Algorithms for Updating Dynamic Networks

By

Sriram Srinivasan

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Dr. Sanjukta Bhowmick

Omaha, Nebraska

March, 2019

Supervisory Committee:

Dr. Sanjukta Bhowmick

Dr. Yuliya Lierler

Dr. Kathryn Cooper

Dr. Mahantesh Halappanavar

Algorithms for Updating Dynamic Networks

Sriram Srinivasan, (Ph.D.)

University of Nebraska, 2019

Advisor: Dr. Sanjukta Bhowmick

The growth of social media increased the interest of analyzing network algorithms. The networks are highly unstructured and exhibit poor locality, which has been a challenge for developing scalable parallel algorithms. The state-of-the-art network algorithms such as Prim's algorithm for Minimum Spanning Tree, Dijkstra's algorithm for Single Source Shortest Path, and iSpan algorithm for detecting strongly connected components are designed and optimized for static networks. The networks that change with time i.e. the dynamic networks (such as social networks) the above-mentioned approaches can only be utilized if they are reimplemented from scratch each time. Performing a re-computation from scratch for a significant amount of changes is not only computationally expensive, however, increases the memory footprint and the execution time. In the case of dynamic networks, developing scalable parallel algorithms is very challenging and there has been a very limited amount of research work that has been performed when compared to developing parallel scalable algorithms for static networks.

To address the above challenges, this Ph.D. dissertation proposes a new high performance, scalable, portable, open source software package and an efficient network data structure to update the dynamic networks on the fly. This approach is different from the naive approach which is the re-computation from scratch and is scalable for random, small-world, scale-free, real-world and synthetic networks. The software package currently is implemented on a shared memory system and updates network properties such as Connected Components (CC), Minimum Spanning Tree (MST), Single Source Shortest Path (SSSP), and Strongly Connected Components(SCC). The key attributes of the software are faster insertions and deletions. Additionally, the software takes less time and memory for updating the networks when compared to the state of the art Galois. The shared memory implementation processes over 50 million updates on a real-world network under 30 seconds. The dissertation concludes with a summarization of the contributions and their improvement on large-scale network analytics, and a discussion about future work on this field.

Contents

List of Figures and Tables

List of Equations

Chapter 1

Introduction

Networks are ubiquitous and are widely used in various application domains to represent objects and their relationships such as a biological, citation and social network. In mathematics, networks are represented as $G = (V, E)$ where V represents the node and E represents the various relationships of V. For example, in Figure 1.1 is a simple representation of a citation network, where the vertices represent the research paper and the edges represent the relationship or citations. Network models are gaining a lot of attention in the last few years as it can provide valuable insight information about large scale systems. In a conventional relational database, performing a join operation is pretty expensive on multiple tables, however, storing data in the form of networks can help to perform traversal and massive queries easily with less computational cost. The citation network is an example of how the graph can intuitively represent relations in large scale applications.

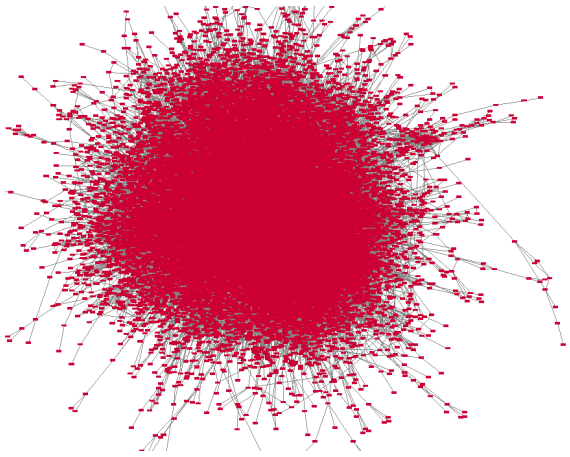

Figure 1.1: Citation Network

Networks can be static or dynamic in nature. Most of the real-world networks such as social or biological networks are dynamic in nature, i.e. there are always changes in the network such as addition/deletion of vertices or edges. Figure 1.2 is an example of a simple dynamic network which changes with time. As the network changes, there are a lot of questions that arise to any domain specialist such as, is the network strongly connected? or what is the optimal path to traverse the entire network? Analyzing the changes faster has many benefits such as administering and giving appropriate responses. For example, if there is a serious epidemic in a county, then as the epidemic grows, vaccinating the highly connected vertices in the network can slow down the spread of the epidemic. Most of these complex systems change with time, accordingly their properties need to be updated. For example, MST and chordal subgraphs are used to identify in gene expression networks. Computing the properties of any network is always considered to be expensive due to their size and unstructured nature. In the case of the dynamic network, the cost will increase if we constantly recompute the properties.

In the case of a static network, there are many highly optimized algorithms and architectures which answer the above questions easily and efficiently. However, when the networks are dynamic, problems become extremely challenging. There isnt much research attempted to optimize the algorithm or update the network properties. In order to address the above mentioned challenges, dynamic graph analytics has evolved into a very active area of research over the last 3 years. This dissertation focuses on addressing important challenges in dynamic graph analytics, introducing a new data structure and highly optimized graph algorithms.

