ \leftarrow x$
9: **if** E is inserted **then** $Dist_u[x] = Dist_u[y] + W(a, b)$
10: **if** E is deleted **then** $Dist_u[x] = INF$
▷ Update the subgraph affected by x
11: **while** (PQ not empty) **do**
12: v=$PQ.top()$
13: $PQ.dequeue()$
14: Process_Vertex(v,G, T, $Dist_u$, $Parent_u$)
15: **procedure** PROCESS_VERTEX(v, G, T, $Dist_u$, $Parent_u$)
16: **for** n where n is neighbor of v in G **do**
17: **if** $Dist_u[v] = INF$ AND $Parent_u[n] = v$ **then**
18: $Dist_u[n] = INF$
19: $PQ.enqueue(n)$
20: **else**
21: **if** $Dist_u[n] > Dist_u[v]$+$W(v, n)$ **then**
22: $Dist_u[n] \leftarrow Dist_u[v] + W(v, n)$
23: $PQ.enqueue(n)$
24: $Parent_u[n] \leftarrow v$
25: **else**
26: **if** $Dist_u[v] > Dist_u[n]$+$W(v, n)$ **then**
27: $Dist_u[v] \leftarrow Dist_u[n] + W(v, n)$
28: $PQ.enqueue(v)$
29: $Parent_u[v] \leftarrow n$

be equally distributed across the graph. Thus, situations can occur where one subgraph processes a lot of changed edges, while another subgraph may not have to process any edges.

The **second** approach, which we will use, is to distribute the work across the set of changed edges. That is, for each changed edge we execute the corresponding Algorithm 2 in parallel. While this is a more scalable approach, there are also several challenges with this strategy as follows.

- *Load Imbalance:* The computation required for a changed edge is proportional to the size of the subgraph affected due to the change. The size can range from 0(insertions or deletions do not affect the SSSP tree) to $|V|$.
- *Locking to Avoid Race Conditions:* The subgraphs affected by two different changed edges can overlap, i.e., include a common set of vertices and edges. To avoid race conditions, updates on overlapping regions have to be executed using locking, a process that significantly reduces scalability.
- *Redundant Computations:* The occurrence of overlapping subgraphs can lead to redundant computations, particularly if the changed edges occur in the same path to the root. Using priority queues can reduce the redundancy, however, priority queues are not amenable to parallelization.

B. Two-Step Algorithm for Parallel Edge Updates

To address these issues we propose the following two-step algorithm for updating the set of changed edges in parallel. Figure 1 gives an example of how our algorithm works.

Inputs and Outputs. The *input* to our algorithm is the set of changed edges and the graph G, whose edges have been partitioned into *key edges* that are in the SSSP and *remainder edges* that are not in the tree. Note that we take the SSSP as an input, because our algorithm focuses on updating an existing SSSP, not re-computing it. The vertices in the SSSP, contain information about the their parent node ($Parent$) and their distance ($Dist$) from the root.

The *output* is the updated SSSP tree with the changed edges incorporated as appropriate. The changed edges that are not in the SSSP tree are added to the remainder edges.

Step 1. Identify Changed Edges that Affect the SSSP. (Algorithm 3): In this step, we process all the changed edges in parallel, and identify the ones that affect the SSSP. Assuming all edge weights are positive, given an edge (u, v), let the distance of u in the SSSP tree be lower than that of v. Figure 1 provides an example.

Let the edge (u, v) be marked for insertion. The edge will affect the SSSP only if by including the edge (u, v), the distance of v is reduced, e.g. edge (A-F) in Figure 1. If this happens then u is marked as the parent of v, and the distance of v is updated. (Algorithm 3 Lines 10-15).

Now consider the edge (u, v) marked to be deleted. If the edge is part of the key edges in the SSSP tree (e.g. edge (A-B) in Figure 1), then the distance of v is changed to INF (a very high value) to mark that the edge connecting v to the tree is deleted. In both cases, insertion and deletion, the vertex v is marked as affected. (Algorithm 3 Lines 16-21).

For the edges marked to be inserted, we conduct a further iteration on them such that if a vertex has multiple changed edges associated with it, it is updated with the edge with the lowest weight (Algorithm 3 Lines 22-34).

Step 2. Update The Subgraphs of the Affected Vertices(Algorithm 4): Once the changed edges that affect the SSSP are identified, we update the subgraphs leading from the affected vertices. Each vertex is associated with a

boolean variable, $affect$, that is set to true if the distance of that vertex has changed. At each iteration, the function $Process\_Vertex\_Parallel$ is applied to all the affected vertices in parallel.The iterations converge when there are no vertices to be updated.

In the example in Figure 1 at the first iteration, the affected nodes are B and F. The distance of B is set to INF, so it also sets the distance of its neighbors D and C to INF. However, since C is also a neighbor of F, C becomes the child of F and thus obtains a lower distance. At iteration 2, C and D are the affected nodes. D updates the distance of E to INF. C updates the distance of B, D, and E. Thus E obtains a lower value than INF. At iteration 3, B, D and E are the affected nodes. Only E updates the distance of D. At the final iteration, D is the affected vertex. However no updates occur, so the iterations converge.

