A Shared-Memory Algorithm for Updating Tree-Based Properties of Large Dynamic Networks

Sriram Srinivasan, Samuel D. Pollard, Boyana Norris, Sajal K. Das, *Fellow, IEEE*, and Sanjukta Bhowmick

Abstract—This paper presents a network-based template for analyzing large-scale dynamic data. Specifically, we propose a novel shared-memory parallel algorithm for updating tree-based structures or properties, such as connected components (CC) and minimum spanning trees (MST), on dynamic networks. The underlying idea is to update the information in a rooted tree data structure that stores the edges of the network that are most relevant to the analysis. Extensive experiments on real-world and synthetic networks demonstrate that, with the exception of the inherently sequential component for creating the rooted tree, our proposed updatiing algorithm is scalable and, in most cases, also requires significantly less memory, energy, and time than recomputing-from-scratch algorithm. To the best of our knowledge, this is the first parallel algorithm for updating MST on weighted dynamic networks. The rooted-tree based framework that we propose in this paper can be extended for updating other weighted and unweighted tree-based properties such as single source shortest path and betweenness and closeness centrality.

Index Terms—Dynamic networks, minimum spanning tree, connected components, shared memory algorithms

1 Introduction

An important problem in big data analysis is to efficiently update results as the data change over time. In this paper, we address this problem in the context of network analysis. Networks (or graphs) are mathematical models for studying systems of interacting entities. The vertices of the network represent the entities in the system and the edges represent their pairwise interactions. Such systems arise in a wide range of disciplines including bioinformatics [1], epidemiology [2] and social sciences [3], to name a few. Structural properties of these networks can provide insights into the characteristics of the underlying systems. For example, high centrality vertices can indicate lethal genes in biological systems [4].

Motivation. Most of the real-world complex systems evolve with time and the analysis of their corresponding network models must be updated. The systems and their network models can be very large, with millions of entities. These large-scale size of such networks paired with their rates of change make it essential to use parallel computing techniques to update or analyze them in a timely fashion. his motivates us to design efficient parallel algorithms for updating large dynamic network properties.

In this paper we present novel algorithms for updating tree-based properties of dynamic networks, specifically connected components (CC) and minimum weighted spanning trees (MST). These tree structures are fundamental building blocks in many applications, such as taxonomy [5], broadcasting in computer networks [6], image processing [7], and as a step in an approximate solution for the traveling salesman problem [8].

1.1 Use of Rooted Tree for Updating Weighted Trees

The primary computational costs in graph algorithms arise from graph traversals. Due to their unstructured nature, graphs exhibit poor locality of access, which in turn increases the time to traverse a path between two vertices. In the case of dynamic graphs, the updates related to each edge, namely addition or deletion, can require several traversals. The crux of our updating algorithm is a rooted tree based data structure to reduce the number and lengths of the traversals.

We describe in detail how the cost of traversals can be reduced by storing information about the maximum weighted edges of the paths in the graph. Because the graphs are typically very large (e.g., with over a million vertices), it is infeasible to store the maximum weighted edge of all possible paths between *all* vertex pairs, as this would require a quadratic amount of storage. Our primary algorithmic innovation is in developing efficient data structures, based on rooted trees, that require *only memory linear in the number of*

- *S. Srinivasan and S. Bhowmick are with the Department of Computer Science, University of Nebraska Omaha, Omaha, NE 68182 USA. E-mail: {sriramsrinivas, sbhowmick}@unomaha.edu.*
- *S.D. Pollard and B. Norris are with the Department of Computer and Information Science, University of Oregon, Eugene, OR 97403 USA. E-mail: {spollard, norris}@cs.uoregon.edu.*
- *S.K. Das is with the Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409 USA. E-mail: sdas@mst.edu.*

Manuscript received 30 Aug. 2017; revised 19 June 2018; accepted 31 July 2018. Date of publication 13 Sept. 2018; date of current version 14 Mar. 2022. (Corresponding author: Sriram Srinivasan.)
Recommended for acceptance by S. Ibrahim, M. Parashar, A. Queralt, and D. Talia.
Digital Object Identifier no. 10.1109/TBDATA.2018.2870136

vertices to store information about selected paths. Using this data structure, we can *limit the length of each traversal to an upper bound on the diameter* of the tree being updated.

To differentiate between operations, we will use the term "recomputing" to indicate the network property (also called feature) is computed from scratch; and the term "updating" to indicate that we incrementally update the feature from a previously computed one with the inclusion of new changes to the network.

1.2 Our Contributions

To the best of our knowledge, ours is *the first algorithm for updating MST in parallel* that can execute on actual multicore machines with scalable performance, beyond theoretical algorithm design for Parallel Random Access Machine (PRAM) models. The main contributions of our paper are as follows;

- We introduce an elegant data structure based on a rooted tree that can significantly reduce the number of graph traversals and bound the length of each traversal by the diameter of the tree.
- We use this data structure to develop shared-memory parallel algorithms to update tree-like properties with similar algorithmic structures, such as connected components and minimum weighted spanning tree. To the best of our knowledge, this is the first practical algorithm for updating MST in parallel.
- We demonstrate the efficiency of our algorithms on large-scale synthetic and real-world networks with respect to time, memory, and energy by comparing their performance with parallel algorithms for recomputing the property.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the standard sequential algorithms for finding CC and MST to demonstrate their similarity. We then propose novel parallel algorithms for updating these properties. Section 3 introduces the proposed data structure and demonstrates how the edge insertion and deletion operations are implemented over this structure. Section 4 describes parallel implementations of the algorithms and analyze their complexity. In Section 5, we prove the correctness of our algorithms. Section 6 presents experimental results on real-world and synthetic datasets and compare with parallel recomputing-based algorithms. In Section 7, we provide an overview of related works in dynamic networks. Section 8 concludes the paper with a discussion on future work.

2 Updating Connected Components and MST

This section describes our proposed parallel algorithms for computing connected components and minimum weighted spanning trees.

2.1 Graph Terminology

A network (or graph) is defined by $G = (V, E)$, where V is the set of vertices and E is the set of edges. An edge can be associated with a real number known as its *weight*. Two 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.

Given a connected undirected graph, a spanning tree is a subset of edges that form a tree and connects all the vertices in the graph. A *connected component* is a subgraph such that there exists at least one path between any pair of vertices in the subgraph. A *minimum weighted spanning tree* is a spanning tree with total weight of edges less than or equal to the weight of every other spanning tree.

2.2 Algorithm Equivalence for Computing CC and MST

The objective of the connected components algorithm is to identify vertices that are in the same component. Two vertices are in the same component if there exists a path between them.

The objective of the minimum weighted spanning tree algorithm is to construct spanning tree(s) in the graph such that the weight of the spanning tree is minimized. Note that the MST of a graph may not be unique in the sense that a graph may have several different MSTs having the same sum of the edge weights.

Algorithm 1. Extracting CC or MST

Input: Graph $G = (V, E)$
Output: Connected Components or MST $G = (V, E_x)$. E_x forms the set of key edges.

if *Extracting MST* **then**
  Sort the edges in E in the increasing order of weights
end
/* Initialize root for each vertex */
for $i \leftarrow 0$ *to* $|V|$ **do**
  $Root[i] = i$
end
/* Set List of Output Edges to Null */
$E_x \leftarrow \emptyset$
/* Check for all Edges */
for $i \leftarrow 0$ *to* $|E|$ **do**
  $this\_edge \leftarrow E[i]$
  v and u are endpoints of $this\_edge$
  /* Find the root of v */
  $root\_v = v$
  while $Root[root\_v] \neq root\_v$ **do**
    $root\_v = Root[root\_v]$
  end
  /* Find the root of u */
  $root\_u = u$
  while $Root[root\_u] \neq root\_u$ **do**
    $root\_u = Root[root\_u]$
  end
  /* If the endpoints are in separate components, join them */
  if $root\_v \neq root\_u$ **then**
    /* Add to Key Edges */
    $E_x \leftarrow E_x \cup this\_edge$
    $min \leftarrow min(root\_v, root\_u)$
    $Root[root\_v] \leftarrow min$
    $Root[root\_u] \leftarrow min$
  end
end

Algorithm 1 describes the process for finding CC and MST (via Kruskal's method. At the end of the execution, each vertex is associated with a *root* that gives a unique identifier to its component. Here the root is set to be the vertex with the lowest id in the component.

The algorithms for computing CC and MST are almost identical, except for the sorting operation to find the edges with the lowest weight for the MST computation.[1] Based on this similarity, we posit that the updating algorithms for these two cases will also be similar. The only difference lies in the technqiue how the weights are handled.

2.3 Overview of Proposed Updating Algorithms

Let us present a high level overview of our algorithms for updating CC and MST when new edges (henceforth referred to as *changed edges*) are added or deleted. We partition the edges of the graph into *key edges* and *remainder edges*. The key edges are those that were used to form the desired structure (i.e., CC or MST). As per the notations in Algorithm 1, the edges in the set E_x would be a candidate for key edges. All other edges in the graph are remainder edges, i.e., $E_r = E - E_x$.

For edge insertion, we check whether the new edge can be included in E_x. For edge deletion, the property is affected only if the deleted edge is part of E_x. If this occurs, then we seek edges from E_r that can repair the break caused by deletion. Details on these operations are as follows.

Edge Insertion. Consider an edge $e = (u, v)$ that is inserted in the graph. For both CC and MST, if there is no path between vertices u and v, then the edge is added to E_x. For MST, if there exists a path between u and v, we check whether the maximum weighted edge in the path, e_h, has a higher weight than the inserted edge, e. If so, e replaces e_h in the key edge set E_x. Otherwise, e is added to the remainder edge set.

Edge Deletion. Consider an edge $e = (u, v)$ that is deleted from the graph. We simply delete e from its respective set (key or remainder edges).

Repairing Tree. Once the insertions and deletions are completed, we first check whether the tree has become disconnected due to the deletion of key edges. If so, we further check whether any edge in the remainder edge set can be inserted into E_x to repair the tree that got disconnected due to deletion. If yes, the edge is removed from the remainder edge set and added to the key edge set.

2.4 Parallelization Strategy

We apply parallelization over the set of changed edges. For each inserted (deleted) edge, we identify in parallel whether it is to be added to (removed from) the set of tree edges or the remainder edges. Once the changed edge set is processed, we then check the edges in the remainder set in parallel to see whether these edges can be used to repair a tree that was disconnected due to edge deletion. We assume that our concurrency model supports atomic writes to variables (writes are executed without interruption) and concurrent reads.

Conflicts During Parallel Edge Insertion. All edge removals can be done in parallel and the inserted edges that belong to the remainder edge set can also be added in parallel. Moreover, for the connected components, edges can be added simultaneously to the key edge set, E_x, since the formation of a cycle does not nullify the detection of connected components.

