A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks

Arindam Khanda, Sriram Srinivasan, Sanjukta Bhowmick, Boyana Norris, and Sajal K. Das, *Fellow, IEEE*

Abstract—The Single Source Shortest Path (SSSP) problem is a classic graph theory problem that arises frequently in various practical scenarios; hence, many parallel algorithms have been developed to solve it. However, these algorithms operate on static graphs, whereas many real-world problems are best modeled as dynamic networks, where the structure of the network changes with time. This gap between the dynamic graph modeling and the assumed static graph model in the conventional SSSP algorithms motivates this work. We present a novel parallel algorithmic framework for updating the SSSP in large-scale dynamic networks and implement it on the shared-memory and GPU platforms. The basic idea is to identify the portion of the network affected by the changes and update the information in a rooted tree data structure that stores the edges of the network that are most relevant to the analysis. Extensive experimental evaluations on real-world and synthetic networks demonstrate that our proposed parallel updating algorithm is scalable and, in most cases, requires significantly less execution time than the state-of-the-art recomputing-from-scratch algorithms.

Index Terms—Dynamic networks, single source shortest path (SSSP), shared-memory parallel algorithm, GPU implementation

1 Introduction

Networks (or graphs) are mathematical models of complex systems of interacting entities arising in diverse disciplines, e.g., bioinformatics, epidemic networks, social sciences, communication networks, cyber-physical systems, and cyber-security. The vertices of the network represent entities and the edges represent dyadic relations between pairs of entities. Network analysis involves computing structural properties, which in turn can provide insights to the characteristics of the underlying complex systems.

Many networks arising from real-world applications are extremely large (order of millions of vertices and order of billions of edges). They are also often dynamic, in that their structures change with time. Thus, fast analysis of network properties requires (i) efficient algorithms to quickly update these properties as the structure changes, and (ii) parallel implementations of these dynamic algorithms for scalability.

- *Arindam Khanda and Sajal K. Das are with the Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409 USA. E-mail: {akkcm, sdas}@mst.edu.*
- *Sriram Srinivasan is with the Department of Radiation Oncology, Virginia Commonwealth University, Richmond, VA 23284 USA. E-mail: sriram.srinivasan@vcuhealth.org.*
- *Sanjukta Bhowmick is with the Department of Computer Science and Engineering, University of North Texas, Denton, TX 76201 USA. E-mail: sanjukta.bhowmick@unt.edu.*
- *Boyana Norris is with the Department of Computer and Information Science, University of Oregon, Eugene, OR 97403 USA. E-mail: norris@cs.uoregon.edu.*

Manuscript received 25 Feb. 2021; revised 12 May 2021; accepted 18 May 2021. Date of publication 26 May 2021; date of current version 15 Oct. 2021. (Corresponding author: Boyana Norris.)
Recommended for acceptance by S. Alam, L. Curfman McInnes, K. Nakajima.
Digital Object Identifier no. 10.1109/TPDS.2021.3084096

We posit that the design of the parallel algorithm for updating the network properties should ideally be independent of the implementation platforms. However, most of the existing literature compound these two aspects by creating algorithms that are closely tied to the parallel platform. This lack of flexibility of applying algorithms across platforms becomes even more critical for *exascale computing*, which is generally achieved through a combination of multi-core, many-core machines as well as accelerators, such as GPUs. This motivated us to develop a *framework for updating Single Source Shortest Path (SSSP) in large dynamic networks* on GPUs and multicore CPUs, which is an important first step in achieving exascale capability.

The SSSP is a fundamental problem of network analysis that has applications in transportation networks [1], communication (wireline, wireless, sensor) [2], social networks [3], and many others. It is also the building block for computing other important network analysis properties such as closeness and betweenness centrality. While numerous algorithms can compute the SSSP in static networks and sequentially update the SSSP on dynamic networks [4], [5], [6], few parallel SSSP *update* algorithms have been created. Because most of the real-world networks are large, unstructured and sparse, developing efficient update algorithms becomes extremely challenging.

In this paper, we present a novel parallel algorithmic framework for updating the SSSP in large-scale dynamic networks. The key idea is to first identify the subgraphs that are affected by the changes, and then update only these subgraphs. The first step is trivially parallel. Each changed edge is processed in parallel to identify the affected subgraphs. The second step is challenging as it requires synchronization when the SSSP tree is altered. We propose an iterative method that converges to the smallest distance, thereby eliminating explicit and expensive synchronization constructs such as critical sections.

In our framework, these two steps can be designed and implemented independently of the parallel architecture platforms. We efficiently implement the proposed parallel SSSP update algorithm on both the GPU platform and the shared-memory platform (the latter extends our previous implementation [7] to handle changes in batches).

We experimentally demonstrate the efficacy of our parallel algorithm by comparing our GPU implementation with state-of-the-art GPU based recomputing-from-scratch SSSP implementation provided by Gunrock [8]. Experimental results exhibit that our implementation most often outperforms Gunrock, achieving up to *5.6x speedup*. We also compare our shared-memory implementation with state-of-the-art shared memory based recomputing-from-scratch SSSP algorithm implementation provided by Galois. Our shared memory implementation shows up to 5.X speed up when compared to Galois.

Our Contributions. The main contributions of this paper are as follows:

- We propose a common framework for efficiently updating SSSP that can be implemented for both CPU and GPU architectures. Our novel framework leverages a rooted tree based data structure to address challenges in load balancing and synchronization
- We implement our parallel algorithm on two platforms – NVIDIA GPUs and shared memory. The latter implementation significantly extends our previous version [9] by handling changes in batches, thus allowing to run up to 100M changes (i.e., 10 times more than in [9]).
- For the GPU implementation, we introduce a novel functional block based approach that breaks a complex function into several simple similar blocks and reduces the redundant computation.

2 Background

This section discusses preliminaries on SSSP and sequential algorithms for computing and updating SSSPs.

Graphs and SSSP. A graph is denoted as $G(V, E)$, where V represents the set of vertices (nodes) and E the set of edges. In this paper, we assume the graphs have non-negative edge weights.

A path between two vertices u and v is called a *shortest path*, if the sum of the weights of edges between these two vertices is the smallest. The Single Source Shortest Path (SSSP) problem is to compute shortest paths from a source vertex s to all other vertices in the graph. The output is a spanning tree, called the *SSSP tree*. We store the SSSP as a rooted tree by maintaining the distance from the root to every vertex and maintaining a parent-child relationship to capture the tree structure. If there exists an edge between u and v in the SSSP tree and the distance of u from the root is lower than that of v, then u is called the parent of vertex v.

Computing SSSP. The most well known sequential algorithms to compute SSSP is due to Dijkstra [10]. It creates the SSSP tree with the source vertex as its root. For a given graph $G(V, E)$, where all edge weights are non-negative, this algorithm maintains two sets of vertices, Set_1 and $Set_2 = V - \{Set_1\}$, where Set_1 contains all the vertices, whose shortest path weight from the source have already been calculated. The algorithm selects the vertex from Set_2 having the minimum shortest-path estimate, and adds it to Set_1 and relaxes all outgoing edges of the selected vertex. This is repeated until Set_2 becomes empty.

Relaxing an edge $e(u, v)$ involves checking if the estimated path distance of v from the source can be decreased by going through vertex u. The implementation of Dijkstra's algorithm using a min-priority queue with a Fibonacci heap requires $O(|E| + |V| \log |V|)$ time. There also exist a plethora of sequential SSSP algorithms (including Bellman-Ford) and their variants [10].

2.1 Sequential Algorithms for Updating SSSP

The structural changes in a dynamic network can be in the form of edge or vertex insertion and deletion as well as changing edge weights. In this paper, we only apply changes in the form of edge addition or deletion, since vertex insertion/deletion can be represented as changes to edges, given that the upper bound on the number of vertices that can be added is known. For a single-edge insertion or deletion scenario, the first step is to identify the affected vertices in the graph, and enter them to a priority queue. In the next phase, using state-of-the-art implementation of Dijkstra's algorithm the distance of the vertices in the queue are updated. This continues until the queue is empty or all the affected nodes and their neighbors from the source have the shortest paths. The sequential approach involves redundant computation as the same set of vertices can be updated multiple times and therefore cannot scale to large networks. Algorithm 1 presents how to update SSSP given an input undirected graph G, initial SSSP tree and a changed edge $\Delta E = e(u, v)$. (An algorithm to update the SSSP tree in a similar sequential manner is proposed in [4].)