$Process\_Vertex\_Parallel$ is similar to Algorithm 2. Here, instead of pushing the vertices to the priority queue, which requires them to be processed sequentially, we mark them as affected, so that they can be processed concurrently.

C. Addressing the Scalability Challenges

Our algorithm addresses the challenges to scalability described in Section III-A as follows.

Load Balancing: For step 1, in Lines 9-21, Algorithm 3, each changed edge has equal computation load. Their end points are compared to see whether the edge can be inserted or deleted. Thus regardless of whether the edge affects the SSSP tree, the computation load is equal and balanced. Similarly, each item in the for loop, Lines 25-34, Algorithm 3, has equal computation cost. The number of items to process decreases over each iteration of the while loop.

For step 2, the operations over the parallel for loop Line 4, Algorithm 4 can be slightly more imbalanced. Since we are synchronizing at each level, the computation load per affected vertex is proportional to the number of neighbors. Thus the load balance depends on the skewness of the degree distribution of the network. In this case, a dynamic scheduler, such as the one provided by OpenMP can be used to obtain an equitable load balance.

Locking to Avoid Race Condition: We eliminate the need for locking by using iterative updates. The distance of vertices in the subgraphs that are affected by multiple changed edges gets updated over multiple iterations.

If the distance of the vertex is set to a higher value in one iteration, in a subsequent iteration, the edge that provides the optimal distance will update it. Note that since the distance values are updated only if they become lower, after a certain iteration the vertex will reach the optimal value and its distance will no longer be updated, (and thus the vertex will not be marked as affected). This will occur for all vertices, hence, ultimately no vertex will be marked as affected and the iterations will converge.

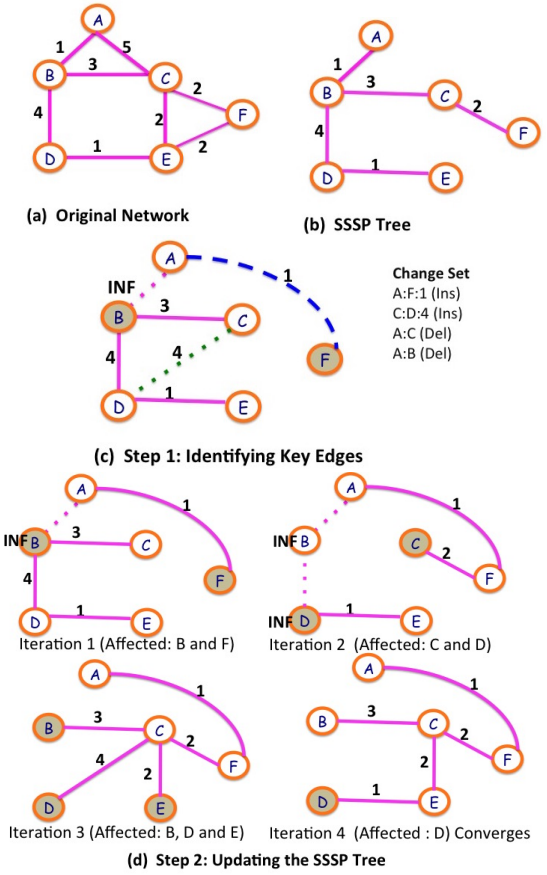

Figure 1. Example of Updating SSSP Tree for a Set of Changes (colors online). Figure (a): The original graph. Figure (b): The SSSP tree from the original graph. Figure (c): Step 1 of the updating algorithm. Each changed edge can be processed in parallel. Key Edge (A-F) that affects the SSSP is given in blue dashed line. Vertex A is now the parent of F instead of C. Edge C-D that does not affect the SSSP is given in green dotted line. Key deleted edge A-B is shown in pink dotted line and the distance of B is set to INF. Figure (d): Iterations in Step 2 that lead to the updated SSSP. The active edges at each iteration are shaded and processed in parallel. At each iteration, the solid edges form a tree (or forest).

Redundant Computations: Redundant computations occur when values associated with the vertices are updated multiple times. As shown in Figure 1, the parent of vertex D is updated three times.

Synchronization at each level can reduce the number of redundant computations since, at the end of each synchronization, the affected vertices will have the lowest distance based on the changes propagated so far. Thus many non-optimal values are not transmitted to subsequent edges in the path and the number of redundant computations is reduced.

IV. Proof of Correctness and Asynchronous Updates

We now provide proofs of correctness of our algorithm, analyze the scalability, and discuss how increasing the level of asynchrony improves the execution time.

Algorithm 3 Step1: Processing Changed Edges in Parallel

Input: Graph $G(V, E)$, $Dist$, $Parent$, SSSP Tree T, Changed Edges CE. Weight of edge (v, u) is $W(v, u)$.