Graph Sparsification: In this dissertation, the author discusses elegant techniques for graph sparsification. The main idea behind graph sparsification is to identify edges relevant to graph properties. In this dissertation such edges are called as key edges. Once the key edges are distributed in a balanced sparsification tree, using that graph updates can be performed in parallel. Consider computing the minimum weighted spanning tree (MST) in a given graph, which can be also used for finding clusters. To track how the clusters change with time, the MST needs to be recomputed each time. The overall complexity for this operation each time step is $O(E + VlogV)$, where E is the number of edges and V is the number of vertices. In contrast, using a graph sparsification, the complexity only depends on how many edges actually alter the current MST. Since the MST has only (V-1) edges, in practice, only a few deleted edges affect the topology. Inserting a new edge have a greater probability of changing the MST. The complexity to insert an edge can vary depends on the length of a path can be max $O(V)$. In general most of the changed edges don't contribute to the new MST, analyzing the edges which contribute to new MST will be an efficient approach. A

sparsification framework will provide a more efficient updating algorithm.

In earlier work graph sparsification has been proved to be scalable on the theoretical PRAM machines [2,3]. This dissertation presents a first scalable parallel implementation of graph sparsification over large networks.

Main Deliverable: The dissertation presents a suite of efficient scalable and portable algorithms for updating properties of large dynamic networks (around billions of vertices). When compared to other network analysis tools and libraries, the proposed software can run on networks with weighted and/or directed edges. The proposed approach can be customized or tuned for better performance on different architectures including distributed, shared, and heterogeneous systems.The results are validated over large real-world and synthetic networks. The datasets and experiment related setup is mentioned in Chapter 3.

1.1 Challenges in Developing Parallel Algorithms for Dynamic Networks

The main challenges of developing parallel algorithms for dynamic networks are as follows:

First challenge is the memory latency due to graph traversal operations. As mentioned above all the graph operations irrespective of the type static or dynamic, are based on graph traversal. Network data in the real-world is highly unstructured, it is impossible to determine in-advance which edges or nodes will be accessed per step of the traversal. The issue becomes further challenging for dynamic networks where network topology changes with time.

Second Challenge is updating network properties more efficient than recomputing In general algorithms for updating dynamic networks cannot be a simple extension of their static counterparts. For example computing MST does not require many traversal, however, in the case of edge insertion, it is required to find a path and replace a higher weighted edge if they exist. This operation is expensive and requires more traversal. Optimizing the state of the art algorithm can depend on several factors such as infrastructure, data size, the network property to be computed, and the computational parameter to be optimized such as time, memory and energy.

1.2 Dissertation Overview

This dissertation presents a scalable, portable, and open source software for updating large scale dynamic networks by performing following research tasks.

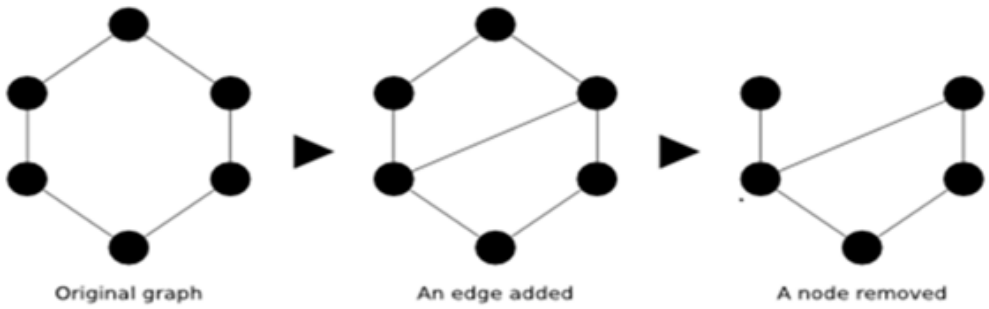

Figure 1.2: Dynamic Network, given a original network at time T_0, later a edge is added, and a node is removed from the original network.

Task 1: Scalable Algorithm for Dynamic Networks In related work the first challenge is addressed by using a technique called graph sparsification to develop parallel algorithm for wide range of network properties. The approach is modified to make the algorithms portable and scalable on different platforms including shared memory, distributed memory and GPUs.

Task 2: Performance Optimization Second challenge is to develop heuristics and approximate algorithms to optimize various performance factors.

Task 3: Handling Error due to Computation Third challenge by developing metrics to measure the stability of the networks. The stability of the networks means if there is a small perturbation in the network structure shouldn't cause a large change in the analysis results. The dissertation develop check pointing strategies for updating network algorithms.

The dissertation is structured as follows:

- **Chapter 2**

  First, the author discusses the related work on dynamic graph analytics, introducing the STINGER framework, limitations of the framework, and the challenges in developing scalable graph algorithms.

- **Chapter 3** In this chapter, the author introduces his first novel contribution, which is developing a scalable network algorithm for updating graph connectivity and Minimum Spanning Tree (MST), using a graph sparsification framework. Followed by scalability results on real -world and synthetic networks. Next author compares the scalability results with Galois and introduces mathematical proofs to validate the proposed approach.

- **Chapter 4** Here the author proposes an improved modified data structure and algorithm to update the Single Source Shortest Path (SSSP). Followed by a discussion on the scalability

results, and sufficient mathematical proofs to validate the contribution. There was a joint work with another collaborator on a distributed platform for updating the SSSP, the author introduces the approach and discusses the results.

- **Chapter 5** This chapter provides a detailed overview of the future work which author proposes to do over the next 2 years for his Ph.D. dissertation.