Algorithm 1. Updating SSSP for a Single Change

Input: Weighted Graph $G(V, E)$, SSSP Tree T, Changed edge $\Delta E = e(u, v)$
Output: Updated the SSSP Tree T_u

1: **Function** `SingleChange` (ΔE, G, T):
`// Find the affected vertex,` x
2:
3: **if** $Dist_u[u] > Dist_u[v]$ **then**
4: $x \leftarrow u, y \leftarrow v$
5: **else**
6: $x \leftarrow v, y \leftarrow u$
`// Initialize Priority Queue` PQ `and update` $Dist_u[x]$
7: $PQ \leftarrow x$
8: **if** E *is inserted* **then**
9: $Dist_u[x] \leftarrow Dist_u[y] + W(u, v)$
10: **if** E *is deleted* **then**
11: $Dist_u[x] \leftarrow infinity$
`// Update the subgraph affected by` x
12: **while** *(PQ not empty)* **do**
13: $Updated \leftarrow False$
14: $z \leftarrow PQ.top()$
15: $PQ.dequeue()$
16: $Updated \leftarrow SSSP$(z, G, T)
`// Calculate the shortest distance from source vertex to the` z
17: **if** $Updated = True$ **then**
18: **for** n *where* n *is the neighbor of* z **do**
19: $PQ.enqueue(n)$

Over the last 30 years, many sequential implementations of updating or incremental algorithms for dynamic networks have been developed. The authors in [5] developed complexity models for the sequential update of dynamic networks. The algorithms in [6] suggest techniques to update the SSSP tree. An implementation to update SSSP in batches is propsoed in [11]. The dynamic SSSP algorithm due to [12] is one of the most recent sequential update algorithms, aiming to reduce the complexity of updates.

3 Related Work

This section highlights related work on parallel implementations for computing SSSP on static and dynamic networks.

3.1 GPU-Based Implementation

Gunrock [8] is a high-performance graph library that provides a data-centric abstraction of a set of vertices or edges and develops a three-step architecture (advance, filter, and compute) to compute SSSP on GPU. The authors in [13] proposed an efficient implementation of the Bellman-Ford algorithm using two queues on the Kepler GPU. Although this algorithm exploits dynamic parallelism, a feature of modern Kepler GPUs, it also uses atomic operation that makes part of the code serial.

In [14], a detailed study is proposed on the performance of various graph algorithms, including SSSP on temporal graphs, on different multicore architectures and GPU accelerator. In [15], a dynamic incremental and decremental SSSP algorithm in JavaScript is implemented. This is the only GPU implementation we found for updating dynamic networks in GPU. Results show the algorithm performs well, only if the number of changed edges is less than 10 percent.

3.2 CPU-Based Implementation

In [7] we developed the first shared-memory algorithm for updating SSSP on dynamic networks, and implemented it using OpenMP. To the best of our knowledge, this is the only shared-memory parallel implementation for updating SSSP on dynamic networks. The ones mentioned below operate on static networks or are sequential implementations on dynamic networks. For example, a Spark-based implementation to update SSSP on dynamic networks is reported in [16], while [17] implemented the Bellman-Ford algorithm using parallel hypergraph algorithms and [18] provided two implementations of Δ–stepping algorithm on static graph in shared memory multicore architecture.

Galois [19] is a shared-memory based amorphous data-parallelism programming model for graph algorithms. It supports priority scheduling and processes on active elements comprised of a subset of vertices. Galois provides a shared-memory parallel implementation of Dijkstra's algorithm. Havoqgt [20] is a software to compute SSSP on a distributed platform; it only allows re-computation from scratch, with no support for dynamic networks. Approximations for streaming graphs are proposed in [21].

4 Our Parallel Dynamic SSSP Algorithm

In this section, we propose a novel framework for updating SSSP in parallel for dynamic networks. We maintain the SSSP values as a tree, which is rooted at the starting vertex. Our algorithm involves efficiently traversing and updating the weights or the structure of the tree. A sketch of the framework is given in Fig. 1.

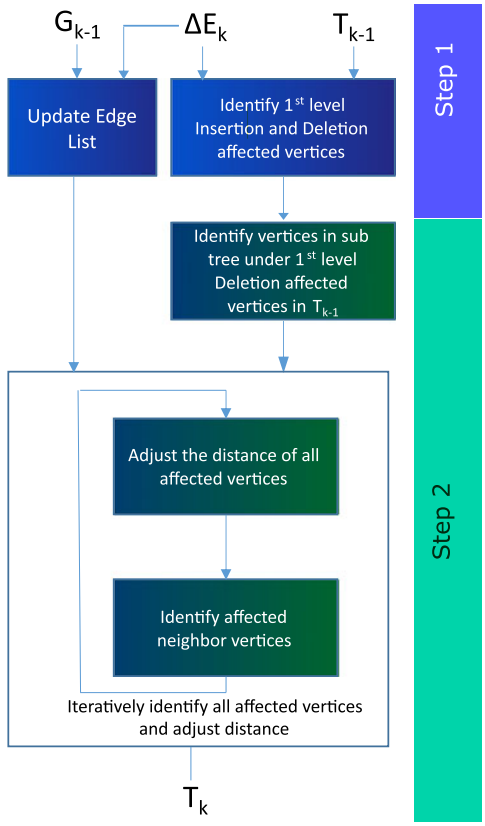

Fig. 1. A Parallel framework to update SSSP in dynamic networks.

Problem Description. Let $G_k(V_k, E_k)$ denotes the graph at time step k. Since SSSP results are non-unique, several SSSP trees are possible. Let T_k be one such SSSP tree of G_k.

Let ΔE_k be the set of edges changed from the time step $k-1$ to time step k. This set consists of two subsets; the set of inserted edges Ins_k and the set of deleted edges Del_k. Thus, $E_k = (E_{k-1} \cup Ins_k) \backslash Del_k$.

Our *goal* is to compute the SSSP tree T_k at time step k, based on the structure and edge weights of the tree T_{k-1}at time step $k-1$ and the changed edge set ΔE_k.

4.1 A Parallel Framework for Updating SSSP

Our framework for updating SSSP consists of two steps; *Step 1:* Identify the subgraph (a set of vertices) affected by changes (Algorithm 2); and *Step 2:* Update this subgraph to create a new SSSP tree (Algorithm 3).

Data Structures. We store the SSSP tree as an adjacent list. Since the number of edges in a tree is one less than the number of vertices, thus the memory would be of order V. Each vertex, except the root, is associated with four variables; *Parent*– its parent in the tree, *Dist*– its shortest distance from the root , *Affected_Del*–whether the vertex has been affected due to deletion, and *Affected*– whether the vertex has been affected by any changed edge. Each of these variables are stored as arrays of length V, with the each index associated with the corresponding vertex.

Step 1: Identifying Subgraphs Affected by the Changed Edge. In this step we process each edge in parallel and identify the vertices affected by these changes. Note that a vertex can be affected due to multiple changes. However, this step determines whether a vertex is affected by *at least one* changed edge. Due to this criterion, no synchronization is required and each changed edge can be potentially be processed in parallel.

In practice, however, the computation is expedited if part of the new SSSP tree is formed in Step 1. This includes removing deleted edges (that can result in some subtrees being disconnected) and changing the parent of affected vertices whose distance to the root is lowered due to insertions.

Edge Deletion. Given two vertices u and v, let the edge between them be $e(u, v)$. In the case of deletion, we first check whether this edge was part of the SSSP tree (Algorithm 2, line 5). If it is not, then it does not affect any vertex, and no further processing is needed.