Output: Updated $Dist_u$, $Parent_u$ and SSSP Tree T_u based on structure of $Parent_u$

1: **procedure** UPDATING_BATCH_CHANGE(CE, G, T, $Dist$, $Parent$)
2: ▷ Initialize Updated Distance and Parent
3: **for** $v \in V$ **do in parallel**
4: $Dist_u[v] \leftarrow Dist[v]$
5: $Parent_u[v] \leftarrow Parent[v]$
6: $Affected[v] \leftarrow False$
▷ Find the affected vertex, x
7: **if** $Dist_u[a] > Dist_u[b]$ **then** $x \leftarrow a$, $y \leftarrow b$
8: **else** $x \leftarrow b$, $y \leftarrow a$
9: **for** Each edge $E(a, b) \in CE$ **do in parallel**
10: **if** E to be inserted **then**
▷ If E inserted, update $Dist$ and $Parent$
11: **if** $Dist_u[x] > Dist_u[y] + W(a, b)$ **then**
12: $Dist_u[x] = Dist_u[y] + W(a, b)$
13: $Parent_u[x] = y$
14: Mark E as inserted to SSSP Tree
15: $Affected[x] \leftarrow True$
16: **else** ▷ E to be deleted
17: Remove $E(a, b)$ from G
18: ▷ Check if Edge in SSSP tree
19: **if** $Parent[a] = b$ OR $Parent[b] = a$ **then**
20: $Dist_u[x] = INF$
21: $Affected[x] \leftarrow True$
▷ Lowest Value Edge gets Inserted
22: $Change \leftarrow True$
23: **while** $Change$ **do**
24: $Change \leftarrow False$
25: **for** Each edge $E(a, b) \in CE$ **do in parallel**
26: **if** E marked to be inserted to SSSP **then**
▷ Find the affected vertex, x
27: **if** $Dist_u[a] > Dist_u[b]$ **then** $x \leftarrow a$, $y \leftarrow b$
28: **else** $x \leftarrow b$, $y \leftarrow a$
29: ▷ Check replaces higher weighted edge
30: **if** $Dist_u[x] > Dist_u[y] + W(a, b)$ **then**
31: $Dist_u[x] = Dist_u[y] + W(a, b)$
32: $Parent_u[x] = b$
33: $Change \leftarrow True$
34: $Affected[x] \leftarrow True$

A. Correctness of the Algorithm

Consider a graph G, with positive edge weights, that is modified to graph G_u due to changes in its structure. We will prove that our parallel updating algorithm, as given by Algorithms 3 and 4, will produce a valid SSSP tree for G_u.

Lemma IV.1. *The parent-child relation between vertices assigned by our parallel updating algorithm produces tree(s).*

Proof: To prove this, we first create a directed graph where the parent vertices point to their children. Now consider a path in this graph between any two vertices a and b. The path goes from vertex a to vertex b. This means that a is an ancestor of b. As per the algorithm for creating the original SSSP tree (Algorithm 1), and the algorithms for parallel update (Algorithms 3 and 4), a vertex v is assigned as a parent of vertex u only if $Dist[v] < Dist[u]$, therefore transitively the distance of an ancestor vertex will be less than its descendants. Thus, $Dist[a] < Dist[b]$.

Since G has non-zero edge weights, it is not possible that $Dist[b] < Dist[a]$. Thus, there can be no path from b to a. Hence, all connected components are DAGs, and thus trees. ■

Lemma IV.2. *The tree obtained by our parallel algorithm will be a valid SSSP tree for G_u.*

Proof: Let T_u be a known SSSP tree of G_u and let T_{alg} be the tree obtained by our algorithm. If T_{alg} is not a valid SSSP tree, then there should be at least one vertex a for which the distance of a from the source in T_{alg} is greater than the distance of a from the source vertex in T_u.

Consider a subset of vertices, S, such that all vertices in S have the same distance in T_u and T_{alg}. This means that $\forall v \in S$, $Dist_{T_u}[v] = Dist_{T_{alg}}[v]$. Clearly, such a set S can be trivially constructed by including only the source vertex.

Now consider a vertex a for which $Dist_{T_u}[v] < Dist_{T_{alg}}[v]$ and the parent of a is connected to a vertex in S. Let the parent of a in T_u (T_{alg}) be b (c).

Consider the case where b=c. We know that the $Dist_{T_u}[b] = Dist_{T_{alg}}[b]$. Also, per Algorithms 1, 3, 4, the distance of a child node is the distance of its parent plus the weight of the connecting edge. Therefore, $Dist_{T_{alg}}[a] = Dist_{T_{alg}}[b] + W(a, b) = Dist_{T_u}[b] + W(a, b) = Dist_{T_u}[a]$.

Now consider when $b \neq c$. Since the edge (b, a) exists in T_u, it also exists in G_u. Since $Dist_{T_u}[v] \neq Dist_{T_{alg}}[v]$, the distance of a was updated either in T_u or in T_{alg}, or in both, from the original SSSP tree. Any of these cases imply that a was part of an affected subgraph. Therefore, at some iteration(s) in Step 2, a was marked as an affected vertex.

Because the edge (b, a) exists in G and a is an affected vertex, in Step 2 (Algorithm 4, lines 13:21), the current distance of a would have been compared with $Dist_{T_{alg}}[b] + W(a, b)$. Since this is the lowest distance of a according to the known SSSP tree T_u, either the current distance would have been updated to the value of

$Dist_{T_{alg}}[b] + W(a,b)$ or its was already equal to the value. Therefore, $Dist_{T_u}[a] = Dist_{T_{alg}}[a]$. ■

Algorithm 4 Step 2: Updating Affected Vertices in Parallel

Input: Weighted Graph $G(V,E)$, $Dist$, $Parent$, $Affected$, SSSP Tree T. Weight of edge (u,v) is $W(u,v)$.