Chapter 2

Related Work

2.1 Related Work

Until the late nineties research work on dynamic networks were primarily focused on developing and updating topological characteristics, and importance was given to the theoretical complexity rather than the implementation. Below author gives a brief overview of the related work in parallel graph algorithms for static and dynamic networks. *Parallel and/or Dynamic Network Algorithms* In the recent years, there has been enormous growth of parallel network algorithms such as Galois [4], PHISH [5], and iSpan [6]. However most of them are focused on static networks. A special issue of Parallel Computing on "Graph analysis for scientific discovery" presents some of the latest advances in this area. The most common abstraction in developing parallel algorithms include;

Matrix Based Graph Approach Network algorithms are expressed as sparse matrix. Sparse matrix operations make it easier to parallelize on distributed memory systems. Implementing complex graph algorithms such as community detection as matrix operation is challenging and not intuitive. To best of the knowledge, this approach has been implemented only for static network.

Vertex Centric Algorithm GraphLab [7] and Pregel [8] use a vertex centric approach, where the vertex is updated locally, for global updates synchronization function is used, currently this approach has been extended to subgraph centric graph algorithms [9]. This approach has been only tested in static networks.

Multithreaded Graph Algorithms uses fine-grained parallelism to generate enough tasks to keep processing units busy, while waiting for data. This approach is suited for platforms which have huge amount of processing units. For algorithms to be scalable, networks must be extremely large,

and diameter should be small. This characteristics is not found in all networks for example, gene correlation networks have large diameters when compared to social networks.

Graph Sparsification This approach uses divide and conquer strategy to reduce the dependence on the edge in a graph. This is a popular approach used for updating dynamic network topological properties. Parallel version of these algorithms are studied only for a conceptual version and no empirical results were presented. This dissertation uses this approach initially to store the network and update the sparsification tree as the network updates over time.

There are numerous parallel network algorithms, that have been developed for particular network property analysis. Some of them are:-

Connectivity-based algorithms such as finding the breadth first search (BFS), depth first search (DFS), or connected components have been developed on distributed memory and GPUs. Sequential algorithms for dynamic updates are discussed in [10].

Centrality metrics includes degree, closeness, betweenness, eigenvector, and Page Rank. These metrics mentioned above measure the importance of vertices. For massively large networks, computing centrality metrics is expensive. To reduce computational complexity there are many approximate algorithms developed for dynamic networks. There are also few algorithms designed to compute approximate centrality metrics in GPUs [11].

Community detection identifying group of vertices that are closely connected with each other when compared to other vertices in the network. There are many community detection algorithms that are implemented in parallel. Methods for dynamic community detection are mostly sequential such as GraphX [12]. Network motifs are subgraphs with small number of vertices whose frequencies are used to understand the network properties. There are few implementations of sequential and parallel for finding motifs.

Parallel Network Analysis Software Parallel Boost Graph Library [13], Knowledge Discovery Tool Kit [14], and Giraph [15] are implemented on distributed memory. The well known shared memory implementation are SNAP [16], Galois [17], and Network Kit [18]. Map-reduced based approach such as PEGASUS [19], PREGEL [8], and GraphLab [20]. There are few parallel static graph partitioning software such as METIS [21] and Zoltan [22]. Software packages that include dynamic graph partitioning are PTScotch [23], and ParMeTIS [24].

Parallel Algorithm for Updating Dynamic Networks The well known software to update dynamic networks are STINGER [25] and PHISH [5]. STINGER [26] is implemented on shared memory systems. Their proposed approach is highly scalable, particularly on massively multithreaded machines. The software available from STINGER is able to compute the clustering coefficient, con-

nectivity and betweenness centrality of a streaming network.To date there are no implementations of SSSP, MST implementations of STINGER [26]. PHISH [5] uses MapReduce approach for updating dynamic networks. PHISH only offers connected components, triangle enumeration and subgraph isomorphism. Both the available softwares are well suited for undirected and unweighted networks, there are no extensions currently available for directed and weighted networks.

2.1.1 STINGER

STINGER stands for Spatio-Temporal Interaction Networks and Graphs Extensible Representation [26]. This software proposes a high performance extensible data structure for dynamic networks. The data structure is a simple extension of linked list. Edges incident for a vertex are stored in a linked list of edge blocks. A edge can be thought as a tuple which holds the ID for the neighbor vertex, type, weight and timestamps. The block holds the metadata.

The proposed data structure allows parallelism at various levels. Each vertex holds its own linked lists of edge blocks. In a given block all edges have a same edge type and blocks are accessed using logical vertex array. Loop is parallelized over these lists. Within a edge block, parallel loops can be used to traverse incident edges. The software allows user to define the level of parallelism.

Data structure uses a secondary index that points all edge blocks of a given type. The software provides a macros for parallel edge traversal. All operations for updating the changed edges are performed in parallel. STINGER is written in C and OpenMP and Cray MTA pragmas for parallelization. The software is highly optimized for Cray XMT machine.

STINGER had two implementations, first they processed each changes edges one at a time. Single edge updates lack concurrency which is required to achieve high performance. The systems with many thread contexts and memory bandwidth there isn't insufficient work or parallelism in data structure to process single updates at a time. To fix this issue the edge updates are processed in batches. In the second implementation all the edges are processed in batches, updates on a given vertex is done sequentially to avoid synchronization. The software is currently only updates connected components and the experimental results presented are only for the synthetic networks RMAT [27].