For the case where the edge $e(u, v)$ is part of the SSSP tree, let us assume that u is the parent of v. Thus only the shortest distance of v is affected. We change $Dist[v]$ to infinity and $Parent[v]$ to $null$, to indicate that vertex v is now disconnected from the parent tree. We further change $Affected\_Del[v]$ and $Affected[v]$ to $true$ to indicate that this vertex is affected by deletion (Algorithm 2; lines 7, 8).

Edge Insertion. Let the weight of an inserted edge $e(u, v)$ be $W(u, v)$. The edge will affect the SSSP tree only if $Dist[u] + W(u, v)$ is less than $Dist[v]$.

If this condition holds, then we set $Dist[v]$ to $Dist[u] + W(u, v)$, update the parent of v, $Parent[v]$ to u, and mark $Affected[v]$ to true to mark that vertex v has been affected (Algorithm 2, lines 15-18).

Algorithm 2. Identify Affected Vertices

```
1: Function ProcessCE (G, T, ΔE, Parent, Dist):
2:   Initialize Boolean arrays Affected_Del and Affected to false
3:   G_u ← G
4:   for each edge e(u,v) ∈ Del_k in parallel do
5:     if e(u,v) ∈ T then
6:       y ← argmax_{x∈{u,v}}(Dist[x])
7:       Change Dist[y] to infinity
8:       Affected_Del[y] ← True
           Affected[y] ← True
9:     Mark e(u,v) as deleted
10:  for each edge e(u,v) ∈ Ins_k in parallel do
11:    if Dist[u] > Dist[v] then
12:      x ← v, y ← u
13:    else
14:      x ← u, y ← v
15:    if Dist[y] > Dist[x] + W(x,y) then
16:      Dist[y] ← Dist[x] + W(x,y)
17:      Parent[y] ← x
18:      Affected[y] ← True
19:    Add e(u,v) to G_u
```

Step 2: Updating the Affected Subgraph. Note that when a vertex gets disconnected from the SSSP tree due to edge deletion, then its descendants would also get disconnected. Thus their distance from the root would be infinity. In the first part of Step 2, we update these distances by traversing through the subtree rooted at the vertices affected by deletion. The distances of all the vertices in these disconnected subtrees are set to infinity. We set the $Affected$ variable for each of these vertices as true. Since the subtrees can be of different sizes, we use a dynamic scheduler for balancing the workload. The pseudo code in given in Algorithm 3, lines 2-8.

In the second part, we update the distances of the affected vertices. For each vertex marked as $Affected$ and its neighbors, we check whether the distance is reduced. If the distance of a neighbor, n is reduced by passing through an affected vertex v, then the distance of n, $Dist[n]$, is updated and $Parent[n]$ is set to v. On the other hand if the distance of v is reduced by passing through a neighbor, n, then $Dist[v]$ is updated and $Parent[v]$ is updated to n. This process is continued iteratively until there is no vertex whose distance can be updated. Since the vertices are always moving towards lower distance, the iterations will converge.

At the end of Step 2, we obtain an SSSP tree with updated distances for every vertex from the root at time t_k. Fig. 2 shows an example of SSSP tree update, where $Dist$ values of the vertices are shown in red color and the affected vertices at each step are shown in green color. Figs. 2c, 2d, 2e, 2f, and 2g show the step and the sub-steps of the algorithm.

Algorithm 3. Update Affected Vertices

```
1: Function UpdateAffectedVertices (G_u, T, Dist,
   Parent Affected_Del and Affected):
2:   while Affected_Del has true values do
3:     for each vertex v ∈ Vsuch that Affected_Del[v] = true
       in parallel do
4:       Affected_Del[v] ← false
5:       for all vertex c, where c is child of v in the SSSP tree T do
6:       Set Dist[c] as infinity
7:       Affected_Del[c] ← True
8:       Affected[c] ← True
9:   while Affected has true values do
10:    for each vertex v ∈ Vsuch that Affected[v] = true in parallel do
11:      Affected[v] ← False
12:      for vertex n, where n ∈ V and n is neighbor of v do
13:        if Dist[n] > Dist[v] + W(v,n) then
14:          Dist[n] ← Dist[v] + W(v,n)
15:          Parent[n] ← v
16:          Affected[n] ← True
17:        else if Dist[v] > Dist[n] + W(n,v) then
18:          Dist[v] ← Dist[n] + W(n,v)
19:          Parent[v] ← n
20:          Affected[v] ← True
```

4.2 Challenges in Achieving Scalability

We discuss the challenges that arise in making the code scalable and how we address these challenges;

Load Balancing. In Step 2 of our algorithm, the number of operations depends on the size of the subgraph rooted at the affected vertices. Since the subgraphs can be of different sizes, the work done by each processing unit can be imbalanced.

While several complicated load balancing techniques can be applied, such as knowing the size of the subtrees apriori and then distributing the vertices according to the size of the subtrees, our experiments show that by simple using

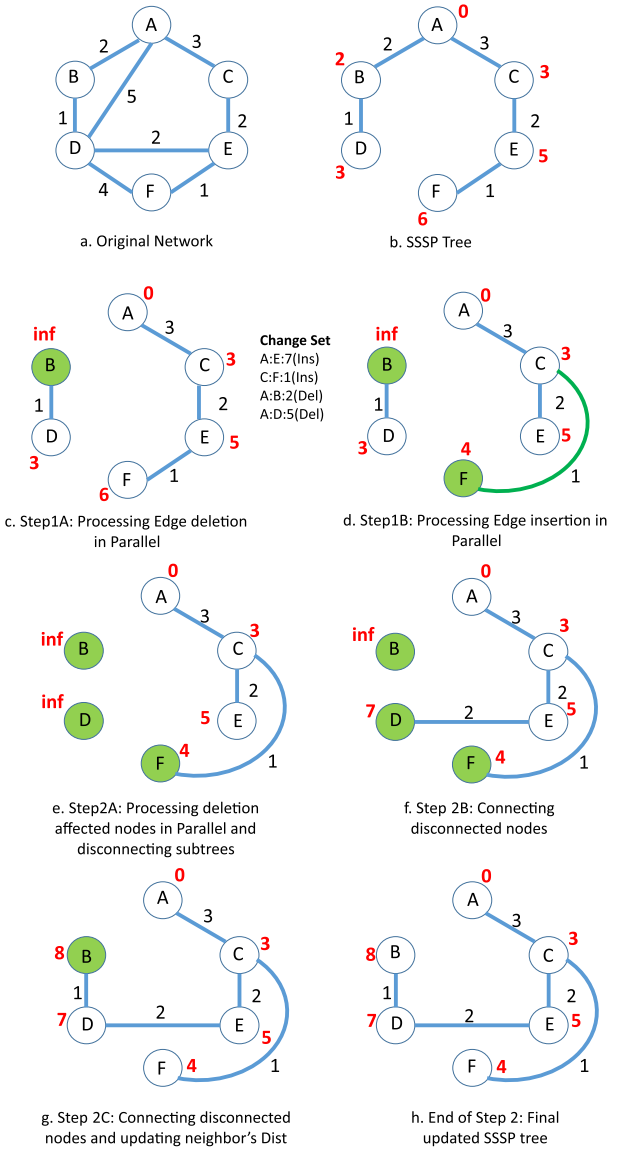

Fig. 2. Illustration of updating SSSP tree.

dynamic scheduling provide good scalability with much lower overhead.

Synchronization. Since multiple processors are updating the affected vertices, thus race conditions can occur where these vertices may not be updated to their minimum distance from the root. The traditional method to resolve this problem is by using locking constructs. However using these constructs reduces the scalability of the implementation.

In our algorithm, instead of using locking constructs, we iteratively update the distances of the vertices to a lower value. Over the iterations the vertices thus converge to their minimum distance from the source node or root of the tree. Although multiple iterations add to the computation time, the overhead is much lower than using locking constructs.

Avoiding Cycle Formation. Our updating algorithm is based on the assumption that the underlying data structure is a tree. However, when multiple edges are inserted in parallel, cycles can be formed.

Consider an edge $e(u, v)$ which is deleted, and v is the affected vertex. Then as per Step 1, the parent of vertex v is set to null to indicate it is disconnected from the main tree.