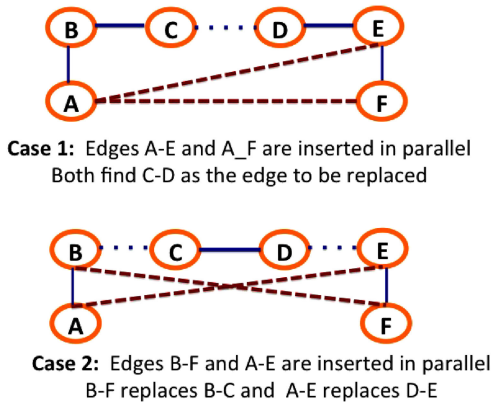

Fig. 1. Illustration of how conflicts in insertion can lead to cycles or disconnected graphs, violating the tree property.

However, inserting two edges simultaneously to the MST tree may create a cycle or, due to multiple replacements, disconnect the tree. Consider two edges (u, v) and (p, q) that are inserted simultaneously. We denote the path in the tree from nodes a to b as P_{ab}, and the maximum weighted edge in that path as e_{ab}^{max}. Let P_{uv} and P_{pq} have some common edges and their maximum weighted edges are part of this set of common edges. This situation can lead to conflicts as follows.

In the first case (Case 1), the maximum weighted edge for the endpoints of both the edges is the same, i.e., $e_{uv}^{max} = e_{pq}^{max}$. If processed in parallel, both inserted edges will remove the same edge, and get added themselves. This will create a cycle in the tree as shown in Case 1, Fig. 1.

To resolve this conflict, we mark each edge to be removed with the index of the changed edge with which it is being replaced. Thus, if a changed edge finds a maximum weighted edge that it can replace, it first checks whether that edge has already been marked to be replaced by another edge of lower weight. If so, it no longer replaces the edge.

In the second case (Case 2), there are two or more edges on the common path that have the same maximum weight, and (u, v) picks one of the possible candidate edges while (p, q) picks another. If both of these edges, occurring on the same path, are removed and replaced by the inserted edges, we can obtain a cycle and also a disconnected component, as shown in Case 2, Fig. 1.

To resolve this conflict we use a tie breaker to select a unique edge when multiple candidates are available. The tie breaker can be as simple as picking the edge with lowest vertex ids. Thus only a unique edge is selected as the maximum weighted one, and the problem reduces to the first case (Case 1) as described earlier.

3 Rooted Tree for Reducing Graph Traversals

The most frequent operation in graph algorithms is traversal. During edge insertion for updating the MST, graph traversal is used to find the path between the two endpoints of the inserted edge. Additionally, if a key edge is deleted,

1. The sorting can be done more efficiently using a heap data structure. Our purpose here is to highlight the similarities in the algorithms.

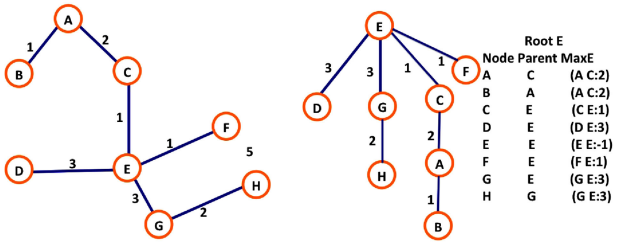

Fig. 2. Rooted tree representation. E is the root and for every vertex, its parent and maximum weighted edge on the path to the root (under column MaxE) are depicted.

for both CC and MST, graph traversal is required to identify the new roots of the disconnected components. The complexity of each traversal can be as much as $O(|V|)$, making this a very expensive operation.

We propose storing the information of the tree structures in a rooted tree as follows. We select the vertex with the highest degree from the graph to be designated as the root of the tree. Each vertex, v, stores the following information, (i) the root of the tree, r, (ii) its parent, R_v, which is the neighbor that is one level higher than it in the breadth first search (BFS) tree with r as the root, and (iii) the maximum weighted edge in its path to the root (e_{vr}^{max}). Fig. 2 gives an example how a weighted tree is stored.

3.1 Inserting a New Edge in the Tree

Consider an edge $e_{ins} = (u, v)$ that is added to the graph. Our goal is to check whether this edge should be added to the tree. For CC, this decision can be made in constant time by simply checking the component ids of the end vertices. For MST, this decision is based on the highest weighted edge, e_{uv}^{max}, on the path P_{uv} from u to v. By using the rooted tree structure, we can find the highest weighted edge in constant time in the best case and $O(h)$ time in the worst case, where h is the height of the tree, as follows. Fig. 3 illustrates the different cases.

Case 1. Vertices u and v are in two separate branches from the root. Then the path passes through the root, i.e., $P_{uv} = P_{ur} + P_{vr}$. In this case clearly, e_{uv}^{max} is the maximum of e_{ur}^{max} and e_{vr}^{max}. Since the maximum weighted edges are already stored, this operation requires $O(1)$ time complexity.

Case 2. Vertices u and v are in the same branch from the root, however, they branch to different paths at a fork vertex f. Therefore, the path from u to v passes through f, i.e., $P_{uv} = P_{uf} + P_{vf}$. This leads to two subcases as follows.

Case 2a. Consider the subcase where $e_{ur}^{max} \neq e_{vr}^{max}$. Note that the path from vertex v or u to the root has the edges on the path from the fork node, f to the root r in common. Therefore, if $e_{ur}^{max} \neq e_{vr}^{max}$, then the higher weighted of these edges have to be at or below the vertex f. Thus it will be on the path P_{uv}. Therefore, e_{uv}^{max} will be the maximum of e_{ur}^{max} and e_{vr}^{max}. This operation also takes $O(1)$ time.

Case 2b. Consider the subcase where $e_{ur}^{max} = e_{vr}^{max}$. By the same argument as above, the maximum weighted edge is on the common path from fork node f to the root r, and therefore, not part of the path from u to v.

Now, to find the maximum weighted edge, we have to traverse the path from v to f and the path from u to f. We identify the maximum weighted edge in these two paths. The number of steps to find the path would be at most $h - 1$, where h is the height of the rooted tree. Hence the time complexity of this step is $O(h)$.

From the above three cases, the maximum weighted edge can be found in the best case in $O(1)$ time, and in the worst case in $O(h)$ time.

3.2 Repairing the Disconnected Tree after Deletion

If a deleted edge (u, v) is part of the key edges, then after deletion, the tree becomes disconnected and the vertices v and u will belong to two different components (or trees). The id of the component of a vertex is required when reconnecting the disconnected trees. If we find a remainder edge where two vertices are in two different trees, then we can add the remainder edge to connect those trees.

However, reassigning the component id to vertices after each deletion is very expensive. This requires traversal of the tree to find the new root of the component. The traversal has to be executed after each deletion. Moreover, the root of one of the components will also change and thus the relations in the tree have to be re-computed. We propose below an algorithm for repairing the tree using the rooted tree data structure that can significantly reduce the traversal costs.

Given an edge $e_{del}(u, v)$ that is deleted from the tree, we first set the weight of e_{del} to a very high value, which is higher than the maximum weighted edge in the tree. Once all the deleted edges are processed, we update the maximum weighted edges for all vertices that are in the path of the end points of the deleted edges to the root. This process requires traversing the tree having complexity of $O(h)$ in the worst case . However, since we update the weights after all the deleted edges have been processed, this traversal will be done only once.

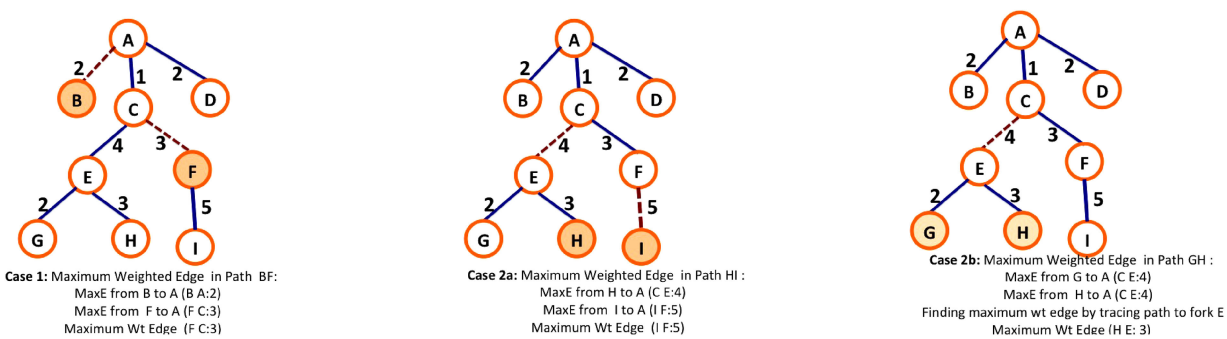

Fig. 3. Three cases by which to find the maximum weighted edge in a path. The colored nodes are the endpoints of the path. The dashed red lines are the maximum weighted edge from the nodes to the root A.

Once the maximum weighted edges are updated for the vertices affected by deletion, we go over the set of remainder edges to see if any edge from that set, $e_{rem} = (a, b)$, can be added to the tree to replace the deleted edges.

In the case of CC, the only edges that would be of higher weight are the ones marked as deleted. In the case of MST, by construction, all the edge weights would be less than or equal to e_{rem}, except the ones that were marked as deleted. Thus, once the remainder edges are processed, all the deleted edges that can be replaced, will be replaced, and the tree will be re-connected as much as possible as per the graph structure.

The process of checking whether a remainder edge can replace an edge marked as deleted is exactly the same as checking whether a new edge can be inserted (as discussed in Section 3.1). Moreover, since we are concerned with finding edges that are marked as deleted, and the deleted edges have the highest weight in the tree, Case 2b will not occur. Thus the process of checking whether a remainder edge can be added to reconnect the tree requires constant or $O(1)$ time.

4 Algorithm Implementation and Complexity

We now present the implementation details and pseudocodes of our proposed algorithms and analyze their complexities. The high level pseudocode of the complete algorithm for updating weighted trees is given in Algorithm 2.

Algorithm 2. Parallel Algorithm for Updating Connected Components or MST

```
   Input: Set E_x of Key Edges; Set E_r of Remainder Edges;
   Set CE of Changed Edges;
   Output: Updated Set E_x of Key Edges for Connected
   Components or MST.
1  Function Main (E_x, E_r, CE)
     /* Rooted Tree is an array of type RV
        of size V                                        */
2    RV RootedT[V]
3    Initialize Each Vertex in Rooted Tree
     /* Create Rooted Tree                              */
4    Create_Tree(RootedT, E_x)
     /* Status and Marked are two arrays of size CE.
        Status gives Operation on Edge. Marked
        stores Edge to be replaced                       */
5    int Status[CE]
6    Edge Marked[CE]
7    while CE is not empty do
       /* Process Changed Edges for Insertion and
          Deletion                                       */
8      Classify_Edges(CE, Status, Marked, RootedT)
       /* Process Edges as per Status                   */
9      Process_Status(E_x, E_r, CE, Status, Marked, RootedT)
10   end
     /* Repair Tree with Remainder Edges. Function
        Repair_Tree is same as Classify_Edges,
        except the edges in E_r are checked only
        whether they can be inserted.                    */
11   Repair_Tree(E_r, Status, Marked, RootedT)
     /* Process Repair Edges                            */
12   Process_Status( E_x, E_r, CE, Status, Marked, RootedT)
13   return;
```

Data Structures. We store each of the list of changed edges and remainder edges as a vector. Each changed edge is marked as to whether it will be inserted or deleted. We store the set of key edges as a rooted tree as described in Section 3. Each vertex in the rooted tree retains information about its parent, the root of the tree, and the maximum weighted edge in its path to the root. The maximum weighted edge stores the endpoints of the edges and the weight. It also contains an integer entry called *replaced id*. When the edge is marked for replacement, then the replaced id is set to the index of the edge in the changed edge set that is replacing it.