2.1.2 Algorithms for Computing Minimum Spanning Tree and CC

There are couple of parallel algorithms available for spanning tree, such as breadth first search (BFS), and MST on static networks such as [28] in distributed memory, [29] in multicores, [30] in

massively multithreaded machines and [31] in GPUs. The well known parallel MST approaches are Shiloach-Vishkin approach [32] and Boruvkas algorithm [33]. Parallel algorithms for updating MST on dynamic networks were proposed in [34,35] for theoretical PRAM machines but no experimental results were presented.

STINGER [26] has parallel implementation of dynamically updating connected components. Srinivasan et al. [36] proposed a parallel implementation of updating connected components using graph sparsification.

2.1.3 Algorithms for Computing Single Source Shortest Path

Single Source Shortest Path is discussed in detail in chapter 4.There exist many parallel implementations of Dijkstra's algorithm for example Galois [4] which is used for comparison. There also exist a distributed framework Havoqgt [37] which also supports parallel SSSP. Dijkstra Strip Mined Relaxation algorithm (DSMR) was proposed by Maleki et al. [38] for both shared and distributed memory.

Dynamic Updates Ramalingam et al. [39] and Narvez et al. [40] proposed a SSSP algorithm for dynamic networks. Bauer at al. [41] proposed a batch-dynamic SSSP algorithm, they also conducted a study of various dynamic SSSP algorithms for batch upates. Vora et al. [42] proposed approximations for streaming graphs. All the above implementations mentioned above sequential implementation. There exist one parallel dynamic algorithm for updating SSSP [43] implemented on GPUs using JavaScript. However, their datasets contains small networks and no scalability results were present.

Updating Centrality Metrics SSSP computation is used to compute vertex centralities. There exist few parallel approximate algorithms [44–46] for computing those centrality metrics implemented on different platforms. There exists few algorithms for dynamic networks [30, 47–49]. However, the dynamic algorithms use approximation for computation.

2.1.4 Algorithms for Computing Strongly Connected Components

Tarjans algorithm [50], the classic sequential method for SCC detection, is an asymptotically optimal linear-time algorithm. Unfortunately, Tarjans algorithm is difficult to parallelize because it extends the depth-first search (DFS) traversal of the graph, which is inherently sequential [26]. Fleischer et al. [51] devised a practical parallel algorithm, the Forward-Backward (FW-BW) algorithm, which motivated further enhancements in following research. The FW-BW algorithm achieves parallelism

by partitioning the given graph into three disjoint subgraphs which can be processed independently in a recursive manner. McLendon et al. [52] added a simple extension to this algorithm, the Trim step, which resulted in a significant performance improvement.

Barnat et al. [53] proposed the recursive OBF algorithm to improve the degree of parallelism compared to the original FW-BW algorithm. However, their method [] did not give a large performance improvement over McLendon et al. when applied to real-world graphs with few large-sized SCCs. Barnat et al. [53] demonstrate a CUDA implementation based on forward reachability that outperforms the sequential Tarjans algorithm, but concede that none of their implementations on a quad-core system were able to outperform Tarjans algorithm.

On the other hand, Hong et al. [54] improved the FB-Trim algorithm with an efficient parallel CPU SCC detection method specifically for processing real-world graphs. They used a two-phase method to handle small-world graphs, and got tremendous speedup on multicore CPUs. Hongs work implies that graph algorithms should be aware of graph properties and make adjustment to handle different situations. Graph properties are also critical for GPU implementations as we evaluated.

Chen et al. [55] decompose the SCC detection into two phases: processing the giant SCC and processing the remaining small-sized nontrivial SCCs. The two phases utilizes different parallelism approaches. The single giant SCC is full of data parallelism while the large amount of small-sized SCCs can benefit from task parallelism. To enable efficient task parallelism in the second phase, we examine optimizations that previously utilized in CPU SCC and port them to the GPU.

Slota et al. [56] rely on an approach that uses multiple steps, each of which can be parallelized effectively, to find the strongly connected components. They find trivial strongly connected components of size one or two first and then use breadth first search to find the largest strongly connected component and an iterative color propagation approach to find multiple small strongly connected components. This algorithm can be parallelized effectively in shared-memory (multicore) computers.

Chapter 3

Scalable Algorithm to Update Network Connected Components & Minimum Spanning Tree

A connected component in an undirected network is a maximal set of vertices such that each pair of vertices is connected by a path. Figure 3.3 shows an example of connected component. The popular approach to compute connected components is using a DFS or BFS based approach. Connected Components have many applications in real-world such as finding vertex reachability. Algorithm 1 given a network $G = (V, E)$ is to identify vertices that are in the same component. In general if two vertices are in same component there exist a path between them. At the end of execution of algorithm 1 each vertex is associated with a root which gives unique identifier to its component.

A minimum spanning tree in a network is given a connected undirected network, is a subset of edges that form a tree and connects all the vertices in the network. For a given network there are various possible spanning tree, however for this dissertation, A minimum weighted spanning tree (MST) is a spanning tree with total edge weight less than or equal to weight of other possible spanning tree of the same network. Algorithm 1 pseudo code briefs the process of finding CC and MST and extension of Kruskal's method. The only difference between CC and MST approach is sorting operation to find edges with lowest weight for MST. The updating algorithm for both cases will also be similar, difference lies in how weights are handled.