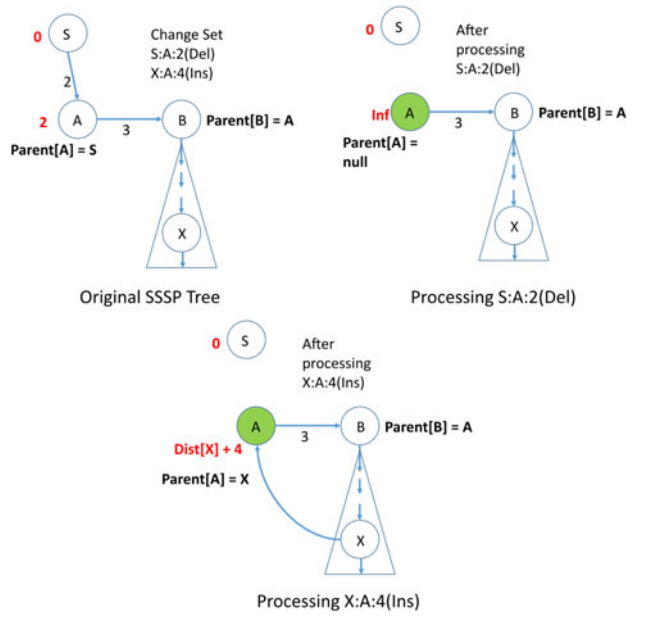

Fig. 3. Type 2 cycle formation.

Let a be a descendant of vertex v, that is it is a node in the disconnected subtree rooted at v. At the end of step 1, the distance of a is not yet updated to infinity. Now if there is an inserted edge $e(a, v)$ from a to v, then the distance of v will be lower after passing through a. Thus a will be set as the parent of v creating a cycle. Fig. 3 illustrates the formation of this kind of cycle.

To avoid this cycle formation, in Step 2, we first process the subtrees of the vertices affected by deletion to mark their distances as infinity. Then the edge between two disconnected vertices in the tree are not processed.

4.3 Algorithm Analysis

We now prove that our algorithm indeed forms a correct SSSP tree for the graph at time step k, and also computes the analytical scalability of our method. These results are reproduced from our earlier paper [7].

Lemma 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 our parallel updating algorithm, 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 2. *The tree obtained by our parallel algorithm will be a valid SSSP tree for G_k.*

Proof. Let T_k be a known SSSP tree of G_k 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_k.

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

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

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

Now consider when $b \neq c$. Since the edge (b,a) exists in T_k, it also exists in G_k. Since $Dist_{T_k}[v] \neq Dist_{T_{alg}}[v]$, the distance of a was updated either in T_k 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_k and a is an affected vertex, in Step 2, 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_k, 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, $Dist_{T_k}[a] = Dist_{T_{alg}}[a]$. □

Scalability of the Algorithm. Assume that we have p processing units, and m changed edges to process. For Step 1, each changed edge can be processed in parallel, requiring time $O(m/p)$.

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 $O(\frac{m}{p})$+$O(\frac{Dxd}{p})$. 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 $E_x < (E + m)$.

5 Implementation Details

In this Section we discuss how we implemented the updating algorithm in shared memory architectures and in GPUs.

5.1 Shared-Memory Implementation

We implemented the proposed algorithm on the shared memory platform where the input consists of a graph $G(V,E)$, changed edges $\Delta E(Del_k, Ins_k)$, source vertex s, and SSSP tree T.

We use *pragma omp parallel for* directive to leverage the shared memory parallelism. The implementation processes the set of Del_k in parallel before processing the set of Ins_k in parallel. In case of deletion, Del_k is first validated by searching if the edge is available in key edges stored in T; if it belongs to the key edges it is processed further, otherwise it is ignored. If the changed edge $e(u,v)$ is marked for insertion, the next step is to check if the addition of this edge helps to reduce the distance of vertex v from the source; if it does, then the edge is marked for update.

Algorithm 4. Asynchronous Update of SSSP

Input: $G(V,E)$, T, $Dist$, $Parent$, source vertex s, $\Delta E(Del_k, Ins_k)$
Output: Updated SSSP Tree T_k

```
1: Function AsynchronousUpdating (ΔE, G, T):
2:   Set Level of Asynchrony to A.
3:   Change ← True
4:   while Change do
5:     Change ← False
6:     pragma omp parallel schedule (dynamic)
7:     for v ∈ V do
8:       if Affected_Del[v] = True then
9:         Initialize a queue Q
10:        Push v to Q
11:        Level ← 0
12:        while Q is not empty do
13:          Pop x from top of Q
14:          for c where c is child of x in T do
15:            Mark c as affected and change distance to
               infinite
16:            Change ← True
17:            Level ← Level + 1
18:            if Level ≤ A then
19:              Push c to Q
20:  Change ← True
21:  while Change do
22:    Change ← False
23:    pragma omp parallel schedule (dynamic)
24:    for v ∈ V do
25:      if Affected[v] == True then
26:        Affected[v] ← false
27:        Initialize a queue Q
28:        Push v to Q
29:        Level ← 0
30:        while Q is not empty do
31:          Pop x from top of Q
32:          for n where n is neighbor of x in G do
33:            Level ← Level + 1
34:            if Dist[x] > Dist[n]+W(x,n) then
35:            Change ← True
36:            Dist[x] = Dist[n]+W(x,n)
37:            Parent[x] = n
38:            Affected[x] ← True
39:            if Level ≤ A then
40:              Push x to Q
41:          if Dist[n] > Dist[x]+W(n,x) then
42:            Change ← True
43:            Dist[n]=Dist[n]+W(n,x)
44:            Parent[n] = x
45:            Affected[n] ← True
46:            if Level ≤ A then
47:              Push n to Q
```

In this implementation, each vertex has a Boolean *Affected* flag which is true when it is identified that the SSSP of the

vertex needs to be recalculated. Now for all the affected vertices, SSSP is computed using the unaffected adjacent vertices and their distances as mentioned in the last part of Algorithm 3. The computation of affected vertices is done in parallel and this process continues until there is no vertex left with the *Affected* flag set to true. Since a vertex can be set to affected multiple times, this process incurs some redundancy of computation. However, the alternate process of using locking constructs to synchronize lead to reduced scalability. We therefore allow for some redundancy in the operations. The shared memory implementation uses C++ and OpenMP library.

Asynchronous Updates. In Step 2 of our updating algorithm, the threads synchronize at the while loop. This synchronization occurs after each vertex accesses its neighbors. However the synchronization can be made less frequent if the vertices access their distance d neighbors for a given value of d. Increasing this level of asynchrony can lead to more redundant computations and increased iterations, but reduced number of synchronizations. In the experimental study (see Section 6), we discuss how changing the levels of synchronization affects performance. The pseudocode in Algorithm 4 demonstrates how asynchronous updates can be implemented for Step 2 of our algorithm.

Processing Batches of Changes. The original implementation discussed in [7] processes all changed edges together. In order to improve the load balancing and avoid having memory hotspot we processed the changed edges in batches. We experimented with different batch sizes to study the performance. Section 6 shows how processing in batches of different size gives a boost in the performance.

5.2 GPU Implementation

To leverage the Single Instruction Multiple Threads (SIMT) execution model of modern NVIDIA GPU, in our implementation we create an array of operands where a single CUDA (Compute Unified Device Architecture) thread is assigned to a single element of the operand array to execute some user defined function. In this architecture a large number of threads (divided into thread, block and grid at the programming level) are executed simultaneously to achieve data-parallelism. Commonly, Grid-Stride Loops are used to cover the whole operand array when the size of the array is larger than the number of CUDA threads available.

Compressed sparse row (CSR) and its variants like Dynamic CSR-based (DCSR) data structure [22] or Packed Compressed Sparse Row (PCSR) [23] are potentially suitable for storing input graphs depending on the application. Here, we consider simple CSR format to store the graph. To store the SSSP tree structure and its properties, we define a data structure T, which is an array of size $|V|$, where the element at index i denotes the properties of vertex i in the SSSP tree. Each element of T contains three variables: *Parent* (stores the parent of the vertex), *Dist* (stores the distance of the vertex from the source), and *Flag* (a Boolean variable to denote the affected vertex).