4.1 Step 1: Creating the Rooted Tree

Our first step is to create the rooted tree based on the key edges. To do so, we identify the vertex with the highest degree and designate it as the root. We execute a BFS traversal from the neighbors of this root to all the vertices reachable from these neighbors. Each BFS traversal is executed in parallel (Algorithm 3, lines 5-23). As part of the traversal, we assign the parent of the vertices and update the maximum weighted edge for each vertex based on the highest edge seen when traversing from the root to the vertex (Algorithm 3, lines 14-22).

Once all the BFSs are completed, we check the vertices to see if there are any whose parents are not assigned. This can occur if the graph has multiple components. If so, this vertex becomes the new root and the BFS traversal is executed again. This process is continued until all the vertices have assigned parents. The pseudocode for creating the rooted tree is presented in Algorithm 3.

Challenges in Parallel Rooting. Creating the rooted tree is an inherently sequential process with little opportunity for parallelism. We briefly discuss why existing algorithms on creating or rooting spanning trees, or computing BFS in parallel, are not effective for the rooted tree design that we propose.

We assume that the spanning tree is part of the input. Therefore, parallel algorithms for finding the spanning tree, such as in [9], are not applicable for creating the rooted tree.

A theoretical parallel algorithm for rooting a spanning tree using a PRAM model is given in [10]. The algorithm uses pointer jumping to efficiently find the vertices in a path. Pointer jumping is effective because this method can skip some of the edges in the path to the root. However, for the rooted tree required by our algorithm, we need to find the maximum weighted edge in the path. Thus we have to check every edge in the path from the vertex to the root and cannot leverage the benefit of pointer jumping.

In the literature, there exist parallel algorithms for BFS traversal [11], [12]. However, these algorithms are scalable only when there are sufficient branches from the root, and each branch contains a reasonably large subtree. While we have used this technique for creating the rooted tree, the scalability is not very good. This is because, unlike a graph, where there are several paths to a vertex and hence several options for branching, in a tree there is only one path through which two vertices are connected. Thus, the opportunity to create a wide BFS is severely limited.

To summarize, the process of rooting the MST is hard to parallelize due to the structure of the tree and also because we have to find the maximum weighted edge in all paths from a vertex to the root. Creating the rooted tree has time

complexity $O(V)$ and due to its inherently sequential nature, is one of the most expensive steps in our algorithm. Nevertheless, this is a one-time cost that is amortized by the reduction of traversal operations when updating the trees.

Algorithm 3. Step1: Creating the Rooted Tree

```
1  Function Create_Tree(RootedT, E_x)
     Input: The set of key edges, E_x
     Output: The rooted tree, RootedT
2    while Parents not assigned for all vertices do
       /* Find vertex r with highest degree, whose
          parent has not been assigned. Set r as
          root                                          */
3      RootedT[r].Parent ← r
4      RootedT[r].Root ← r
5      for All vertices v that are neighbors of the root
       do in parallel
         /* Set Root, Parent and maximum weighted
            edge of v                                   */
6        RootedT[v].Root ← root
         RootedT[v].Parent ← mynode
7        myE = (v, root)
8        RootedT[v].maxE ← myE
         /* Initialize Queue for BFS                   */
9        NodeQ ← ∅
10       Push v into NodeQ
11       while NodeQ ≠ ∅ do
           /* Pop front element                        */
12         Mynode ← NodeQ.top()
           /* For all neighbors                        */
13         Neigh ← neighbors of Mynode as per E_x
14         for i ∈ Neigh
15           if RootedT[i].Parent = −1 then
               /* Assign Root and Parent               */
16             RootedT[i].Root ← root
               RootedT[i].Parent ← mynode
               /* Check if myE = (i, mynode) is the
                  maximum weighted edge in the path
                  from i to the root                    */
17             if myE.edgewt < RootedT[i].maxE.edgewt
               then
18               RootedT[i].maxE ← myE
19             end
20             Push i into NodeQ
21           end
22       end
23     end
24   end
25   return;
```

4.2 Step 2: Classifying the Changed Edges

We now process the set of changed edges to determine whether they will be added (or deleted) to (or from) the tree edge or the remainder edge set. All the edges are classified in parallel and their status is marked in an array, named *Status*. The actual insertion or deletion is done in the next step (Step 3). The pseudocode for Step 2 is given in Algorithm 4.

The Status of all the changed edges is initialized to *NONE*. As part of the classification, Status can take the following values.

Deletion. We mark the Status of the edges as *DEL*, if the corresponding edge is to be deleted (Algorithm 4, lines 6–8).

Insertion without Replacement. If a changed edge is marked as inserted, and it connects vertices with different roots, then this indicates that the vertex is connecting two disconnected trees. We mark the Status as *INS* (Algorithm 4, lines 12–14).

Insertion with Replacement. When updating the MST, we also check whether a changed edge marked for insertion can replace an existing key edge. For this purpose, we first find the maximum weighted path between the endpoints of the edge to be inserted, as described in Section 3.

Let the edge to be inserted be denoted as e_{ins}, and let it replace a key edge, e_{rpl}. We check the *replaced id* variable associated with this key edge, e_{rpl}, to see whether it is already marked to be replaced by another changed edge, say e_{oth}. The edge e_{ins} will replace e_{rpl} only if it is of lower weight than e_{oth}.

In this case, the replaced id of e_{rpl} is marked by i, where i is the index of e_{ins} in the changed edge set. We maintain an array, named *Marked*, which stores the replaced edge e_{rpl} at index i, denoting that e_{ins} will replace e_{rpl}. The Status of e_{ins} is set to *RPL*, indicating that it is replacing an edge (Algorithm 4, lines 17–22).

Algorithm 4. Step 2: Classifying the Changed Edges

```
1  Function Classify_Edges (CE, RootedT, Status, Marked)
     Input: Changed Edge Set, CE; Rooted Tree, RootedT.
     Output: Status of Changed Edges, Status; Marking Edges
             to be Replaced, Marked
     /* For all edges in CE                            */
2    for i = 0 to |CE| do in parallel
       /* Initialize Status and Marked.                */
3      E ← CE[i]
4      Status[i] ← NONE
5      Marked[i] ← dummy_edge
6      if E marked as deleted then
7        Status[i] ← DEL
8      end
9      else
         /* E marked as inserted                       */
10       u ← E.node1
11       v ← E.node2
         /* If connecting disconnected
            components                                  */
12       if RootedT[u].Root ≠ RootedT[v].Root then
13         Status[i] ← INS
14       end
15       else
           /* Check if edge can be replaced            */
16         Find maximum weighted edge, MaxW, in path
           from u to v
           /* Check if E can replace MaxW              */
17         if MaxW.edgewt > E.edgewt then
18           x ← MaxW.Rpl_Id
19           if x = −1 OR
             CE[x].edgewt > E.edgewt then
20             Marked[i] ← MaxW
               Status[i] ← RPL
               MaxW.Replace_Id = i
21           end
22         end
23       end
24     end
25   end
```

4.3 Step 3: Processing Edges by Status

Once all the changed edges are classified, they are added to or removed from their respective edge sets (Algorithm 5). All the statuses except RPL simply involve adding or removing the edge from its respective edge set. When a key edge is deleted, its weight in the rooted tree is set to infinity (or a very high value) to signify that it is deleted (Algorithm 5, lines 5–13). However, replacing the edges require a few more steps as follows.

Algorithm 5. Step 3: Processing Edges By Status

1 **Function** `Process_Status` (E_x, E_r, CE, $Status$, $Marked$, $RootedT$)
  Input: Changed Edge Set, CE; Status of Edges, $Status$; Marking Edges to be Replaced, $Marked$.
  Output: Set E_x of Key Edges; Set E_R of Changed Edges; Rooted Tree, $RootedT$
2 **for** $i = 0$ *to* $|CE|$ **do in parallel**
    /* `Get Edge and its Status` */
3    $E \leftarrow CE[i]$
4    $S \leftarrow Status[i]$
    /* `Deleting Key Edge` */
5   **if** $S = DEL$ **then**
6   Delete Edge E from Appropriate Edge Set
      Set Weight of Edge E to INF, i.e., very high value to mark it as deleted
7   **end**
    /* `Edge into Remainder Edges` */
8   **if** $S = NONE$ **then**
9   Add Edge E to the Remainder Edge Set
10   **end**
    /* `Edge into Disconnected Tree` */
11   **if** $S = INS$ **then**
12   Add Edge E to the Key Edge Set
13   **end**
    /* `Replacing Edge from Key Edge` */
14   **if** $S = RPL$ **then**
      /* `Find Edge to be Replaced` */
15     $E_{rpl} \leftarrow Marked[i]$
      /* `Replacing Edge Matches Current Inserting Edge` */
16   **if** $E_{rpl}.Rpl\_Id = i$ **then**
17   Add Edge to the Key Edge Set, E_x
18   Set Weight of E_{rpl} to -1 to mark it as deleted
19   **end**
20   **else**
        /* `Replacing Edge Does Bot Match` */
21   Add E to set of new changed edges
22   **end**
23   **end**
24 **end**
25 Execute BFS on E_x from $root$, to assign the parents and maximum weighted edges in the modified $RootedT$

Let e_{new} be the edge marked with RPL. The marked index associated with e_{new} contains the edge e_{rpl}, indicating that e_{new} will replace e_{rpl} in the set of key edges. We check the index stored at the *replaced id* associated with e_{rpl} to see if it points to the index of e_{new} in the list of changed edges (Algorithm 5, lines 14–23).

If the indices do not match or the weight of e_{rpl} is set to -1, this indicates that e_{rpl} is marked to be, or has been already, replaced by a different edge, say e_{oth}. Most of the time, the weight of e_{oth} will be less than the weight of e_{rpl}, thus making e_{oth} the more suitable edge to be added. However, since we are not using locks, in some rare cases e_{oth} might have higher weight. To address this rare case, when the indices do not match, we store e_{rpl} in a new changed edge set (Algorithm 5, lines 20–22). Once Step 3 is completed, we again execute Step 2 over the new set of changed edges. If e_{rpl} has indeed a lower weight, it will be marked to be added to the key edge set. We iterate over Step 2 and Step 3 until the changed edge set is empty. The iterations will converge, because at each step, the changed edge set becomes successively smaller. In practice, we do not need to perform more than one iteration.

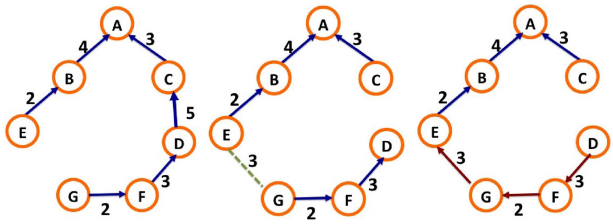

Fig. 4. Example of reversing path due to edge insertion. Each node points to its parents. Edge E-G is replacing edge C-D. The pointer to the parent is reversed from D to G.