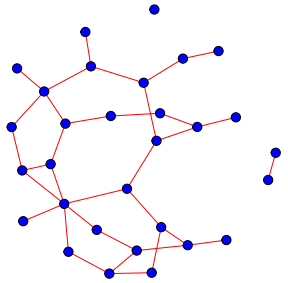

Figure 3.1: Example of Connected Components

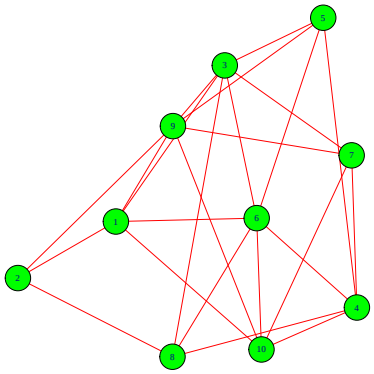

Figure 3.2: Toy Network

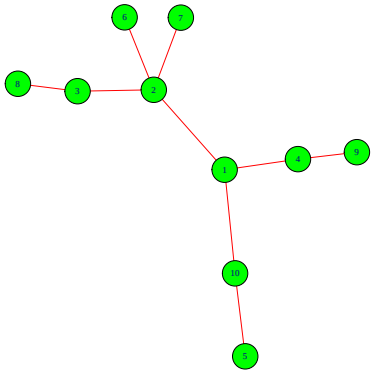

Figure 3.3: Example of Minimum Spanning Tree (MST) of the above Toy Network Figure 3.2

Algorithm 1 Extracting CC or MST

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

```
   /* Initialize root for each vertex                                   */
 1 for i ← 0 to |V| do
 2 |  Root[i] = i
 3 end
   /* Set List of Output Edges to Null                                  */
 4 E_x ← ∅
   /* Check for all Edges                                               */
 5 for i ← 0 to |E| do
 6 |  this_edge ← E[i]
   |  v and u are endpoints of this_edge
   |
   |  /* Find the root of v                                            */
 7 |  root_v = v
 8 |  while Root[root_v] ≠ root_v do
 9 |  |  root_v = Root[root_v]
10 |  end
   |  /* Find the root of u                                            */
11 |  root_u = u
12 |  while Root[root_u] ≠ root_u do
13 |  |  root_u = Root[root_u]
14 |  end
   |  /* If the endpoints are in separate components, join them        */
15 |  if root_v ≠ root_u then
   |  |  /* Add to Key Edges                                          */
16 |  |  E_x ← E_x ∪ this_edge
   |  |   min ← min(root_v, root_u)
   |  |   Root[root_v] ← min
   |  |   Root[root_u] ← min
17 |  end
18 end
```

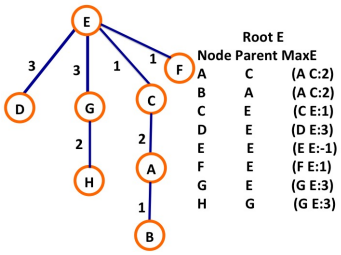

Figure 3.4: Rooted Tree

3.1 Rooted Tree for Reducing Graph Traversals

In general it has been observed that most network algorithms frequent operations are traversal. Traversal is considered to be an expensive operation, complexity of traversal can be a much as O($|V|$). In the case of CC if a key edge is deleted traversal is required to identify new roots of the disconnected components. To solve the traversal challenge, author proposes storing the information of the tree structures in a rooted tree as follows. Vertex with the highest degree from the network is selected as the root of the tree. Each vertex v in the rooted tree stores the following information, (1) root of the tree,r, (2) parent R_V, which is also the neighbor that is also one level higher than its BFS tree. (3) Maximum weighted edge in the path to the root. Figure 3.4 gives an example of the proposed weighted rooted tree.

3.1.1 Inserting and Deleting the New Edge in the Proposed Weighted Tree

Considering an edge needs to be added to the graph. In the case of CC, the decision to add this edge to the tree can be done in the constant time. If a edge needs to be deleted, considering that the edge is a key edge and after deletion the tree becomes disconnected. For MST this decision is based on highest edge weight on that path. Rooted tree structure helps finding the highest weighted edge in the constant time. Figure 3.5 illustrates the different cases.

Case 1: Vertices a, and b are in different branches from the root. In this case path passes through the root and the weighted edges are already stored, this operation requires constant time.

Case 2: Both vertices are in the same branch from the root. Additionally, they branch to different paths at the fork vertex.