5.2.1 Functional Block

When numerous threads are operating on shared variables and attempting to write concurrently, it may lead to race condition or erroneous update. To avoid this situation, CUDA atomic operations are used. However, the use of atomic operation makes the computation serial and increases the overall execution time. Therefore, we define a *functional block*, which minimizes the use of atomic operation while maintaining the correctness of the algorithm.

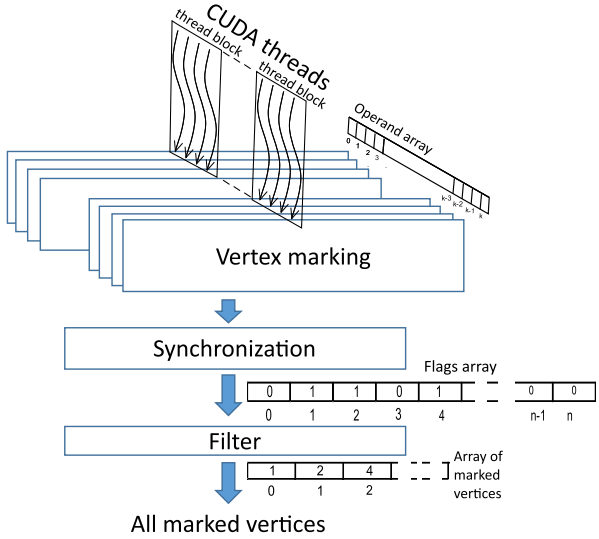

Fig. 4. Vertex-marking functional block (VMFB).

Our functional block, called the *Vertex-Marking Functional Block* (VMFB), is created solely for graph-related operations. It can accept an operand array $Array_{operand}$, a function Fx, an array (*Flags*) of size $|V|$ to store flags for each vertex of a graph $G(V, E)$ and a predicate P as input. The VMFB has three main functional steps, as described below.

Vertex Marking. The VMFB allocates parallel CUDA thread for each element of $Array_{operand}$, while each thread performs Fx and alters the flag values in the shared array, *Flags*. It is considered that Fx is capable of doing other computations along with changing the flag values (primary function). Here, all the CUDA threads operate without any guaranteed synchronization (no control over synchronization at this stage) without using any atomic operation. They update *Flags* with a constant value (e.g., updating to 1) only when some specified condition mentioned in Fx is satisfied.

Although, in this step multiple threads can asynchronously update the same element of the array *Flags*, the correctness is preserved as all threads update a flag with a single constant value. More specifically, the process is one-directional and a vertex can be marked in the process, but cannot be unmarked.

Synchronization. This step puts a global barrier for all the threads, and the code execution halts here for all threads to complete their execution before going further.

Filter. This step stores the index value from the array *Flags* using CUDA *__ballot_sync* when the predicate P is true. In general, the predicate simply states comparison between flag value and a given constant. The filter step identifies and stores all marked vertices in an array without duplication. Moreover, this operation reduces the redundant computation due to overlapping affected subgraphs.

Fig. 4 depicts the VMFB, in which only the filter operation uses the CUDA *atomicAdd* operation while maintaining the dimension of the list of marked vertices.

TABLE 1
Notation Used for GPU Implementation

Used Notation	Detail
VMFB	Vertex Marking Functional Block
ProcessIns()	Operate on an edge being inserted and finds 1st level insertion affected vertices
ProcessDel()	Operate on an edge being deleted and finds 1st level deletion affected vertices
DisconnectC()	Operate on a deletion affected vertex and disconnects child neighbors
ChkNbr()	Operate on an affected vertex to connect disconnected vertices and update neighbors
P	Predicate to find if a flag is raised
Reset()	Function to reset all values of an array to 0

5.2.2 SSSP Update Using Vertex Marking Functional Block

The idea of implementing a graph algorithm using VMFB is simple. We write a function which takes a single vertex or a single edge and operates on it and marks some vertices in the flag array depending on user defined clauses. Let our function can work on a vertex, then we pass an array of vertices(as operand array), the function we have written and an associated flag array to a VMFB. VMFB executes the user defined function on every element of the operand array in parallel and generates an array of marked vertices. Therefore, to implement SSSP update we first write four functions named $ProcessIns$, $ProcessDel$, $DisconnectC$ and $ChkNbr$. The functionality of these are given in Table 1.

Algorithm 5. GPU Implementation

Input: $G(V, E)$, $T(Dist, Parent)$, $\Delta E(Del_k, Ins_k)$
Output: Updated SSSP Tree T_u
1: **Function** `GPUimplementation` (G, T, ΔE):
2: Initialize a flag array of size $|V|$ named $Flags$ with all values 0
`/* Process` Del_k `in parallel */`
3: $Aff_{del} \leftarrow VMFB(Del_k, ProcessDel(G, T), Flags, P)$
`/* Process` Ins_k `in parallel */`
4: $Reset(Flags)$
5: $Aff_{ins} \leftarrow VMFB(Ins_k, ProcessIns(G, T), Flags, P)$
`/* Mark all vertices in deletion affected sub-tree */`
6: $Aff_{Alldel} \leftarrow Aff_{del}$
7: **while** Aff_{del} *is not empty* **do**
8: $Reset(Flags)$
9: $Aff_{del} \leftarrow VMFB(Aff_{del}, DisconnectC(T), Flags, P)$
10: $Aff_{Alldel} \leftarrow Aff_{Alldel} \cup Aff_{del}$
`/* Connect disconnected vertices and update neighbor distance */`
11: $Aff_{All} \leftarrow Aff_{Alldel} \cup Aff_{ins}$
12: **while** Aff_{All} *is not empty* **do**
13: $Reset(Flags)$
14: $Aff_{All} \leftarrow VMFB(Aff_{All}, ChkNbr(G, T), Flags, P)$

Algorithm 5 presents the pseudocode for GPU implementation using VMFB blocks and the notations used in the pseudocode are listed in Table 1.

TABLE 2
Networks in Our Test Suite

Name	Num. of Vertices	Num. of Edges
BHJ2015-M87101049(BHJ-1)	1,827,148	193,540,306
BHJ2015-M87101705(BHJ-2)	1,827,168	202,250,875
soc-Orkut	2,997,166	106,349,209
LiveJournal	12,693,249	161,021,950
graph500-scale23-ef16	4,606,315	258,503,995
RMAT24_G	16,777,215	134,511,383

BHJ: bn-human-Jung.

Step 1. We implement the first step of the SSSP update algorithm using two VMFBs. The first VMFB (Line 3 in Algorithm 5) accepts the array of deleted edges as the operand array and marks the first-level affected vertices due to edge deletion. As additional function, it disconnects the first-level deletion affected vertices from their parent vertices in the SSSP tree. This block mainly implements lines 5–9 in Algorithm 2.

The second block takes the array of inserted edges as $Array_{operand}$ and identifies the first-level insertion affected vertices by processing them using $ProcIns$ function, which is actually the implementation of lines 11–19 in Algorithm 2.

As in VMFB, the parallel threads are distributed across the operand array, every changed edge is processed by different CUDA thread in Step 1.

Step 2. The second step of the SSSP update algorithm has two main parts. The first part iteratively disconnects the subtrees under every first-level deletion-affected vertex (using the $DisconnectC$ function) by passing the function and affected vertices to a VMFB. Specifically, the $DisconnectC$ function implements lines 4–8 in Algorithm 3. The VMFB here uses the output of the first VMFB (an array of first level deletion-affected vertices) as input $Array_{operand}$ and marks all the children of the first-level deletion-affected vertices in the SSSP tree and then disconnects them. In the next iteration, the newly affected vertices are used as the operand array for the same VMFB, and all the deletion-affected vertices are marked and disconnected iteratively (Algorithm 5, lines 7-10).