If the indices match, then edge e_{new} is added to the set of key edges (Algorithm 5, lines 16–19). Let the endpoints of e_{new} be a and b. If a is at a higher level in the tree than b, then a is marked as a parent of b. The edge, e_{rpl}, is removed from the set of key edges. Its weight in the rooted tree is set to -1 to mark it as replaced. Let the endpoints of e_{rpl} be c and d, and d is marked as the parent of c. Since the edge e_{rpl} is deleted, we have to reset the parent information of c

To reset the parent information due to an edge being deleted, we have to find the path from c to b in the current tree, and reverse the direction of the edge to mark the parents. Consider two nodes, x and y in the path from c to b. If before replacement x was the parent of y, after replacement, y becomes the parent of x. An example is given in Fig. 4. This process of reversing the parent is very expensive (worst case complexity of $O(h)$) if done for each replacement separately. Instead, we delay the assignment of the parents of the nodes until all the edges that are marked to be replaced have been replaced. After that we reassign the parents by conducting a BFS from the root (Algorithm 5, line 25). This operation also serves to update the maximum weighted edges of the vertices that are affected by the deletion. Note that *all* changed edges that can replace e_{rpl} are marked to replace it. However, e_{rpl} stores the index of only one of the changed edges in its replaced id, guaranteeing that *exactly one* of the changed edges will replace it.

4.4 Step 4: Repairing the Tree

Once all the changed edges have been processed, we reconnect the tree if it was disconnected, using the remainder edges. For each remainder edge e_{rem}, we check whether the maximum weighted path between the two endpoints of e_{rem} is set to infinity. If so, we mark the status of the remainder edge as RPL and the *replaced id* of the edge to be replaced with the index of e_{rem}. The process is similar to the classification for edge insertion in Step 2. Note here that we

only consider the case where an edge from the remainder set can replace an edge with weight assigned to infinity. Therefore, the other Status values such as INS and DEL are not applicable here.After the remainder edges are processed, the ones marked with RPL replace the deleted edges in a process similar to the one described in Step 3.

4.5 Complexity of the Algorithm

Given a graph $G = (V, E)$, a set of changed edges with x' insertions and y' deletions, and p' processing units, the complexity of the operations are discussed below in four steps.

Step 1 (Creating the Rooted Tree): it involves a set of BFS traversals over the key edges. Since we consider spanning trees, all vertices are visited at least once. This operation has time complexity of $O(V)$, since there is little opportunity for parallelism.

Step 2 (Classifying the Changed Edges): It is performed in parallel. The complexity of marking the Status of an edge as DEL is $O(1)$, since the edge will be marked for deletion in the changed edge set. The worst case complexity of an insertion is $O(h)$, where h is the height of the tree. This is because we may have to traverse up to the root to find the maximum weighted edge. Therefore, the complexity is $(x'O(h) + y')/p'$.

Step 3 (Processing the Edges by Status): For deletion, if one endpoint of the edge marked for deletion is the parent of the other endpoint, then the edge becomes part of the key edges; otherwise, it is part of the remainder edges. This operation takes constant time. Inserting an edge in the remainder edge (when Status = $NONE$) or key edges also take constant time. Hence, if executed in parallel, the complexity of this step is $O(x' + y')/p'$. The final BFS to reassign the parents is again $O(V)$.

Step 4 (Repairing the Tree): The remainder edges are processed in parallel. The number of remainder edges is $E - E_x$, therefore processing of all the remainder edges is very expensive. Moreover, most of them do not contribute to repairing the tree. In practice, we select only a certain percentage of the edges (or those whose weight is within a threshold when updating MST). Let z be the number of remainder edges processed.

In the worst case, we have to traverse the rooted tree up to the root in order to find the maximum weighted edge in the path between the two endpoints. Thus the worst case time is $zO(h)/p$. The appropriate edges are then added to the key edges as in Step 3. The complexity of this process is $O(z/p)$.

Taking all the steps together, the total complexity of the update operations is $O(V) + (x + z)(1 + O(h))/p + 2O(y/p)$, which can written as $O(V) + O((x + z)h + y)/p)$. Therefore, to benefit from parallel processing, the number of changed set of edges must be significantly larger than the number of vertices in the original graph.

5 Correctness of Algorithms

Consider a graph $G = (V, E)$ and a set of inserted edges, E_{ins} and a set of deleted edges, E_{del}. Let the rooted tree created with the set of key edges in the original graph be denoted as RT. An active edge in RT is one that is not marked as deleted. After updating with the changed edges, following our algorithms, RT forms a new rooted tree, RT_{new}. Let $G_{new} = (V, E_{new})$ be the new graph, where $E_{new} = E \cup E_{ins} \setminus E_{del}$. Note that, if G and G_{new} have disconnected components, then RT and RT_{new} may consist of several disconnected trees, each corresponding to a connected component. Our updating algorithms will guarantee the following.

Lemma 5.1. *When updating for connected components, if there exists a path between two vertices u and v in G_{new}, then there also exists a path between these two vertices in RT_{new}.*

Proof. Since insertion and deletion of edges do not conflict, except for the rare case of lucky cancellation, we will prove the lemma separately for edge insertion and deletion.

We observe that the original rooted tree, RT, is created from the connected components in G. Therefore, at the initial step, if there is a path between any two vertices u and v in G, then there is also a path between these vertices in RT.

Edge Insertion. To prove by contradiction, we assume that there is at least one pair of vertices a and b that are connected in G_{new}, but not in RT_{new}. Given we are only considering insertion, if a and b are not connected in RT_{new}, then they are also not connected in RT and hence in the original graph G. Consequently, a and b will have different roots in RT.

Since the vertices are connected in G_{new}, this means a new edge e_{ins} has been added to G to create a path between a and b. According to our proposed algorithm, since the roots of a and b are not equal, e_{ins} will be added to the set of key edges, and subsequently be included in RT_{new}. Therefore, there will be a path from a and b in RT_{new}, thereby contradicting our assumption.

Edge Deletion. Now consider an edge, e_{del}, being deleted from the original graph G. If e_{del} was not part of RT, then after its deletion, vertices that were connected in the old graph G will still remain connected in the new graph G_{new}. Moreover, since no edge in RT was affected due to this deletion, RT_{new} is the same as RT. Given all the vertex pairs connected in G are also connected in RT, it follows that all vertex pairs connected in G_{new} are also connected in RT_{new}.

Now, let e_{del} be part of RT. Removing e_{del} disconnects RT into two components. If e_{del} is the only edge in G connecting these two components, then the paths that are deleted in creating RT_{new}, are also the paths that are deleted in G_{new}.

If there exists at least another edge that connects the disconnected components of RT_{new}, then the connections will remain intact in G_{new}. Let this alternate edge be $E_{alt} = (p, q)$. In RT_{new}, the path through the vertices p and q contains the edge E_{del},which is also the maximum weighted edge in this path. Then, based on our algorithm for repairing the tree, E_{alt} will replace E_{del}, thereby restoring the connectivity. □

Lemma 5.2. *When updating for MST, the sum of the edge weights of a MST obtained from G_{new} is equal to the sum of the weights of the active edges in RT_{new}.*

Proof. As in the proof of the previous lemma, we will prove the statement separately for edge insertion and deletion. Since the original rooted tree is created from the MST in G, therefore at the initial step, the sum of the weights of the active edges in RT is equal to the sum of the weights

of the edges in a MST obtained from G. To simplify the proof, we consider the case where there is just one MST. In other words, RT consist of only one tree.

Edge Insertion. Let e_{ins} be the new edge inserted. If e_{ins} does not replace an existing edge in RT, all the edges in RT have equal or lower weight than that of e_{ins}. Therefore, when creating the MST from G_{new}, which is G with the inserted edge e_{ins}, there exists a sorting order where all the edges in RT are processed before e_{ins}. These edges will create an MST, the same as that obtained from G. Thus RT_{new} is the same tree as RT, and the MST from G is the same as that from G_{new}.

In the case where e_{ins} replaces an edge e_{rpl} in RT, the replacement still maintains the tree structure. The replacement happens because the weight of e_{rpl} is higher than e_{ins}. Therefore, when creating the MST from G_{new}, once the edges are sorted, e_{ins} will be processed before e_{rpl}. Since all other edges remain the same, the MST will be formed using e_{ins}, and e_{rpl} will not be added to the tree. Therefore, in both the cases, the sum of the edge weights of RT_{new} is equal to the sum of the edge weights of the MST from G_{new}.

Edge Deletion. Let an edge e_{del} be deleted from the graph G. If e_{del} is not part of RT, then the edges in RT, will form a valid MST for G_{new}, and the deletion will have no effect.

If e_{del} is part of RT, then it is marked with a very large weight. Let $e_{alt} = (p, q)$ be the smallest weighted edge that is in G but not in RT, and whose endpoints are in a path that contains e_{del}. Since RT is a tree, replacing e_{del} with e_{alt} will maintain the tree structure in RT_{new}.

In the graph G_{new}, once e_{del} is removed, edges that are further in the sorted listed, i.e., of higher weight than e_{del}, have to be processed to complete the tree. The only edges that can be included in the tree are those which connect the components disconnected due to e_{del}. This is equivalent to finding edges that close the cycles in the paths containing e_{del}. Of these, the first edge to be processed would be e_{alt} (or an edge of equal weight), since this is the smallest weighted edge. Thus the weight of the new MST will be equal to the weight of the original MST having the weight of e_{del} plus the weight of e_{alt}, which is equal to the weight of the edges of RT_{new}. □

Since a graph can have multiple MSTs, our algorithm only guarantees that the updating will create one valid MST. However, the MST created by our updating algorithm is likely to have more common edges with the original MST. In practical applications of MST, such as graph clustering, this provides an extra advantage, since the number of changes in the graph structure is minimized.

6 Experimental Results

In this section, we present our experimental results for computing the connected components and MST on a dynamic network. We performed experiments on a 36-core (72 thread) Intel Haswell server with 256 GB DDR4 RAM, with two Intel Xeon E5-2699 v3 2.30 GHz CPUs. The operating system is Ubuntu 16.04. Our code is based on C++ and OpenMP and was compiled with GCC version 4.8.5.

6.1 Datasets and Implementation Specifics

We used synthetic networks generated using R-MAT model [13], which uses recursive Kronecker matrices. The model takes in as input the number of vertices in the graph, the average number of edges per vertex, and four values a, b, c, d, one for each quadrant of the matrix such that $a + b + c + d = 1$.

We generate two types of RMAT networks. The first labeled RMAT-G has scale-free degree distribution and is generated using the following parameters $a = 0.45, b = 0.15, c = 0.15, d = 0.25$. The second labeled RMAT-E is a random network with normal degree distribution and is generated using the following parameters $a = b = c = d = 0.25$. The networks have 2^{24} (RMAT24), 2^{25} (RMAT25), and 2^{26} (RMAT26) vertices respectively, with an average of 8 edges per vertex.