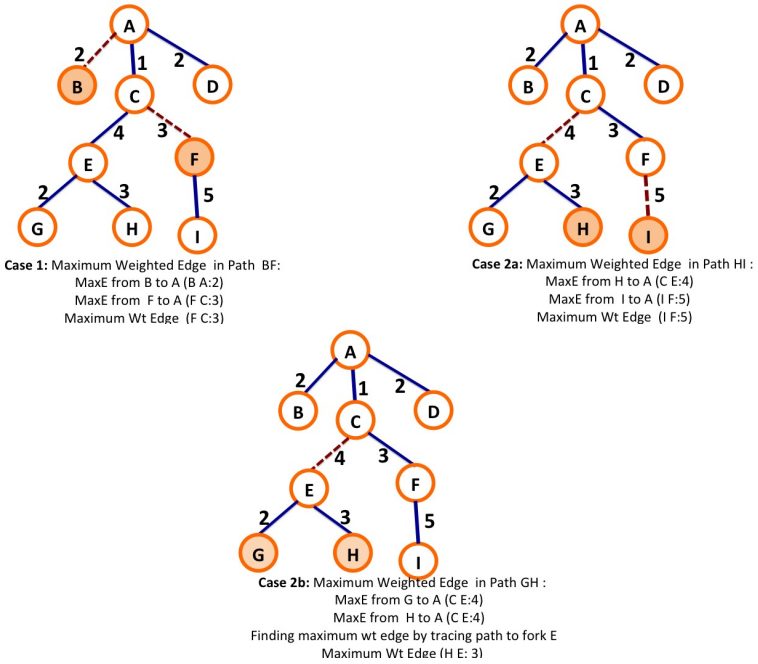

Figure 3.5: Three cases by which to find the maximum weighted edge in a path. The colored nodes are the endpoints of the path. The dashed red lines are the maximum weighted edge from the nodes to the root A.

3.2 Algorithm Implementation And Complexity

Algorithm 2 presents a high level overview of the proposed approach for updating weighted trees. **Data Structures:** Given the list of changed edges and remainder edges are stored as a vector. All edges are marked either for insertion or deletion.Each vertex in the rooted tree maintains the information about its parent, and root of the tree.

3.2.1 Step 1: Creating the Rooted Tree

The first step is to create a rooted tree based on the key edges.First thing is given a network identify the vertex in the network which has a high degree, and assign it as a root. Next step is to perform traversal from the neighbors of the root node, in this case there are two options, first is the Depth first search (DFS), next is the Breadth first search (BFS). Author has chosen to do BFS, each BFS traversal can be done in parallel. As a part of the traversal each vertex is assigned a parent. Once all the BFSs are completed, next step is to check if there are any such vertices where parents are not assigned. This can occur if graph has multiple components, in that case those vertices becomes the new root and BFS traversal is executed again. This process is repeated until each vertices has been assigned a parent. Pseudo code for creating the rooted tree is presented in the Algorithm 3

Algorithm 2 Parallel Algorithm for Updating Connected Components

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.

```
19 Function Main(E_x, E_r, CE)
     /* Rooted Tree is an array of type RV of size V                    */
20   RV RootedT[V]
21   Initialize Each Vertex in Rooted Tree
     /* Create Rooted Tree                                              */
22   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                               */
23   int Status[CE]
24   Edge Marked[CE]
25   while CE is not empty do
       /* Process Changed Edges for Insertion and Deletion              */
26     Classify_Edges(CE, Status, Marked, RootedT)
       /* Process Edges as per Status                                   */
27     Process_Status(E_x, E_r, CE, Status, Marked, RootedT)
28   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.                                                    */
29   Repair_Tree(E_r, Status, Marked, RootedT)
     /* Process Repair Edges                                            */
30   Process_Status( E_x, E_r, CE, Status, Marked, RootedT)
31   return
```

3.2.2 Challenges in Parallel Rooting

Creating a rooted tree is mostly a sequential process, with a very little opportunity to parallelize. There are few alternatives for creating rooted tree in parallel such as computing BFS in parallel. However they aren't effective when compared to the proposed approach.

3.2.3 Step 2: Classifying the Changed Edges

The set of changed edges are processed to determine whether they will be added or deleted from the tree edge or the remainder edge set. All the set of changed edges are processed in parallel and their status is marked in an array, named status. The Pseudo code for step 2 is given in the algorithm 4 .

The status of all changed edges is initially set to $NONE$. During the process of classifying changed edges status can take following values.

Deletion: If the changed edges has to be deleted status is marked as DEL. *Insertion:* If the changed edges had to be inserted status is marked as $INSERTION$.

Algorithm 3 Step1: Creating the Rooted Tree

Function `Create_Tree(`$RootedT$, E_x`)`
Input : The set of key edges, E_x
Output: The rooted tree, $RootedT$
while *Parents not assigned for all vertices* **do**
`/* Find vertex` r `with highest degree, whose parent has not been assigned. Set` r `as root */`
$RootedT[r].Parent \leftarrow r$
$RootedT[r].Root \leftarrow r$
for *All vertices* v *that are neighbors of the root* **do in parallel**
`/* Set Root, Parent and maximum weighted edge of` v `*/`
$RootedT[v].Root \leftarrow root$ $RootedT[v].Parent \leftarrow mynode$
$myE = (v, root)$
$RootedT[v].maxE \leftarrow myE$
`/* Initialize Queue for BFS */`
$NodeQ \leftarrow \emptyset$
Push v into $NodeQ$
while $NodeQ \neq \emptyset$ **do**
`/* Pop front element */`
$Mynode \leftarrow NodeQ.top()$
`/* For all neighbors */`
$Neigh \leftarrow$ neighbors of $Mynode$ as per E_x
for $i \in Neigh$
if $RootedT[i].Parent = -1$ **then**
`/* Assign Root and Parent */`
$RootedT[i].Root \leftarrow root$ $RootedT[i].Parent \leftarrow mynode$
`/* Check if` $myE = (i, mynode)$ `is the maximum weighted edge in the path from` i `to the root */`
if $myE.edgewt < RootedT[i].maxE.edgewt$ **then**
$RootedT[i].maxE \leftarrow myE$
end
Push i into $NodeQ$
end
end
end
end
return