Output: Updated $Dist_u$, $Parent_u$ and SSSP Tree T_u based on structure of $Parent_u$

1: **procedure** PROCESS_VERTEX_PARALLEL(v, G, T, $Dist_u$, $Parent_u$)
2: $Change \leftarrow True$
3: **while** $Change$ **do** $Change \leftarrow False$
4: **for** $v \in V$ **do in parallel**
5: **if** $Affected[v] = False$ **then** Skip the vertex
6: $Affected[v] \leftarrow False$
7: **for** n where n is neighbor of v in G **do**
8: **if** $Dist_u[v]$=INF & $Parent_u[n] = u$ **then** $Dist_u[n] = INF$
9: $Affected[n] \leftarrow True$
10: **else**
11: **if** $Dist_u[n] > Dist_u[v]$+$W(v,n)$ **then** $Dist_u[n] \leftarrow Dist_u[v] + W(v,n)$
12: $Affected[n] \leftarrow True$
13: $Parent_u[n] \leftarrow v$
14: **else**
15: **if** $Dist_u[v] > Dist_u[n]$+$W(v,n)$ **then**
16: $Dist_u[v] \leftarrow Dist_u[n] + W(v,n)$
17: $Affected[v] \leftarrow True$
18: $Parent_u[v] \leftarrow n$

B. Scalability of the Algorithm

Assume that we have p threads, and m changed edges to process, out of which there are a insertions and b deletions that affect the SSSP.

For Step 1, each changed edge can be processed in parallel, requiring time $O(m/p)$. For the edges that are marked to be inserted, we perform further iterations such that, for vertices associated with multiple changed edges, the edge with the lowest weight is selected. The number of iterations is generally low (i.e. 1-5), and can be treated as a constant. Each of these edges can also be processed in parallel requiring time $O(a/p)$.

For Step 2, the parallelism depends on the number of affected vertices, and the size of the subgraph that they alter. At each iteration, an affected vertex goes through its neighbors, so the work is proportional to its degree.

Assuming that x vertices are affected, $x \leq m$, and the average degree is a vertex is d, then the time per iterations is $O(xd/p)$. The maximum number of iteration required would be the diameter of the graph D. Thus an upper bound time complexity for Step 2 is $D * O(xd/p)$.

Taken together, the complexity of the complete updating algorithm is $O(\frac{m+a}{p})$+$O(\frac{Dxd}{p})$. The term Dxd represents the number of edges processed. Let this be equal to E_x. The number of edges processed in the recomputing method would be $E + m$. Therefore for the updating method to be effective, we require $E_x < (E + m)$.

C. Reducing Iterations Through Asynchronous Updates

We now consider how we can reduce the value of E_x. One way to do this is to reduce the number of iterations. This can be done by increasing the level of asynchrony, i.e. reducing the frequency of synchronization at the for loop.

To increase the asynchrony, we process for longer paths, rather than only the neighbors. While synchronizing the updates after each set of neighbors are processed can improve load balancing and reduce the redundant computations, increased asynchrony can reduce the number of iterations and consequently reduce the execution time, by reducing the number of times the parallel for loop is invoked. In Section IV-C, we empirically demonstrate how increasing the rate of synchronization affects the execution time and redundancy of computations.

V. Experimental Results

A. Datasets and Experimental Environment

In this section we present our experimental results. We use several synthetic and real world graphs for evaluating the performance of our algorithms. For experiments we used a machine with 36-core dual-socket Intel Xeon E5-2699 v3 2.30 GHz (Haswell) CPUs and 256GB DDR4 RAM.

Table I
Real-World of Graphs in Our Test Suite.

Name	Num. of Vertices	Num. of Edges
com-Live Journal	3,997,962	34,681,189
com-Youtube	1,134,890,	2,987,624
soc-Pokec	1,632,803	30,622,564

Synthetic Graphs: We used the R-MAT model based on recursive Kronecker matrices to generate the synthetic graphs. The degree distribution of the graph is defined by four values (a,b,c,d) whose sum adds up to 1.

We generated two types of RMAT graphs. The first ($a = .45, b = .15, c = .15, d = .25$), labeled **G**, has a scale-free degree distribution. The second ($a = b = c = d = .25$), labeled **ER**, is a random network with normal degree distribution. Comparing between these graphs help us to study how degree distribution affects the performance. Graphs RMAT24 and RMAT25 have about 16M vertices, 268M edges and 33M vertices, 536M edges respectively.

Real-world Graphs: Our test suite contains three real-world networks from the Stanford Network Database [3] You Tube, Pokec, and Live Journal (Table I lists their sizes).