Our test suite also consists of three real-world social networks from the groups: YouTube (1.1M nodes, 3.0 M edges), Pokec (1.6 M nodes, 30.6 M edges), and LiveJournal (4.8 M nodes, 68.9 M edges). These networks were collected from the Stanford Network Database [14].

Since the networks were not weighted, we invoke a random number generator for each edge, and assign each edge a weight from 1 to 100. The set of changed edges are similarly weighted from 1 to 100. This step ensures that the variations in results will be only due to the structure of the network and type of changed edges, and not the distribution of the edge weights. For the repair stage, we process all the remainder edges that have a weight less than or equal to 10.

While in the parallel algorithm design, Step 3 (processing edges based on status) can be theoretically executed in parallel, the current version of OpenMP version does not support safe parallel insertion into vectors. For this reason, the implementation of Step 3 is done sequentially. However, since the operations require constant time, using a sequential algorithm does not significantly affect the scalability.

6.2 Comparing Different Classes of Networks

Fig. 5 exhibits the time and scalability results for computing MST on the synthetic and real-world networks. We updated each of these networks by inserting or deleting 50 million edges, with the percentage of insertions ranging from 100, 75 and 50 percent of the total changed edges. 50 million edges represent 37, 19, and 9 percent of the original edge counts for RMAT24, 25, and 26, respectively. For the real-world datasets 50 million edges represent 1,673 percent (YouTube), 163 percent (Pokec), and 73 percent (LiveJournal). With YouTube (3.0 M edges) for example, nearly all of the deleted and inserted edges did not exist in the original network. We show results for MST because it is the more complex operation. Results for CC (connected components) show a similar trend.

The results are scalable but they flatten out at higher scales due to the inherently sequential nature of creating the rooted trees. The percentage of insertion does not make a difference in the random networks, while in the real-world and scale-free graphs, updates due to 100 percent insertions are faster.

The degree distribution has an important influence on the time and scalability. Even though they have approximately the same number of vertices and edges, the scale-free

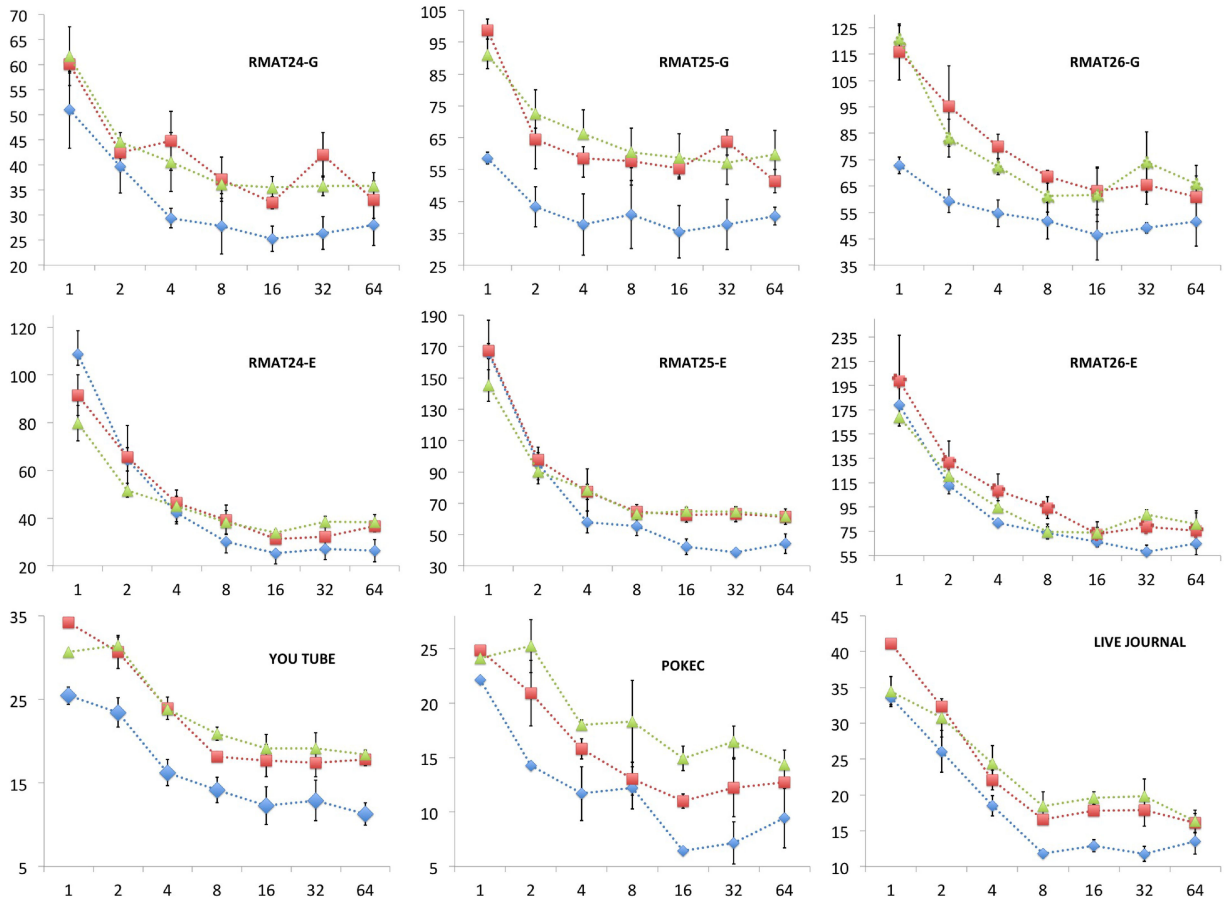

Fig. 5. Scalability results for updating networks. Top: Networks with scale-free degree distribution of order of 2^{24}, 2^{25}, 2^{26} vertices. Middle: Random Networks with normal degree distribution of order of 2^{24}, 2^{25}, 2^{26} vertices. Bottom: Real-world Networks, left to right (YouTube, Pokec, LiveJournal). Blue 100 percent insertions, Red 75 percent insertions, and Green 50 percent insertions. The X-axis gives the number of threads and Y-axis gives the time in seconds. The average time over 4 runs is plotted and error bars showing the standard deviation is given.(color online).

networks take significantly less time to update compared to networks with normalized degree distribution. We surmise that this is because the diameter of the scale-free graphs is shorter than random graphs. Therefore, given equal distribution of weights in the edges, the MST is likely to have a lower diameter, leading to lower height of the rooted-tree, which in turn reduces runtime.

We observe that among the real-world networks, even though the Pokec network is 30 times denser than the Youtube network, the timings are very similar. This observation indicates that the execution time of our updating algorithm is relatively independent of the density of the network.

6.3 Effect of Selection of Root on Time

Because the complexity of insertion and deletion of an edge is proportional to the height of the tree, we investigated how the height of the tree affects the time for updates. For one synthetic network (RMAT24-G) and one real-world network (LiveJournal), we selected the root that gives the tree with the shortest height, the root with the tree that gives the greatest height, and six other heights in between. As can be seen in Fig. 6, while there is some variation of time when the number of processors is small, the variations become negligible as the number of processors increases. We also observe that the difference between the shortest and the tallest height is not very large and hence does not make a significant difference in the updating time.

6.4 Comparison with Recomputing Algorithm

We compare the time taken to update the CC and MST with the time taken to recompute them from scratch using the Galois software [15]. Galois has a template that identifies lower level parallelism, such as loops, in the graph algorithms. By simply parallelizing the loops, Galois has been shown to be very fast compared to other parallel network packages. Galois has a parallel algorithm for finding CC and uses a parallel Boruvka algorithm [15] for computing the MST. Since the speedup and memory performance results exhibit similar trends for both algorithms, we focus on the MST in this section.

Parallel Speedup. Parallel speedup is defined as the ratio of sequential to parallel execution time, i.e., T_1/T'_p where p' is the number of threads. In Fig. 7, we show the speedup values for RMAT24-G and three real networks with one million changed edges and 100 percent insertions (top row) and 75 percent insertions (bottom row). The ideal speedup is equal to the number of threads. The speedups for both of our algorithm and Galois degrade as the number of threads increases, but our updating method degrades faster. Unsurprisingly, the speedup for both methods is better for denser networks, which require more computation than sparser ones such as Youtube, because there is more work overall. Improving the speedup of dynamic algorithms is ne of our future objectives.

The speedup does not consider the rooting tree computation because it is a one-time cost that is later amortized by

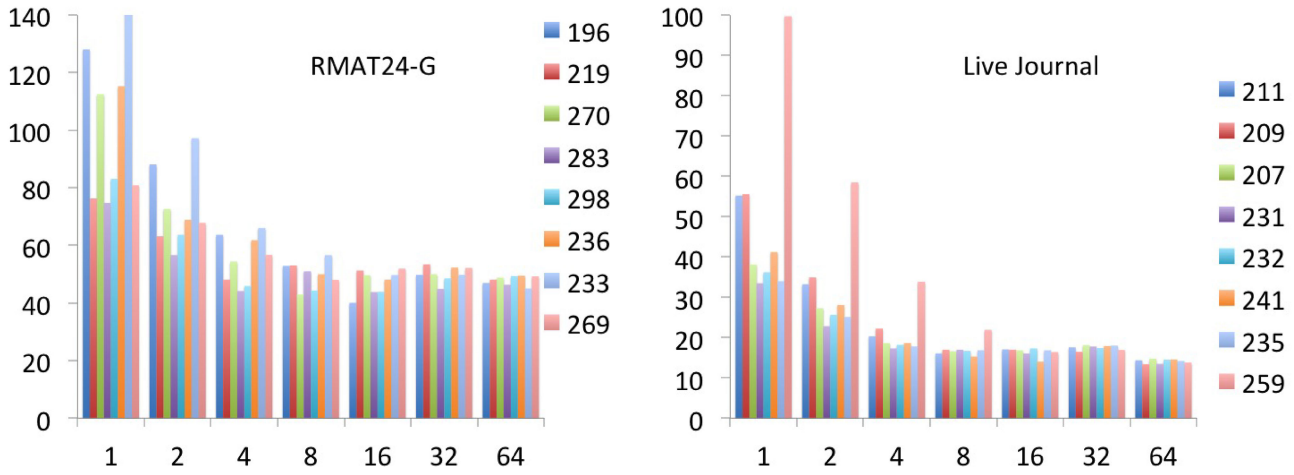

Fig. 6. Variations of updating time based on the choice of root. Left: RMAT24-G. Right: LiveJournal. The Y-axis gives the time in seconds. The X-axis gives the number of threads. Each colored bar represents a tree of different height. The heights are given in the legend. The results are for 100 percent insertions of 50M edges. While there are some variations in time for lower processors, the times are almost equivalent as the number of processors increase.