Algorithm 4 Step 2: Classifying the Changed Edges

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 */`

56 **for** $i = 0$ *to* $|CE|$ **do in parallel**

`/* Initialize Status and Marked. */`

57 $E \leftarrow CE[i]$

58 $Status[i] \leftarrow NONE$

59 $Marked[i] \leftarrow dummy\_edge$

60 **if** *E marked as deleted* **then**

61 $Status[i] \leftarrow DEL$

62 **end**

63 **else**

`/* E marked as inserted */`

64 $u \leftarrow E.node1$

65 $v \leftarrow E.node2$

`/* If connecting disconnected components */`

66 **if** $RootedT[u].Root \neq RootedT[v].Root$ **then**

67 $Status[i] \leftarrow INS$

68 **end**

69 **else**

`/* Check if edge can be replaced */`

70 Find maximum weighted edge, $MaxW$, in path from u to v

`/* Check if` E `can replace` $MaxW$ `*/`

71 **if** $MaxW.edgewt > E.edgewt$ **then**

72 $x \leftarrow MaxW.Rpl\_Id$

73 **if** $x = -1$ *OR* $CE[x].edgewt > E.edgewt$ **then**

74 $Marked[i] \leftarrow MaxW$ $Status[i] \leftarrow RPL$ $MaxW.Replace\_Id = i$

75 **end**

76 **end**

77 **end**

78 **end**

79 **end**

3.2.4 Step3: Processing Edges by Status

After step2 is performed, they are either added or removed from their respective edge sets.

Algorithm 5 Step 3: Processing Edges By Status

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*

80 **for** i = *0 to* $|CE|$ **do in parallel**

`/* Get Edge and its Status */`

81 $E \leftarrow CE[i]$

82 $S \leftarrow Status[i]$

`/* Deleting Key Edge */`

83 **if** $S = DEL$ **then**

84 Delete Edge E from Appropriate Edge Set
**** What does "Appropriate" mean here **** Assign Weight of Edge E to INF, i.e., very high value to mark it as deleted

85 **end**

`/* Edge into Remainder Edges */`

86 **if** $S = NONE$ **then**

87 Add Edge E to the Remainder Edge Set

88 **end**

`/* Edge into Disconnected Tree */`

89 **if** $S = INS$ **then**

90 Add Edge E to the Key Edge Set

91 **end**

`/* Replacing Edge from Key Edge */`

92 **if** $S = RPL$ **then**

`/* Find Edge to be Replaced */`

93 $E_{rpl} \leftarrow Marked[i]$

`/* Replacing Edge Matches Current Inserting Edge */`

94 **if** $E_{rpl}.Rpl\_Id = i$ **then**

95 Add Edge **** Which Edge ? **** to the Key Edge Set, E_x

96 Set Weight of E_{rpl} to -1 to mark it as deleted

97 **end**

98 **else**

`/* Replacing Edge Does Bot Match */`

99 Add E to set of new changed edges

100 **end**

101 **end**

102 **end**

103 Execute BFS on E_x from *root*, to assign the parents and maximum weighted edges in the modified *RootedT*

3.2.5 Repairing the Tree

After changed edges have been processed, if the tree is disconnected, repairing is performed using the remainder edges to reconnect. For each remainder edge, validation is performed whether the maximum weighted path between the two endpoints and is set to infinity. Status of the remainder

edge as RPL and the replaced id of the edge to be replaced with the index. This process is similar to edge classification as mentioned in the step2. Once the remainder edges are processed, the once marked with RPL replace the deleted edges in a process similar to the one described in Step3.

3.3 Experimental Results

3.3.1 Experimental Setup

In this section author presents experimental setup such as datasets specification, and machine used for computation.

Datasets

Synthetic Graphs:

For synthetic networks R MAT model is used which is based on recursive Kronecker matrices. The degree distribution of the graph is defined by four values (a,b,c,d), whose sum adds up to 1.

For all the experiments two types of RMAT graphs are generated. The first type of RMAT has following specification ($a = 0.45, b = 0.15, c = 0.15, d = 0.25$) labeled G and has scale free degree distribution. The second specification has ($a = b = c = d = 0.25$), labeled ER, is a random network with normal degree distribution. comparing between these graph can help to determine how degree distribution affects the performance. Graphs RMAT24, RMAT25, and RMAT26 have around 16M vertices, 268M edges, 33M vertices, and 536M edges, 67M vertices, and 1000M edges respectively. Each vertices in the synthetic networks has an average of 8 edges per vertex.

Real-world Graphs:

This dissertation consist of three real world networks from the Stanford Network Database [16], YouTube, Pokec and Live Journal, (Table 3.1 mentions the sizes)

Table 3.1: Real-World of Graphs used in this dissertation.

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

Few synthetic networks are not weighted, using a random number generator for each edge, all edges are assigned weight from 1 to 100. The set of changed edges are weighted from 1 to 100.

3.3.2 Infrastructure

All experiments for dissertation were performed on a 36-core (72 thread) Intel Haswell server with 256GB DDR4 RAM, with two Intel Xeon E5-2699 v3 2.30 GHz CPUs. The operating system is Ubuntu 16.04. Shared memory code is implemented in C++ and OpenMP, and was compiled with GCC version 4.8.5.