The last part of the implementation, connects the disconnected vertices and updates the distance of all affected vertices. The set of all deletion affected vertices and first-level insertion affected vertices are combined (via set union operation) to form an array of all affected vertices (both deletion and insertion affected) and is used as initial input operand array for the last VMFB (line 14 in Algorithm 5). The last VMFB block connects the disconnected vertices (applicable to deletion affected vertices) and updates the $Dist$ value of neighbors (applicable to both deletion and insertion affected vertices) using $ChkNbr$ function (implementation of lines 11–20 in Algorithm 3). Vertices are marked as affected whenever their $Dist$ values are changed and the filter operation in the block gathers all affected vertices in an iteration as an output array of marked vertices. In the next iterations, the output of the last iteration is provided as input into the VMFB. This process ends when the updated $Dist$ is computed for all affected vertices and there is no marked vertex left for further processing.

6 Experimental Results

For the experiments, we choose five large graphs from the network-repository collection [24]; and one synthetic

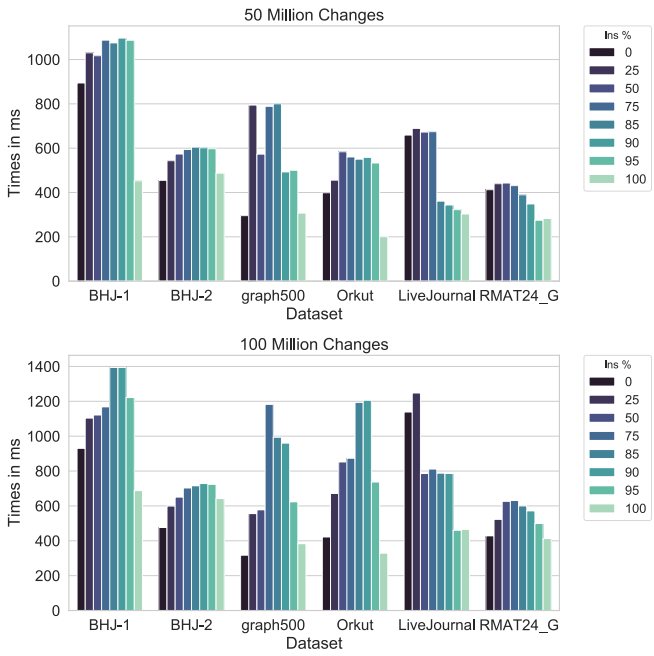

Fig. 5. Execution time of proposed GPU implementation for networks in Table 2 with 50M and 100M changed edges consisting of p = 100, 95, 90, 85, 75, 50, 25, and 0 percent insertions and $(100 - p)$% deletions.

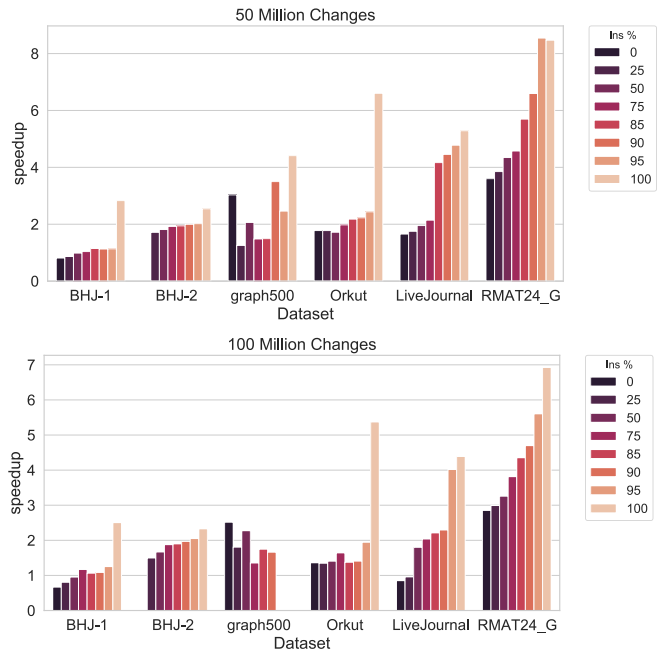

Fig. 6. Comparison of our GPU implementation of the SSSP update algorithm for dynamic networks with Gunrock implementation computing SSSP from scratch on static networks. The Y-axis is the speedup; the X-axis is the graph input. The speedup is measured for all networks with 50M and 100M changed edges for 0, 25, 50, 75, 85, 90, 95 and 100 percent edge insertions (the rest of the edge changes are deletions).

R-MAT graph generated with probability (a = 0.45, b = 0.15, c = 0.15, d = 0.25), labeled as G, which is a random network with scale-free degree distribution. Table 2 lists the names, the number of vertices and edges of our graph datasets.

All GPU experiments have been conducted on a NVIDIA Tesla V100 GPU with 32GB memory. The host processor was dual 32 core AMD EPYC 7452. For the shared memory experiments we used Intel(R) Xeon(R) Gold 6148 CPU with 384GB memory.

6.1 GPU Implementation Results

Fig. 5 shows the execution time of CUDA implementation of our SSSP update algorithm. We present the time for updating the shortest path following a 50-million and 100-million edge updates. The edge updates consist of different mixes of edge deletions and insertions. Specifically, edge insertion percentage means that if the dataset contains $p\%$ insertions then there is $(100 - p)\%$ edge deletion. For all of our experiments, only the execution time is used instead of TEPS (Traversed Edges per Second). This is because, for dynamic networks update, we typically aim to traverse fewer edges; thus maximizing TEPS is not the right metric.

The execution time mainly depends on the total number of affected vertices, which in turn depends on the network structure and location of changes. The changes and network structure are random in our experiments. Therefore, the affected subgraph size and its location are also random. In general, for $p = 100\%$ edge insertions (i.e., no edge deletion), the GPU implementation skips the part of the algorithm where all vertices in the deletion affected subgraph get disconnected and require comparatively less time. We observe that the execution time increases when the percentage of edge insertions is decreased from 100 percent, i.e., the percentage of edge deletions is increased. This is because, for edge deletion, the algorithm disconnects all deletion affected vertices at first and then tries to reconnect them with the actual SSSP tree. However, when one deletion affected subgraph overlaps with another deletion affected subgraph, the $Filter$ operation in VMFB combines them to avoid rework. Therefore, in Fig. 5 we notice that most of the time, the execution time decreases when the percentage of insertions is further reduced (i.e., the percentage of deletions is increased) below 75 percent. Although, an exception to this trend occurs when the change edges are random and the affected subgraphs do not overlap.

6.2 GPU Performance Comparison

We compare the GPU implementation of our SSSP update algorithm with other available GPU implementation for computing SSSP. The implementation reported in [15] is not appropriate for comparison because because it is useful only if the edge updates are less than 10 percent of the total number of edges. To the best of our knowledge, there is no other implementation of updating SSSP on dynamic networks on GPUs. Therefore, we compare the time taken by our algorithm to update the SSSP with the time taken to recompute the SSSP from scratch by using Gunrock [8], a widely used tool for (static) graph analytics on GPUs. The Gunrock implementation shows good speedups when compared to other GPU graph analytics packages.

Fig. 6 shows the speedup of the SSSP update algorithm compared with the Gunrock SSSP implementation on the networks discussed in Table 2. The changed edges are added to the original network and used as an input to the Gunrock software. We only consider the time taken by Gunrock to compute SSSP for the modified network, ignoring the time to create the input network with new changes. Each comparison experiment was repeated six times; we took the average and measured the improvement.

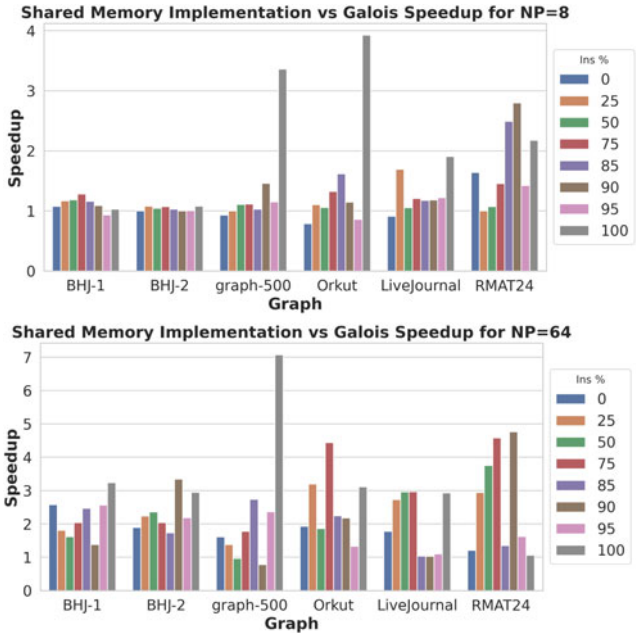

Fig. 7. Speedup of our shared-memory update algorithm compared to the recomputation approach in Galois.