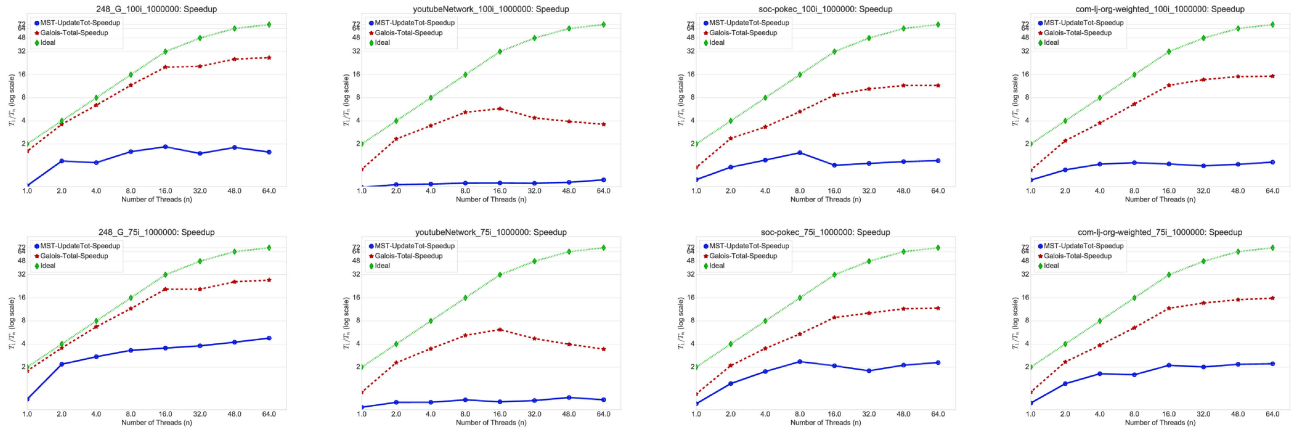

Fig. 7. Parallel speedup (log scale) for computing the MST of RMAT-24G and three real-world networks (described in Sec 6.1). Speedup was computed based on one batch update, with batch size 1,000,000 edges. Top: 100 percent insertions. Bottom: 75 percent insertions.

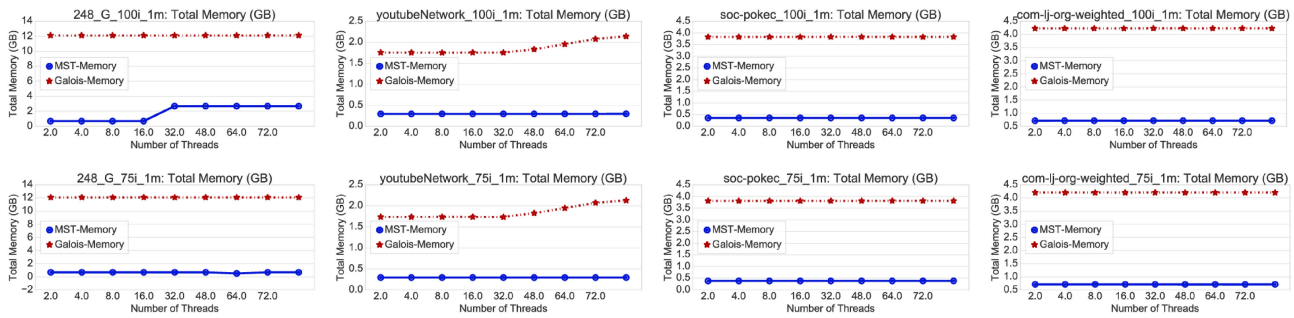

Fig. 8. Total memory use for computing the minimum spanning tree of RMAT-24G and three real-world networks (described in Sec 6.1). Top: 100 percent insertions. Bottom: 75 percent insertions.

the cost of multiple updates. We do show the breakdown of the times (in seconds) for the different phases in Table 1. The first column indicates the network (among the datasets described in Section 6.1, while the second column specifies the percentage of insertions in the update (batch size is 1,000,000 edges). The third column indicates the number of threads in the experiment, and the MST Root, MST Ins-Del and MST Repair columns show average times for the rooting, insertion/deletion, and tree repair phases of the algorithm. The MST Update column is the sum time for insertion, deletion, and repair of the tree (after deletions). The Galois Boruvka time includes the full tree computation, excluding I/O.

Memory. In Fig. 8, we show the total memory used by our algorithm and Galois' Boruvka for RMAT24-G for three real networks with a million changed edges and 100 percent insertions (top row) and 75 percent insertions (bottom row). Note that this value is the maximum resident memory size for the entire computation, which includes initialization and in the case of dynamic algorithm, it includes the initial

TABLE 1
Execution Times (in Seconds) for Different Phases of Our Update Algorithm and Galois' Recomputation

Network	% Ins.	Num. Thr.	MST Root	MST Ins-Del	MST Repair	MST Update	Galois Total
RMAT24-G	100%	1	7.46	0.97	0	0.97	101.92
RMAT24-G	100%	2	9.15	0.95	0	0.95	55.71
RMAT24-G	100%	4	7.71	0.6	0	0.6	28.24
RMAT24-G	100%	8	7.53	0.62	0	0.62	15.94
RMAT24-G	100%	16	7.24	0.53	0	0.53	8.72
RMAT24-G	100%	32	7.4	0.5	0	0.5	5.07
RMAT24-G	100%	48	7.54	0.54	0	0.54	4.96
RMAT24-G	100%	64	7.38	0.5	0	0.5	3.99
RMAT24-G	100%	72	7.36	0.53	0	0.53	3.83
RMAT24-G	75%	1	7.39	0.85	5.58	6.43	103.78
RMAT24-G	75%	2	9.54	0.8	4.36	5.16	54.28
RMAT24-G	75%	4	9.39	0.57	2.38	2.95	29.19
RMAT24-G	75%	8	7.89	0.6	1.76	2.35	15.47
RMAT24-G	75%	16	10.35	0.6	1.34	1.94	8.94
RMAT24-G	75%	32	7.06	0.47	1.35	1.82	5.00
RMAT24-G	75%	48	7.24	0.48	1.22	1.7	5.01
RMAT24-G	75%	64	7.91	0.46	1.06	1.52	4.01
RMAT24-G	75%	72	8.61	0.44	0.91	1.35	3.81
LiveJournal	100%	1	1.64	1.05	0	1.05	14.03
LiveJournal	100%	2	1.74	0.91	0	0.91	10.11
LiveJournal	100%	4	1.72	0.75	0	0.75	6.37
LiveJournal	100%	8	1.71	0.69	0	0.69	3.75
LiveJournal	100%	16	1.75	0.67	0	0.67	2.11
LiveJournal	100%	32	1.86	0.68	0	0.68	1.21
LiveJournal	100%	48	1.89	0.7	0	0.7	1.02
LiveJournal	100%	64	1.82	0.69	0	0.69	0.93
LiveJournal	100%	72	1.93	0.67	0	0.67	0.92
LiveJournal	75%	1	1.64	0.91	1.38	2.3	14.11
LiveJournal	75%	2	1.77	0.83	1.16	1.99	9.98
LiveJournal	75%	4	1.87	0.68	0.75	1.43	6.04
LiveJournal	75%	8	1.87	0.65	0.6	1.24	3.66
LiveJournal	75%	16	1.74	0.68	0.58	1.26	2.16
LiveJournal	75%	32	1.98	0.6	0.48	1.09	1.21
LiveJournal	75%	48	2.04	0.66	0.48	1.14	1.03
LiveJournal	75%	64	1.98	0.64	0.41	1.05	0.93
LiveJournal	75%	72	1.86	0.62	0.41	1.04	0.89
Pokec	100%	1	0.69	0.76	0	0.76	7.77
Pokec	100%	2	0.72	0.66	0	0.66	5.36
Pokec	100%	4	0.73	0.52	0	0.52	3.29
Pokec	100%	8	0.74	0.47	0	0.47	2.33
Pokec	100%	16	0.73	0.42	0	0.42	1.47
Pokec	100%	32	0.78	0.51	0	0.51	0.89
Pokec	100%	48	0.66	0.49	0	0.49	0.74
Pokec	100%	64	0.75	0.48	0	0.48	0.67
Pokec	100%	72	0.77	0.47	0	0.47	0.67
Pokec	75%	1	0.68	0.68	1.5	2.18	7.65
Pokec	75%	2	0.72	0.63	1.29	1.92	5.59
Pokec	75%	4	0.72	0.52	0.84	1.35	3.65
Pokec	75%	8	0.75	0.46	0.69	1.15	2.18
Pokec	75%	16	0.7	0.44	0.49	0.93	1.42
Pokec	75%	32	0.7	0.53	0.52	1.05	0.86
Pokec	75%	48	0.74	0.5	0.64	1.14	0.76
Pokec	75%	64	0.77	0.46	0.57	1.03	0.67
Pokec	75%	72	0.75	0.49	0.47	0.95	0.65
YouTube	100%	1	0.37	2.31	0	2.31	1.69
YouTube	100%	2	0.43	2.35	0	2.35	1.21
YouTube	100%	4	0.38	2.21	0	2.21	0.73
YouTube	100%	8	0.37	2.18	0	2.18	0.49
YouTube	100%	16	0.38	2.14	0	2.14	0.33
YouTube	100%	32	0.36	2.13	0	2.13	0.29
YouTube	100%	48	0.35	2.14	0	2.14	0.39
YouTube	100%	64	0.4	2.1	0	2.1	0.43
YouTube	100%	72	0.37	2	0	2	0.47
YouTube	75%	1	0.38	2.22	0.3	2.51	1.62
YouTube	75%	2	0.37	2.11	0.26	2.38	1.15
YouTube	75%	4	0.38	1.97	0.17	2.14	0.71
YouTube	75%	8	0.38	2	0.14	2.14	0.47
YouTube	75%	16	0.36	1.93	0.11	2.04	0.31
YouTube	75%	32	0.36	1.99	0.12	2.12	0.26
YouTube	75%	48	0.38	1.95	0.12	2.07	0.34
YouTube	75%	64	0.4	1.83	0.12	1.95	0.41
YouTube	75%	72	0.38	1.92	0.12	2.03	0.47

MST computation, as well as the update (a single batch size of 1,000,000 edges). For the networks we tested, the memory footprint of the recomputing (Galois) algorithm uses up to 24x the amount of memory used by the update algorithm.

Power and Energy. We now compare the power and energy consumption of our code with the Boruvka algorithm in Galois. We use the Performance Application Programming Interface (PAPI) [16]. PAPI interfaces with Intel's Running Average Power Limit (RAPL), which provides access to a set of hardware counters measuring energy usage. From these, PAPI computes average energy in nanojoules for a given time interval. We performed power and energy measurement experiments on a dual socket Intel® Xeon® E5-2699 v3 @ 2.30 GHz chips (36 hardware threads each, 72 total). These support direct per-socket power measurement (unlike earlier generations where power and energy values were estimated based on performance counters). We instrumented the source code to measure only the actual update-related computation.

We considered three real-world networks and the RMAT-24 G synthetic network (described in Section 6.1) to evaluate the power and energy scaling of our algorithm and compare them to the best available parallel shared-memory implementations of the equivalent recomputing algorithm from Galois. For the dynamic updating algorithm, we show measurements for Steps 2 and 3 (termed insertion and deletion) and Step 4 (termed repair, occurs only for deletion), which constitute a single batch update. The batch size of each update is 1,000,000 edges. For the recomputing algorithm from Galois, we show the total time, power and energy measurements when applying the algorithm (not considering I/O) to each updated network. Each measurement corresponds to the mean of four identical experiments performed in the dedicated mode. All experiments were also performed in the same time period (to minimize influence of environmental conditions such as ambient temperature differences). As a baseline for power measurements, we also include the overall machine power measured during a system call to `sleep(10);` we took the minimum from testing with various numbers of threads, which was 25 W.