B. Scalability Results

Figures 2 and 3 show the parallel scaling of our algorithm on two synthetic and three real-world graphs, respectively (the graphs are described in Table I). We show the time for updating the shortest path following a 50-million-edge update with different fractions of edge insertions and deletions. Specifically, $x\%$ insertions mean that the dataset contains $x\%$ edge insertions and $(100 - x)\%$ edge deletions. We compare time instead of TEPS(traversed edges per second) because (i) time is a more fundamental metric and (ii) TEPS is not a suitable metric for dynamic networks since our goal is to process as few edges as possible.

The results show that our algorithm is scalable for all the graphs, although the scalability decreases as the number of processors increase (as the amount of work per thread shrinks). We observe that random(ER) graphs are more scalable than the scale free(G) graphs.

For the real-world graphs, the time increases for processing deletions, whereas there is not a significant change in time for the synthetic graphs. We presume that this is because the real-world graphs are smaller, and therefore the percentage of affected vertices due to deletion is higher.

1) Effect of Choice of Source Vertex: For consistency, we use vertex 0 as the source vertex in all cases in Figures 2 and 3. On the other hand, Figure 4 explores the effect of choosing a different source vertex. We selected two other sources vertices at random from two of the graphs and computed the time for updating over 100% insertions. As shown in Figure 4, the timings for the three vertices were almost identical with very low standard deviation between them. This result indicates that the selection of source vertices does not significantly affect the scalability or the execution time.

2) Addition and Deletion of Vertices: Our algorithm also supports the addition and deletion of vertices. To accommodate this, our graph data structure, which is an adjacency list, maintains a buffer space for possible addition of new vertices. In effect, this is equivalent to storing several disconnected vertices. Once a new vertex is added, the edges are inserted in the corresponding row of the adjacency list as per our template. To delete a vertex, all its edges are marked to be deleted. Figure 5 shows the strong scaling results for adding 100 vertices and deleting 50 vertices from RMAT24, ER and G.As in the case of only edge changes, the scalability is better for the random graphs, than the scale free graphs.

C. Comparison with Galois

We compare the time taken to update the SSSP with the time taken to recompute the shortest path from scratch by using the latest stable release, 2.2.1, of the Galois software [4]. Galois identifies fine-grained parallelism in graph algorithms and has been shown to be very fast compared to other parallel network packages. Galois computes the solution to the SSSP problem with non-negative edge weights using an iterative Δ-stepping algorithm [5], which is a SSSP algorithm whose schedule can be adjusted to fall between Dijkstras and the Chaotic Relaxation algorithms (we used the default Δ value).

Figure 6 shows the execution time of the update algorithm compared with the Galois static SSSP implementation on the RMAT24-G and RMAT24-ER graphs (described in Sec V-A). Each experiment was repeated four times. The "SSSP Total" time is the cumulative time for one update of one million edges, with 100% (left) and 75% (right) insertions. The Galois time is for computing SSSP on the full updated graph.

Table II summarizes the improvement over Galois for different thread counts and two RMAT graphs. The improvement was computed by averaging the times from four experiments for each parameter combination and dividing the Galois average time by the SSSP Update algorithm average time. In all cases, the update algorithm is faster than the recomputation.

Table II
IMPROVEMENT OF OUR UPDATE ALGORITHM OVER GALOIS'S STATIC ALGORITHM FOR A 1,000,000-EDGE UPDATE WITH 100% AND 75% INSERTIONS ON THE RMAT24-ER AND RMAT24-G GRAPHS.

Experiment	Threads	Galois	SSSP Update	Improv.
RMAT24_ER_100i	1	53.5	25.5	2.1
RMAT24_ER_100i	2	27.2	16.5	1.6
RMAT24_ER_100i	4	14.4	8.8	1.6
RMAT24_ER_100i	8	7.8	4.9	1.6
RMAT24_ER_100i	16	4.6	3.1	1.5
RMAT24_ER_100i	32	3.0	2.5	1.2
RMAT24_ER_100i	48	3.0	2.4	1.2
RMAT24_ER_100i	64	2.8	2.4	1.2
RMAT24_ER_100i	72	2.9	2.5	1.1
RMAT24_G_100i	1	69.4	17.5	4.0
RMAT24_G_100i	2	36.5	10.6	3.4
RMAT24_G_100i	4	17.9	5.7	3.1
RMAT24_G_100i	8	10.1	3.3	3.1
RMAT24_G_100i	16	5.8	2.1	2.8
RMAT24_G_100i	32	3.7	2.0	1.9
RMAT24_G_100i	48	3.5	1.8	1.9
RMAT24_G_100i	64	3.4	2.0	1.7
RMAT24_G_100i	72	3.3	2.0	1.6

D. Effect of Increasing Asynchrony

We now explore how increasing the level of asynchrony effects the execution time. We define the level of asynchrony as the one minus the length of the path to be traversed before synchronization at the for loop. Level 0 means that only the neighbors are processed, Level 1 means that distance-2 neighbors are processed and so on. We updated RMAT24 G and ER graphs with 10M edges with 100% and 75%