Fig. 6 clearly shows that, in most of the cases, our implementation achieves good speedup (up to 8.5x for 50 million changed edges and up to 5.6x for 100 million changed edges) when the percentage of insertion is more than 25 percent. When the percentage of insertion is below 25 percent (i.e., the deletion percentage is above 75 percent), it implies 75 percent or more of the total 50 million (or 100 million) changed edges are deleted from the original set of edges and the rest of the edges (which is very small) is supplied to Gunrock as input. This is why at lower percentage of insertion (i.e., high deletion), Gunrock performs better than our approach.

We observed that if the number of total changed edges is greater than 50 percent of the network size and the majority (more than 50 percent) of the changed edges involve edge deletion, it is better to recompute from scratch rather than updating. Our proposed implementation works better in most cases when the majority of the changed edges are marked for insertion.

6.3 Shared-Memory Implementation Results

We compare our shared-memory implementation with the state-of-the-art CPU based shared memory recomputation approach [25] on a dual 40-core Intel(R) Xeon(R) Gold 6148 CPU @ 2.40 GHz with 384 GB DDR4 2666 RAM. Fig. 7 provides a comparison between the proposed shared memory implementation and Galois [25] for all graphs. We have used 100M edge changes with different combinations of edge insertion percentage. For the recomputation based approach we modify the network by adding the changes and then re-run the SSSP from scratch. We ignore the time it takes to add those changes to the original network. We observe that in most cases, the proposed shared-memory implementation performs better than the recomputing approach. We performed this experiment for all networks three times with three different set of changed edges generated for each network. The average speed up for each network for different edge percentage is used for comparison. We didn't see much difference in the speedup for all 3 runs. We did observe that speed up is not consistent for networks such as Graph-500, we noticed that the speedup is less than 1 when more than 85 percent percentage of total nodes affected for the given changes. We believe that if the changes affect more than a certain threshold percentage which varies for each network than recomputation is recommended than using the update algorithm. For the networks used in this paper for scalability experiments in Fig. 8 the threshold ranges were around 75–85 percent.

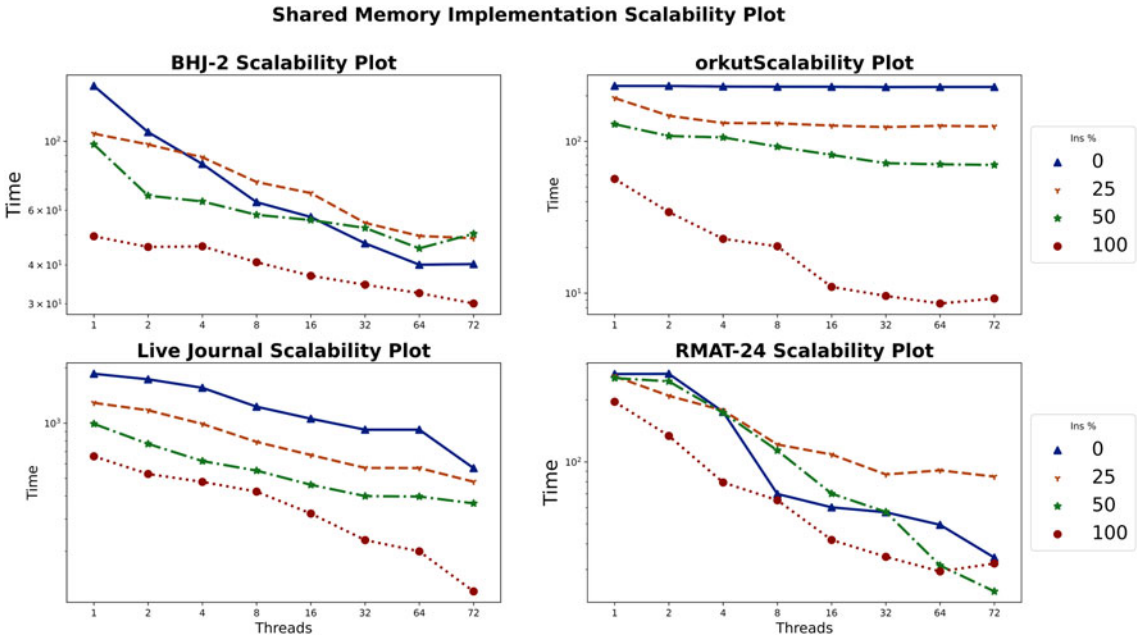

Fig. 8. Shared-memory implementation scalability plot for the four networks: BHJ-2, Orkut, Live Journal, and RMAT-24. Here the X-axis is the number of threads and the Y-axis is the runtime in log scale. For this experiment, we used 100M changes and different combinations of edge insertion percentages.

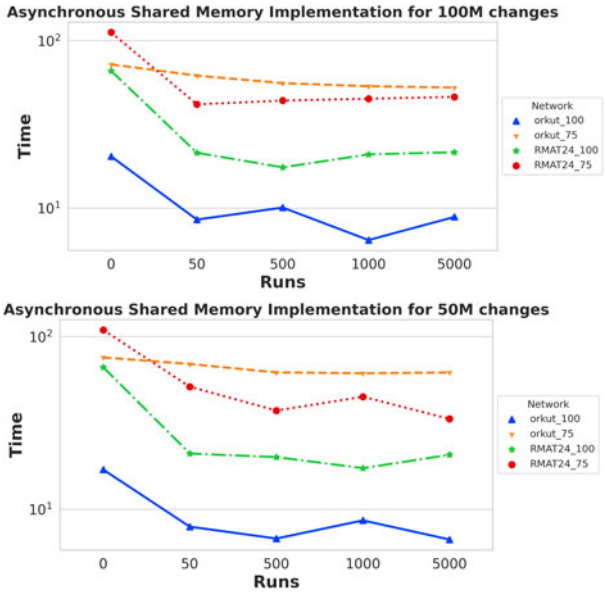

Fig. 9. Shared-memory asynchronous implementation for 100M and 50M changes with different levels of asynchrony. The X-axis is the level or length to be traversed before synchronization, and the Y-axis is the execution time in log scale. For this experimental setup, we used 8 threads; we tried different thread combinations and the execution with 8 threads gave good performance.

Fig. 8 shows the performance of the shared memory implementation with different combinations of edge insertion percentages. For this experiment, we ran the proposed implementation on four networks with 100 M changes on different thread counts. As we increase the number of threads, the time for updating decreases in most cases. We ran the proposed implementation on five different sets of changed edges (generated for each network) for the four networks and with different combinations of edge insertion percentage and averaged the time of five runs for each configuration. There are few cases where time to update doesn't show significant improvement when we increase the thread count. For example the Live Journal network for 25 and 50 percent edge insertion, Orkut Network for 0 percent. Our investigations revealed that the given changes did not affect more than 10–15 percent of the nodes (for each run) where as for the networks which didn't show this trend around 55–60 percent of the nodes were affected during the multiple runs. We did also notice few spikes for Live journal and RMAT 24, but when we took average of five runs, there were only two occasions where we observed spikes for those networks. We believe this could be due to load imbalance, however the remaining three times with the same input network and change edges the time to update decreased and there were no spikes.

The proposed implementation is sensitive to changes and the overall percentage of nodes being affected due to the changes. If the percentage of nodes affected is above 80, we do see time to update decreases but the change is not very significant.

In Fig. 9, we explore how increasing the level of asynchrony impacts the execution time. The definition of asynchrony is provided in Section 5.1. We observed that for both 50M and 100M changes on two networks, Orkut and RMAT-24, with edge insertion percentages of 100 and 75 percent, increasing the level of asynchrony reduces the execution time in all cases except for the Orkut network where 75 percent of changed edges are marked for insertion.