Fig. 9 shows the power and energy consumption of the dynamic minimum spanning tree algorithm applied to RMAT-24G for different percentages of insertions and similar measurements for the Boruvka algorithm in Galois. The patterns for both MST and CC computations are similar, hence we focus on the MST. The power and energy measured with RAPL [17] are shown per CPU socket, each socket is indicated by the "-p0" or "-p1" suffix in the plot legends. The *MST-UpdateTot* values include the total for both CPUs for the MST-ins-del and MST-repair operations (which constitute a full update). Because it is a one-time cost which will be amortized with multiple updates, the rooting tree phase is not included in this total (instead, we summarize the times for different phases in Table 1). For Galois, *Galois-Total* includes the complete algorithm totals for both CPUs. We also include the execution time along with the corresponding power and energy data for easy reference, even though some of that information is also available in the scaling studies described in Section 6.4. When there are no deletions (first row in Fig. 9), the dynamic algorithm has significantly lower power requirements than Galois on more than two threads, and the power for increasing the number of threads remains constant, whereas it grows rapidly with Galois. Deletions, and in particular, repairing the tree, require more power than insertions, as

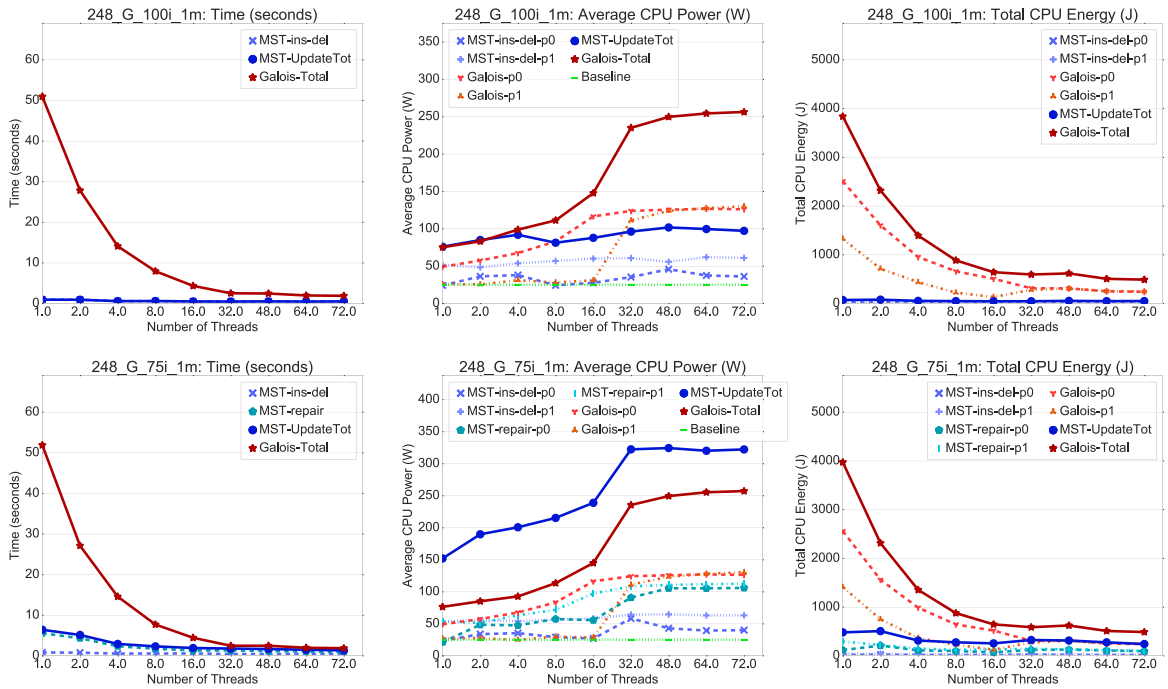

Fig. 9. Comparison of per-socket and total power and energy measurements: a single batch update (blues), broken into "Insertion and Deletion and "Repair," compared with recomputing using Galois (reds) of the minimum weighted spanning tree. Top: 100 percent insertions Bottom: 75 percent insertions. Left: runtime. Center: power measurements. Right: energy measurements.

evident fromm the experimental data in the second row in Fig. 9 and even rows of Fig. 10. Galois computations require significantly more energy than the updating algorithm, indicating that updating using a rooted tree is more energy-efficient for this type of network.

Fig. 10 shows the time, power, and energy characteristics (similar to Fig. 9) for the real-world networks described in Section 6.1. Focusing on the power measurements (middle column), one common pattern across all the networks is that Galois' power consumption is relatively low for small thread counts, then increases sharply when using more than 16 threads on both CPU sockets. For 16 or fewer threads, clearly just one of the CPUs is being used, but the increase in power is not just in the second CPU (bad power scaling), which can present problems in power-capped environments. In contrast, the power consumption of edge insertion (rows 1, 3, and 5 of Fig. 10) in the update algorithm remains relatively flat as more threads are used. For insertion-only updates, with lower thread counts($n < 16$) on few real-world networks, Galois consumes less power than the update algorithm. If the number of thread increases, Galois consumes significantly more power than the update algorithm for insertion updates. While Galois is faster at larger thread counts, its higher power use leads to higher energy consumption for two of the three real-world networks, LiveJournal and Pokec (with 100 percent insertions). Again, in the presence of deletions, the repair computation makes the update algorithm less power-efficient than the recomputation. Identifying the insertion percentages for which the update is better in terms of time, power, and energy is a topic of future research.

6.5 Summary of Comparisons

From the experiments, we observe that our proposed algorithm requires significantly less memory than Galois. While part of this depends on the internal data structures of Galois, we note that our algorithm requires only an extra $O(V)$ storage in addition to the graph and the set of changed edges. This makes our algorithm very lightweight.

The experiments also show that our updating algorithm is faster and uses less energy than Galois. Furthermore, it generally uses less power and has flatter power profiles at all levels of parallelism, making it better suited for power-capped environments than Galois. Even at higher number of threads, where the scalability is worse, we are comparable to Galois. The reason for the improved performance is that Galois has to recompute the entire MST or connected components after including the changes in the graph. However, most of the changes do not affect the structure of the tree. Our updating algorithm can identify these edges and does not include them in updating the tree structure. Moreover, by storing the information on maximum edge weights in the rooted tree data structure, we can update the tree very quickly, in the best case in constant time and in the worst case in $O(h)$ time.

Galois has better speedup than our algorithm. This is mainly due to the fact that the update performs significantly less work than Galois algorithm, limiting effective parallelization to fewer threads. However, as our results show, the speedup does not necessarily mean that the time taken is actually less. Galois algorithm is faster than ours for one of the networks we tested even for 100 percent insertions, and faster at higher thread counts for 75 percent insertions. For high-deletion scenarios, the recomputation algorithm in Galois may be more appropriate because the cost of repairs exceeds the benefit of not recomputing the tree from scratch.

7 Related Research

Here we discuss related research in parallel dynamic graph algorithms and in analyzing their power requirements.

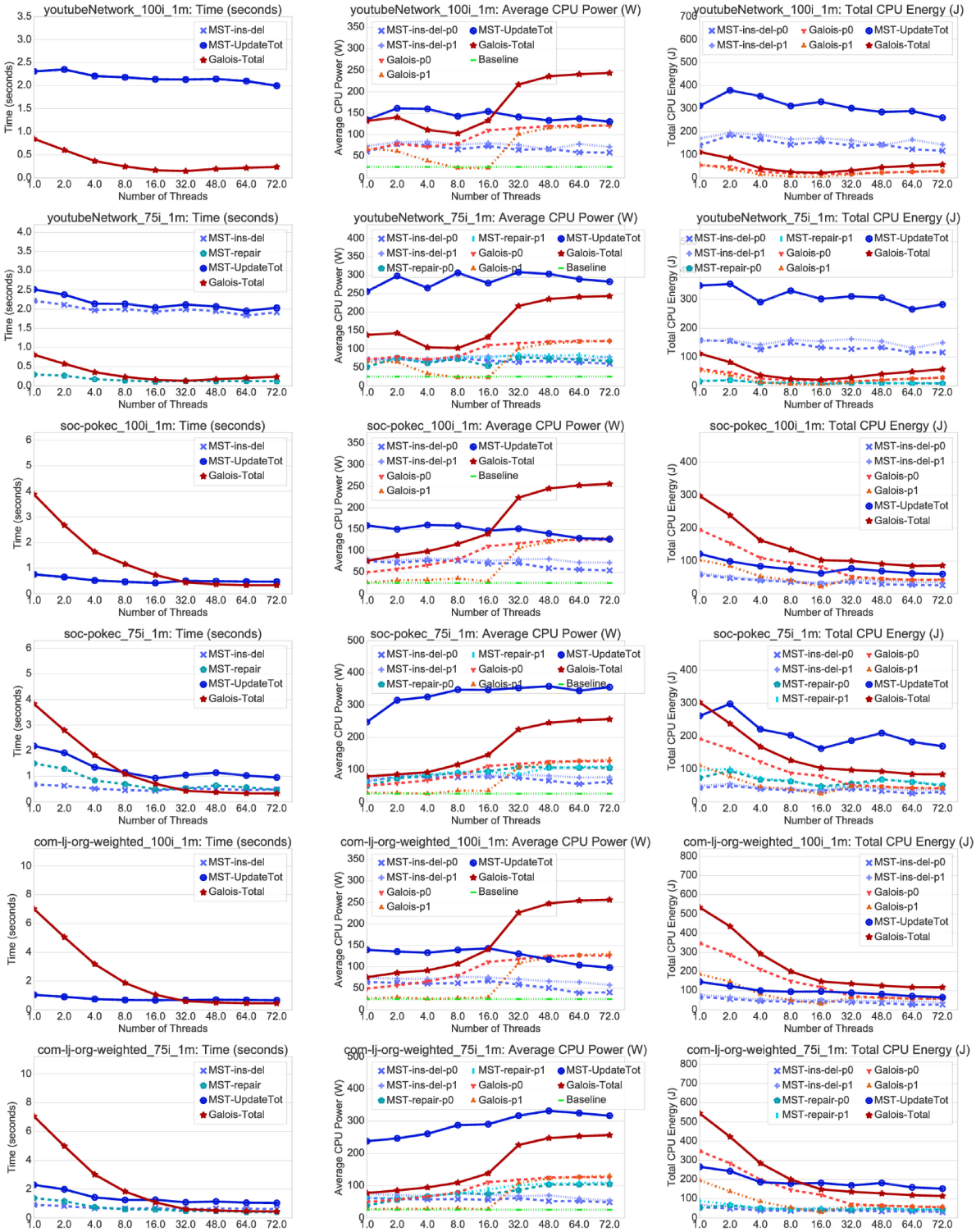

Fig. 10. Real-world network comparison of per-socket and total power measurements: a single batch update (blues), broken into "Insertion and Deletion" and "Repair," compared with recomputing using Galois (reds) of the minimum weight spanning tree. Top two rows: YouTube (1.1 M nodes, 3.0 M edges). Middle two rows: Pokec (1.6 M nodes, 30.6 M edges). Bottom two rows: LiveJournal (4.8 M nodes, 68.9 M edges). For each network, the first and second rows represent 100 and 75 percent insertions, respectively.