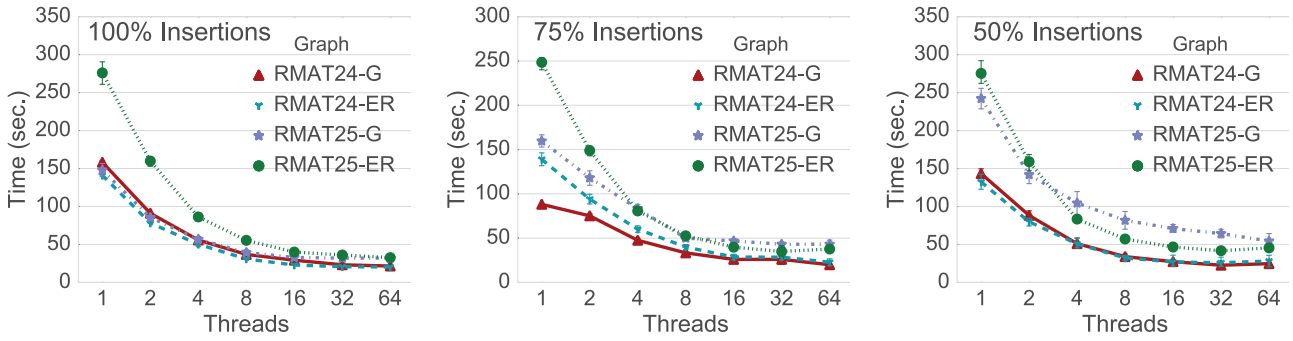

Figure 2. Scalability of shared-memory parallel SSSP computation with two RMAT (synthetic) graphs (described in Table I) for 50 million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

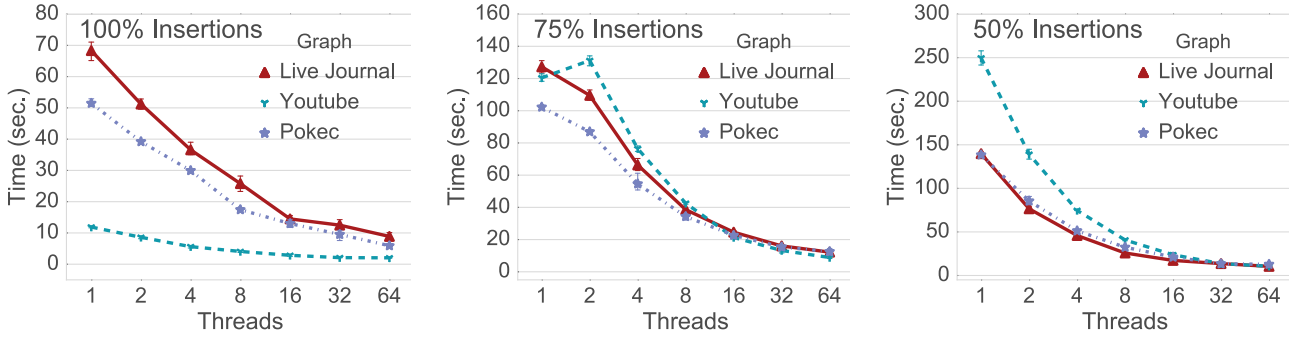

Figure 3. Scalability of shared-memory parallel SSSP computation for three real-world graphs (described in Table I), for 50 Million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

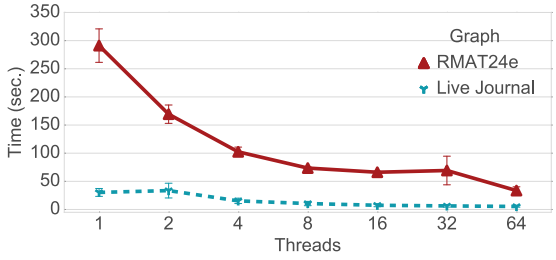

Figure 4. Comparison of the execution time and scalability for updating SSSP based on different source vertices with 100% insertion (0% deletion). For both the graphs, the choice of the source vertex has very little effect on the time and scalability.

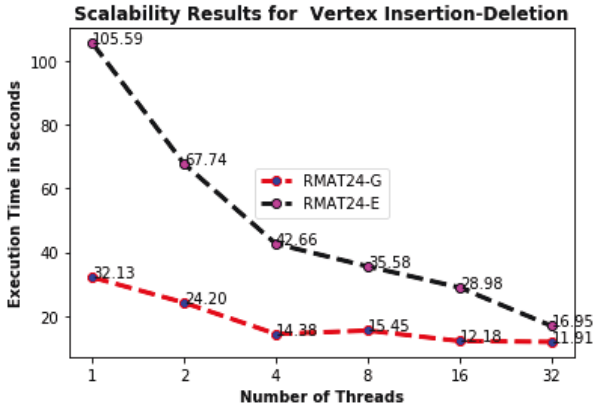

Figure 5. Scalability of adding and deleting vertices. RMAT24-ER,G networks with 10 million edges changed, 100 vertices inserted and 50 vertices deleted.

insertions. For each of these four cases, we varied the level of asynchrony by 0, 50, 100, 500 1K and 5K.

Figure 7 gives the execution time with increasing asynchrony for 8 threads. There is a sharp decrease in time as the level of asynchrony increases from 0 to 50. The execution time decreases with further increase in asynchrony, although the slope is not as steep.