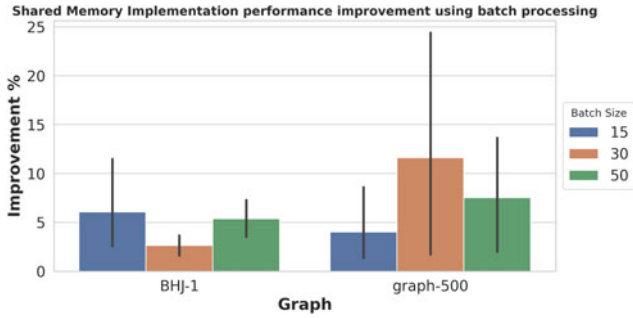

Fig. 10. Shared-memory implementation performance improvement when edges are processed in batches.

In Fig. 10, we explore how processing changed edges in batches can help improve the performance of the proposed implementation. For this experiment we took 100M changed edges and we choose one BHJ-1 and graph-500 network and processed changed edges with (25,50, and 100) percent edge insertion in batches of different sizes (15, 30, and 50). We compared this with our original implementation [7], which doesn't use any batch processing for the same networks, change edges, and for 64 and 72 threads. We repeated this experiment 10 times and then took the average. We observe that processing changed edges in batches does show performance improvement for higher thread counts, such as 64 and 72. For lower thread counts we did not see significant performance improvement when edges are processed in parallel.

7 Conclusion

In this paper we proposed a novel parallel framework to update SSSP graph property for dynamic networks. The framework was implemented on two different HPC environments, namely a CPU-based shared memory implementation and an NVIDIA GPU-based implementation. Our experiments on real-world and synthetic networks show that the proposed framework performs well compared to the re-computation based approaches (e.g., Galois for CPU and Gunrock for GPU) when the edge insertion percentage is 50 percent or more of the total changed edges. Our observation also shows that as the number of deletions increases beyond 50 percent, the recomputing approach performs well.

In future we plan to extend our framework with a hybrid approach where the changes for a given batch can determine which approach—recomputing or updating—would provide faster results. Investigation of the performance for non-random batches is one of our future interests. We also plan to explore predictive algorithms, where knowing the changed set apriori can lead to more optimized updates.

Acknowledgments

This work was supported in part by the NSF OAC under Grants 1725755, 1725566, and 1725585 for the collaborative SANDY project and in part by the NSF OAC under Grant 1919789.

REFERENCES

[1] F. B. Sorbelli, F. Corò, S. K. Das, and C. M. Pinotti, "Energy-constrained delivery of goods with drones under varying wind conditions," *IEEE Trans. Intell. Transp. Syst.*, to be published, doi: 10.1109/TITS.2020.3044420.

[2] X. Wu, G. Chen, and S. K. Das, "Avoiding energy holes in wireless sensor networks with nonuniform node distribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 5, pp. 710–720, May 2008.

[3] S. Ghariblou, M. Salehi, M. Magnani, and M. Jalili, "Shortest paths in multiplex networks," *Sci. Rep.*, vol. 7, no. 1, 2017, Art. no. 2142.

[4] G. Ramalingam and T. Reps, "On the computational complexity of dynamic graph problems," *Theor. Comput. Sci.*, vol. 158, no. 1–2, pp. 233–277, 1996.

[5] L. Roditty and U. Zwick, "On dynamic shortest paths problems," *Algorithmica*, vol. 61, no. 2, pp. 389–401, Oct. 2011.

[6] P. Narvaez, K.-Y. Siu, and H.-Y. Tzeng, "New dynamic algorithms for shortest path tree computation," *IEEE/ACM Trans. Netw.*, vol. 8, no. 6, pp. 734–746, Dec. 2000.

[7] S. Srinivasan, S. Riazi, B. Norris, S. K. Das, and S. Bhowmick, "A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks," in *Proc. IEEE Int. Conf. High Perform. Comput.*, 2018, pp. 245–254.

[8] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, pp. 1–12.

[9] S. Srinivasan, S. Pollard, S. K. Das, B. Norris, and S. Bhowmick, "A shared-memory algorithm for updating tree-based properties of large dynamic networks," *IEEE Trans. Big Data*, to be published, doi: 10.1109/TBDATA.2018.2870136.

[10] T. H. Cormen, C. E. Leriserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[11] R. Bauer and D. Wagner, "Batch dynamic single-source shortest-path algorithms: An experimental study," in *Proc. Int. Symp. Exp. Algorithms*, 2009, pp. 51–62.

[12] M. Alshammari and A. Rezgui, "A single-source shortest path algorithm for dynamic graphs," *AKCE Int. J. Graphs Combinatorics*, vol. 17, no. 3, pp. 1063–1068, 2020.

[13] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler GPU architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 8, pp. 2222–2233, Aug. 2016.

[14] A. Rehman, M. Ahmad, and O. Khan, "Exploring accelerator and parallel graph algorithmic choices for temporal graphs," in *Proc. Int. Workshop Program. Models Appl. Multicores Manycores*, 2020, pp. 1–10.

[15] A. Ingole and R. Nasre, "Dynamic shortest paths using javascript on GPUS," in *Proc. IEEE Int. Conf. High-Perform. Comput.*, 2015, pp. 1–5.

[16] S. Riazi, S. Srinivasan, S. K. Das, S. Bhowmick, and B. Norris, "Single-source shortest path tree for big dynamic graphs," in *Proc. IEEE Int. Conf. Big Data*, 2018, pp. 4054–4062.

[17] J. Shun, "Practical parallel hypergraph algorithms," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 232–249.

[18] E. Duriakova, D. Ajwani, and N. Hurley, "Engineering a parallel δ-stepping algorithm," in *Proc. IEEE Int. Conf. Big Data*, 2019, pp. 609–616.

[19] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 456–471.

[20] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 549–559.

[21] K. Vora, R. Gupta, and G. Xu, "KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 237–251.

[22] J. King, T. Gilray, R. M. Kirby, and M. Might, "Dynamic sparse-matrix allocation on GPUs," in *Proc. Int. Conf. High Perform. Comput.*, 2016, pp. 61–80.

[23] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2018, pp. 1–7.

[24] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 4292–4293.

[25] K. Pingali *et al.*, "The tao of parallelism in algorithms," in *Proc. ACM SIGPLAN conf. Program. Lang. Des. Implementation*, 2011, pp. 12–25.

Arindam Khanda received the BTech degree in ECE from the Institute of Engineering and Management in 2015 and the MTech degree in software systems from BITS Pilani in 2019. He is currently working toward the PhD degree with the Missouri University of Science and Technology. His research interests include parallel programming models, dynamic graphs, and HPC.

Sriram Srinivasan received the PhD degree from the University of Nebraska, Omaha. His research focuses on developing parallel scalable dynamic graph algorithms.

Sanjukta Bhowmick received the PhD degree from Pennsylvania State University. She is currently an associate professor with the Computer Science and Engineering Department, the University of North Texas. She was a joint postdoc with Columbia University and Argonne National Lab. Her research interests include understanding change in complex network analysis, with a focus on developing scalable algorithms for large dynamic networks and developing uncertainty quantification metrics for network analysis.

Boyana Norris is currently an associate professor with the Department of Computer and Information Science, University of Oregon. Her research interests include parallel algorithms, performance modeling, automated performance optimization (autotuning) of parallel scientific applications, embedding of domain-specific languages into legacy codes, adaptive algorithms for HPC, and component-based software engineering for HPC.

Sajal K. Das (Fellow, IEEE) is currently a professor of computer science and Daniel St. Clair endowed chair of the Missouri University of Science and Technology. His research interests include parallel and cloud computing, sensor networks, mobile and pervasive computing, cyber-physical systems, IoT, smart environments, cybersecurity, and biological and social networks. He is currently the editor-in-chief of the *Pervasive and Mobile Computing* and an associate editor for the *IEEE Transactions of Mobile Computing*, the *IEEE Transactions on Dependable and Secure Computing*, and the *ACM Transactions on Sensor Networks*.