7.1 Algorithms on Dynamic and Parallel Graphs

Several interesting solutions have been proposed for developing parallel algorithms for large networks. They include: (1) *Matrix-based graph approach*, where the network algorithms are expressed as sparse matrix operations [18] to make them easier to parallelize on distributed memory systems. (2) *Vertex centric graph algorithms* used by GraphLab [19] and Pregel [20], where each vertex is updated locally, and then a synchronization function is used for global updates. The algorithms then converge over several iterations. (3) *Massively multithreaded graph algorithms* use fine-grained parallelism to generate enough tasks to keep the processing units busy while they are waiting for data [21], [22]. This approach is suited for platforms with very large numbers of processing

units such as massively multithreaded machines. These methods, with the exception of the multithreaded approach, apply only to static networks. Two projects that on dynamic graph algorithms are STINGER [22] and PHISH [23]. However, they deal with only unweighted graphs.

Parallel algorithms related to spanning tree, such as breadth first search and minimum weighted spanning trees, on static networks are available for many HPC platforms including distributed memory [11], multicores [24], massively multithreaded machines [21], and GPUs [12]. Parallel algorithms for dynamically updating connected components have been developed using the STINGER framework [25] and in our previous work using a sparsification-based technique [26].

Parallel implementations of MST include the Shiloach-Vishkin approach [27] and Boruvka's algorithm [28]. Sequential algorithms for updating MST are discussed in [29]. Parallel algorithms for dynamic MST were proposed in [30], [31] for theoretical PRAM machines but no empirical results were reported. To the best of our knowledge, our algorithm is the first practical parallel implementation that can be executed on multicores for very large networks.

7.2 Power and Energy of Static Graph Computations

The Green Graph 500 list [32] collects performance-per-watt metrics and acts as a forum for vendors and data center operators to compare the energy consumption of data intensive computational workloads on their architectures. The benchmark consists of undirected graph generation and breadth-first search. In [33], we explored fine-grained power and energy modeling of the BFS benchmark using custom hardware to provide component-level high-frequency power and energy measurements.

Other approaches to evaluating power and energy of graph algorithms (e.g., Graphicionado [34]), typically focus on special purpose hardware for graph computations. Graph algorithms research that targets execution time improvement also indirectly leads to reduced energy, but most works in that area do not explicitly analyze the power use of these algorithms.

Power and energy can be estimated based on hardware performance counters (e.g., [35]) or direct measurements, either supported by commodity hardware such as recent generations (Haswell) of Intel server processors, or through additional hardware instrumentation. In this work, we rely on the Intel Running Average Power Limit [17] for obtaining direct power measurements.

8 Conclusion

In this paper we presented a template for designing parallel algorithms for updating tree-like properties of dynamic networks. Our algorithm is primarily based on storing information in a rooted tree, using which the graph traversal time is significantly reduced. Although creating the rooted tree has little opportunity for parallelism, our experiments on large-scale real-world and synthetic networks demonstrate that our updating algorithm is faster and requires less energy and memory than state-of-th-art parallel algorithms for recomputing the graph properties.

It is also observed that edge deletions require more resources than edge insertions. This is because re-joining the tree requires searching through all the remainder edges and is often equivalent to an intermediate phase in the static algorithm. This is a typical situation for any dynamic system, where the effect of change can become so large that recomputing is more efficient than updating. As part of our future work, we aim to determine this critical point for different dynamic algorithms. Another solution to finding the critical edges is to find the betweenness centrality of the remainder edges and process them accordingly.

In the future we aim to investigate heuristics to help reduce the cost of these operations. We plan to explore faster heuristics for creating the tree as needed (e.g., lazy creation). We will also explore the design of adaptive algorithms, where one can switch between updating and recomputation at runtime based on the type and percentage of changed edges and the computational resource to be optimized. Finally, we plan to develop parallel algorithms for updating other properties of weighted graphs, such as the single source shortest path, using the rooted tree approach.

Acknowledgments

The authors would like to thank the anonymous reviewers and guest editors for insightful comments and constructive suggestions.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 Samuel Pollard are supported by the NSF CCF Award #1725585.

References

[1] B. H. Junker and F. Schreiber, *Analysis of Biological Networks (Wiley Series in Bioinformatics)*. New York, NY, USA: Wiley, 2008.

[2] E. Stattner and N. Vidot, "Social network analysis in epidemiology: Current trends and perspectives," in *Proc. 5th Int. Conf. Res. Challenges Inf. Sci.*, May 2011, pp. 1–11.

[3] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Conf. Internet Meas.*, 2007, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298311

[4] H. Jeong, S. P. Mason, A. L. Barabasi, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, no. 6833, pp. 41–42, May 2001. [Online]. Available: http://dx.doi.org/10.1038/35075138

[5] P. H. A. Sneath, "The application of computers to taxonomy," *Microbiology*, vol. 17, no. 1, pp. 201–226, 1957. [Online]. Available: http://mic.microbiologyresearch.org/content/journal/micro/10.1099/00221 287–17-1-201

[6] Y. K. Dalal and R. M. Metcalfe, "Reverse path forwarding of broadcast packets," *Commun. ACM*, vol. 21, no. 12, pp. 1040–1048, Dec. 1978. [Online]. Available: http://doi.acm.org/10.1145/359657.359665

[7] B. Ma, A. Hero, J. Gorman, and O. Michel, "Image registration with minimum spanning tree algorithm," in *Proc. Int. Conf. Image Process.*, 2000, pp. 481–484.

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

[9] D. A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs)," *J. Parallel Distrib. Comput.*, vol. 65, no. 9, pp. 994–1006, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731505000882

[10] G. Cong and D. A. Bader, "The euler tour technique and parallel rooted spanning tree," in *Proc. Int. Conf. Parallel Process.*, Aug. 2004, pp. 448–457.

[11] A. Yoo, E. Chow, K. Henderson, W. Mclendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *Proc. ACM/IEEE Conf. Supercomput.*, 2005, Art. no. 25.

[12] L. Luo, M. Wong, and W.-M. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. 47th Des. Autom. Conf.*, 2010, pp. 52–55. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837289

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 2–3.
[14] J. Leskovec, "Stanford large network dataset collection," (2014, Jun.). [Online]. Available: http://snap.stanford.edu/data/
[15] 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 Notices*, vol. 46, no. 6, pp. 12–25, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993501
[16] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Proc. Tools High Perform. Comput.*, 2010, pp. 157–173.
[17] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *Proc. ACM/IEEE Int. Symp. Low-Power Electron. Des.*, Aug. 2010, pp. 189–194.
[18] Knowledge discovery toolkit, 2014. [Online]. Available: http://kdt.sourceforge.net/
[19] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212354
[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184
[21] 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.
[22] Stinger-streaming graph analytics, 2015. [Online]. Available: http://www.stingergraph.com/
[23] J. Plimpton and T. Shead, "Streaming data analytics via message passing with application to graph algorithms," (2014). [Online]. Available: http://www.sandia.gov/ sjplimp/phish/papers.html
[24] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.46
[25] R. McColl, O. Green, and D. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *Proc. IEEE Int. Conf. High Perform. Comput.*, 2013, pp. 246–255.
[26] S. Srinivasan, S. Bhowmick, and S. Das, "Application of graph sparsification in developing parallel algorithms for updating connected components," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2016, pp. 885–891.
[27] D. A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs)," *J. Parallel Distrib. Comput.*, vol. 65, no. 9, pp. 994–1006, Sep. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2005.03.011
[28] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the GPU," in *Proc. Conf. High Perform. Graph.*, 2009, pp. 167–171. [Online]. Available: http://doi.acm.org/10.1145/1572769.1572796
[29] G. Cattaneo, P. Faruolo, U. F. Petrillo, and G. Italiano, "Maintaining dynamic minimum spanning trees: An experimental study," *Discr. Appl. Math.*, vol. 158, no. 5, pp. 404–425, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X09003928
[30] D. Eppstein, Z. Galil, and G. F. Italiano, "Dynamic graph algorithms," *Algorithms and Theory of Computation Handbook*, M. J. Atallah, Ed., CRC Press, 1999, ch. 8.
[31] S. Das and P. Ferragina, "An erew pram algorithm for updating minimum spanning trees," vol. 9, pp. 111–122, 1999.
[32] T. Hoefler, "Green Graph 500," (2012, Jul.). [Online]. Available: http://green.graph500.org/
[33] M. Rashti, G. Sabin, and B. Norris, "Power and energy analysis and modeling of high performance computing systems using WattProf," in *Proc. IEEE Nat. Aerosp. Electron. Conf.*, Jul. 2015, pp. 367–373.
[34] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2016, pp. 1–13.
[35] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, "A study on the use of performance counters to estimate power in microprocessors," *IEEE Trans. Circuits Syst. II: Express Briefs*, vol. 60, no. 12, pp. 882–886, Dec. 2013.

Sriram Srinivasan received the MS degree in computer science from the University of Nebraska and is currently working toward the PhD degree His current interests include HPC, dynamic graphs and machine learning.

Samuel D. Pollard received the BS degree in mathematics and the MS degree in computer science from Western Washington University, in 2014 and 2016, respectively. He is working toward the PhD degree at the University of Oregon with a focus on HPC. He has also researched job scheduling at Lawrence Livermore National Lab and formal methods at Sandia National Lab. His research interests include performance analysis, program transformation, parallel programming models, and numerical analysis.

Boyana Norris received the BS degree in computer science from Wake Forest University, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. She is an associate professor with the Computer and Information Science Department, University of Oregon. Her research in high-performance computing focuses on methodologies and tools for performance reasoning and automated optimization of scientific applications, while ensuring continued or better usability of HPC tools and libraries and improving developer productivity. She also works on adaptive parallel algorithms in numerical linear algebra and graph processing.

Sajal K. Das [F'15] is a professor of computer science and Daniel St. Clair Endowed chair at the Missouri University of Science and Technology, Rolla. His research interests include wireless sensor networks, cyber-physical systems and IoT, parallel/distributed and cloud computing, big data analytics, cyber security, biological and social networks. He is a recipient of 10 Best Paper Awards and numerous awards for teaching, mentoring and research including the IEEE Computer Society's Technical Achievement Award for pioneering contributions to sensor networks and mobile computing. He serves as the founding editor-in-chief of the Elsevier's *Pervasive and Mobile Computing Journal*, and as associate editor of several journals including the *IEEE Transactions of Mobile Computing* and the *ACM Transactions on Sensor Networks*. He is a fellow of the IEEE.

Sanjukta Bhowmick received the BTech degree in CSE from the Haldia Institute of Technology and the PhD degree from the Pennsylvania State University. She has worked on developing efficient solvers for large linear using machine learning and on automatic differentiation. Her current research is on analyzing large scale graph on different parallel architectures and studying the effect of dynamicity and noise on these graphs.