We now study the effect of asynchrony on vertex updates. If a vertex is updated multiple times then each update is counted. Thus the updates include both the necessary as well as redundant computations. Figure 8 compares how the percentage of updates, computed as $\frac{updates\_at\_level\_r - updates\_at\_level\_0}{updates\_at\_level\_0}$, change with increasing

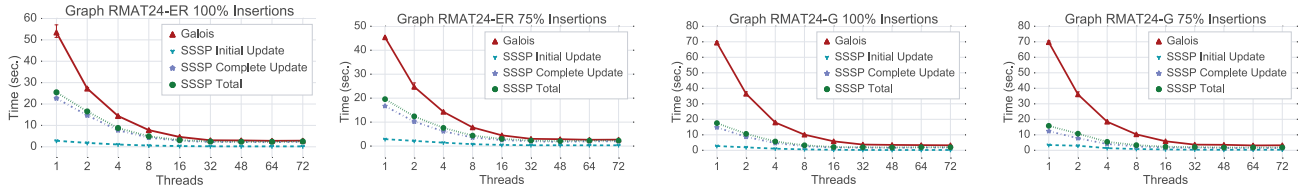

Figure 6. Comparison of our algorithm with Galois for two different RMAT graphs of degree 24 with 1,000,000 edge changes consisting of 100% insertions and 0% deletions (first and third plots), and 75% insertions and 25% deletions (second and fourth plots). The update phases of our algorithm, Initial Update (Step 1) and Complete Update (Step 2) are shown individually, and their sum is reflected in the SSSP Total value reflecting the total update cost. The Galois measurement is for applying the Δ-stepping algorithm over the modified graph with the default Δ value.

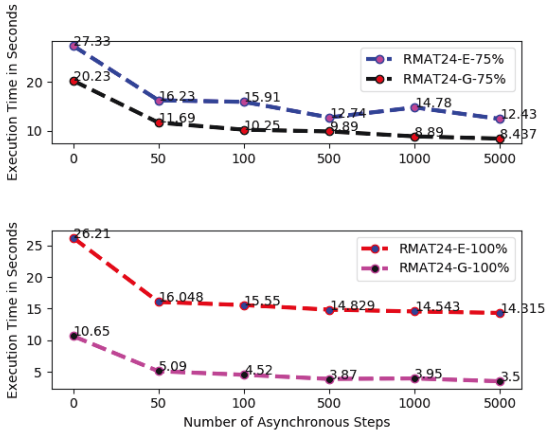

Figure 7. Change in execution time with increased levels of asynchrony on 8 threads. X-axis: level of asynchrony. Y-axis: time to update SSSP. Higher asynchronous levels lead to lower time.

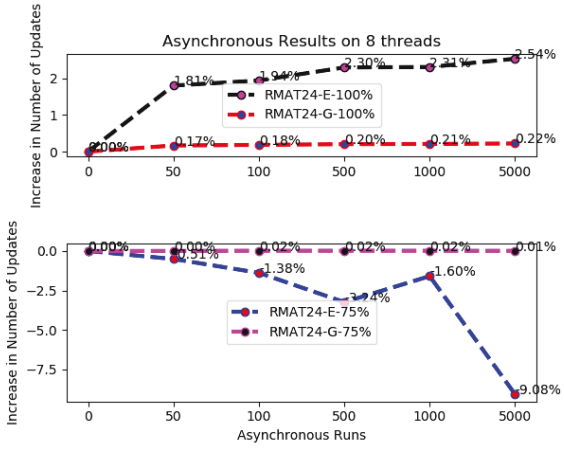

Figure 8. Percentage change in number of updates of vertex values with respect to updates for the synchronous case (set at 0) on 8 threads. X-axis: level of asynchrony. Y-axis: percentage change in updates.Asynchrony, in general, leads to more updates of the vertices.

level of asynchrony, with respect to the updates for the synchronous (level 0) method on 8 threads. Except for RMAT24E 75% update, for all other cases, the number of updates increase, indicating increase of redundant computations. Conversely the time is lower. We hypothesize that this is due to the reduced calls for synchronization.

Figure 9 compares the scalability of the synchronous update to asynchrony of 50k. Asynchrony at level 50K requires less time, but is as scalable as the synchronous update. These results demonstrate that correctly tuning the level asynchrony can lead to faster and scalable update of the SSSP tree.

VI. Related Work

We discuss related work on static computation and dynamic updates for SSSP and centrality computations.

1) Algorithms for Computing SSSP: Many parallel implementations of Dijkstra's algorithm exist, for example Galois [1], which we use for our comparison. A distributed memory framework, Havoqgt [6], also supports parallel SSSP. Maleki *et. al.* [7] proposed Dijskstra Strip Mined Relaxation algorithm (DSMR) for shared and distributed memory systems. Dijkstra's algorithm has been parallelized using transactional memory and helper threads [8].

2) Algorithms for Dynamic Updates: Ramalingam *et. al.* [2], presented an SSSP algorithm for a dynamic network. Narvez *et. al.* [9] proposed algorithms to dynamically update shortest path trees. Bauer *et al.* [10] proposed a batch-dynamic SSSP algorithm. They also conducted an empirical study of different dynamic shortest path algorithms for batch updates. All these algorithms are sequential. Vora *et. al.* [11] have proposed approximations for streaming graphs.

We know of only one parallel dynamic algorithm for updating SSSP [12]. This algorithm is implemented on GPUs using JavaScript. However, the dataset did not contain very large graphs and no scalability results were provided.

3) Computing and Updating Centrality Metrics: SSSP is the base algorithm for computing vertex centralities for which many approximate algorithms have been developed, including parallel implementations on different platforms [13]–[15], algorithms for dynamic networks [16]–[19]. However, the dynamic algorithms use approximate computations.

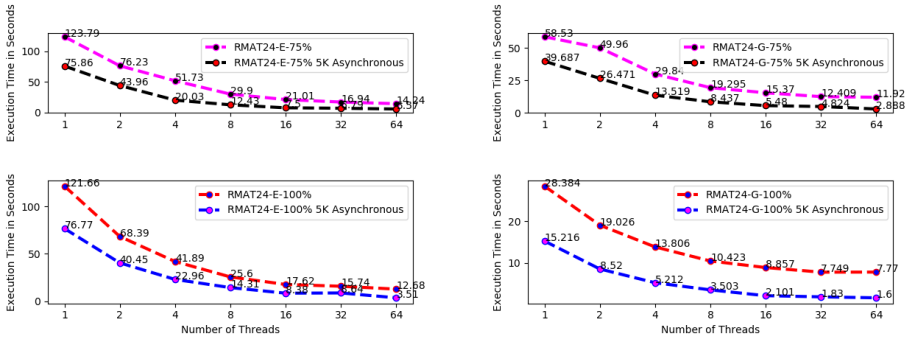

Figure 9. Comparison of scalability of synchronous and asynchronous (level 5000) updates. Asynchronous updates are faster and equally scalable.

VII. Conclusions and Future Work

In this paper we presented a shared-memory algorithm for updating the SSSP on dynamic graphs. We also demonstrated that increasing the level of asynchrony can further reduce the execution time. To the best of our knowledge, this is the first practical and scalable parallel algorithms for updating SSSP for shared memory systems [2]. In future work, we aim to extend our algorithm to provide exact updates of centralities on weighted graphs. We also plan to extend this idea on other parallel platforms, such as distributed clusters and GPUs.

Acknowledgment

Sanjukta Bhowmick and Sriram Srinivasan are supported by the NSF CCF Award #1533881 and #1725566. Sajal Das is supported by the NSF CCF Awards #1533918 and #1725755. Boyana Norris and Sara Riazi are supported by the NSF CCF Award #1725585.

References

[1] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of parallelism in algorithms," *SIGPLAN Not.*, vol. 46, pp. 12–25, June 2011.

[2] G. Ramalingam and T. Reps, "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, no. 1-2, pp. 233–277, 1996.

[3] J. Leskovec, "Stanford Large Network Dataset Collection." http://snap.stanford.edu/data/.

[4] "Galois system." http://iss.ices.utexas.edu/?p=projects/galois, 2018. Accessed on 2018-03-37.

[5] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, pp. 114–152, Oct. 2003.

[6] R. Pearce, M. Gokhale, and N. M. Amato, "Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates," pp. 549–559, IEEE, Nov. 2014.

[7] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarn, D. Padua, and K. Pingali, "DSMR: A parallel algorithm for single-source shortest path problem," pp. 1–14, ACM Press.

[8] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris, "Employing transactional memory and helper threads to speedup Dijkstra's algorithm," in *2009 International Conference on Parallel Processing*, pp. 388–395.

[9] P. Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng, "New dynamic algorithms for shortest path tree computation," vol. 8, no. 6, pp. 734–746.

[10] R. Bauer and D. Wagner, "Batch dynamic single-source shortest-path algorithms: An experimental study," in *Experimental Algorithms* (J. Vahrenhold, ed.), pp. 51–62, Springer Berlin Heidelberg.

[11] K. Vora, R. Gupta, and G. Xu, "KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations," pp. 237–251, ACM Press, 2017.

[12] A. Ingole and R. Nasre, "Dynamic shortest paths using javascript on gpus," 2015.

[13] K. Madduri and D. A. Bader, "Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis," in *IPDPS*, pp. 1–11, 2009.

[14] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *IPDPS*, pp. 1–8, 2009.

[15] 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*, GPGPU-6, (New York, NY, USA), pp. 76–85, ACM, 2013.

[16] K. Lerman, R. Ghosh, and J. H. Kang, "Centrality metric for dynamic networks," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, (New York, NY, USA), pp. 70–77, ACM, 2010.

[17] A. E. Sariyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Streamer: A distributed framework for incremental closeness centrality computation," in *CLUSTER*, pp. 1–8, 2013.

[18] E. J. Riedy and D. A. Bader, "Massive streaming data analytics: A graph-based approach," *ACM Crossroads*, vol. 19, no. 3, pp. 37–43, 2013.

[19] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 572–583, IEEE Press, 2014.

[2]Source code and instructions to reproduce our scalability results are available on https://github.com/DynamicSSSP/HIPC18/