Now experimental results are presented for computing the connected components and Minimum Spanning Tree (MST) on a dynamic network. Figure 3.3.4 represents the time and scalability results for computing MST on the synthetic and real-world networks. Each of the networks were updated by 50 million edges, with the percentage of insertions ranging from 50 %, 75 %, and 100 % of the total changed edges. 50 million changed edges represent 37 %, 19 %, and 9 % of the original edge counts for RMAT24,25 and 26 respectively. Real-world datasets 50 million edge represent 1673 % (Youtube), 163 % (Pokec), and 73 % (LiveJournal). Figure 3.3.4 only shows scalability for the MST, as it has more complex operations, however CC also shows similiar trend. In general results are scalable, on few occasion they flatten out at higher scales due to the inherently sequential nature of creating rooted trees. In random networks percentage of insertion makes no difference in random networks, in the case of real-world and scale-free graphs updated due to 100 % insertion are faster. The time and scalability is sensitive to the degree distribution. Given the same network size scale-free networks takes significantly less time when compared to the other networks with normal degree distribution. The diameter of scale-free graph is shorter than the random graphs. Given the equal distribution of weights in the edges, MST is likely to have a lower diameter, which leads to lower height of the rooted-tree, which reduces runtime. In real-world networks Pokec network is 30 times denser when compared to Youtube, however, the timings for both are similar. Execution time of for updating algorithm is not sensitive to the density of the network.

3.3.3 Effect of Selection of Root

The complexity of insertion and deletion of an edge is proportional to the height of the tree. Considering one synthetic network (RMAT24-G), and one real-world network (Live Journal), next while selecting the root, choosing the one which gives shortest height, the root with trees that gives greatest height and six other heights in between. Figure 3.7 shows very little significance of time when number of processing unit is small and variation becomes negligible if the number of processing unit increases. The difference between the tallest and the shortest height is not very large and there is very little significant difference during the update phase.

3.3.4 Comparison with Recomputing Approach

In this section a detailed comparison is made comparing the time taken to update the CC and MST with time taken to recompute from scratch using Galois []. Galois software identifies low level parallelism such as loops in the graph algorithms. Galois have parallelized the loops and has shown good performance and very fast compared to other parallel network packages. Galois has a parallel algorithm for finding CC and uses parallel Boruvka algorithm for computing MST. Both CC and MST exhibit same speedup and memory performance, this dissertation focuses on the MST.

Parallel Speedup

The ratio of sequential to parallel execution time i.e. (T_1/T'_P), where p is the number of threads. Figure 3.6 shows a speedup values for RMAT24-G , 3 real-world networks with 1 million changed edges, and 75% insertions (bottom row) 100 % insertions(top row). The speedup for both galois and proposed approach degrades as number of thread increases. The speedup for both methods is better fore dense or sparse networks. Sparse and dense network require more computation such as Live Journal. This dissertation focuses on improving the speedup of dynamic algorithms. As rooting tree computation is a one-time cost and it gets amortized by the cost of multiple updates, it is not counted towards the speedup. Table 3.2 shows breakdown of the times (measured in seconds) for different phases. First column in the table 3.2 indicates the network from the dataset mentioned in chapter 2, second column mentions the percentage of insertions in the update. The third column is the number of threads and MST Root, MST Ins-Del, MST Repair, MST Update and Galois Total shows average times for rooting, insertion/deletion, repair, and Galois time includes the full tree computation.

Memory

Figure 3.9, demonstrates total memory used by the proposed approach, and Galois for 3 real-world networks, and RMAT24-G. All the experiments for memory were performed with a million changed edges. For all the networks the memory footprint of the recomputing (Galois) algorithm uses up to 24x the amount of memory used by update algorithm.

Power and Energy

Power and Energy consumption of the proposed approach is compared with the Galois. Performance Application Programming Interface (PAPI) [] with Intel's running average power limit (RAPL),

provides access to a set of hardware counters measuring energy usage. PAPI computes energy in nanojoules for a given time period. All power and energy measurement experiments were performed on dual socket Intel Xeon E5-2699 @2.30 GHZ chips.

Three real-world networks and RMAT24-G synthetic network are considered to evaluate the power and energy scaling of the proposed approach and compare them with the Galois. figure 3.10 demonstrates the power and energy consumption of the dynamic MST computation applied to RMAT24-G. The experiments for power involved different percentage of insertions. The proposed approach consumes lower power when compared to the Galois. Deletions and repairing in the proposed approach takes more power than insertions. Galois computations require significantly more energy than the update algorithm. Figure 3.11 demonstrates the power,energy and time for real-world networks. Galois power consumption is relatively low for small thread counts, increases sharply for more than 16 threads. The update algorithm power consumption's remains relatively flat as more threads are used. In the case of insertion-only updates, few real-world networks on low thread counts Galois consumes less power than the proposed approach. As the number of threads increases, Galois consumes significantly more power than the proposed approach. In the case of deletion repair component makes the update algorithm less power-efficient than the recomputation.

Summary

To summarize the comparison experiments, proposed approach requires less memory than Galois. The proposed approach to update is faster, consumes less energy than Galois. In general the proposed approach has flatter power profiles at all levels of parallelism. Galois has overall better speedup than proposed algorithm, due to less amount of work performed by the update algorithm. In the case of high deletion scenarios the recomputation approach using Galois is recommended than the proposed approach. This is due to the fact that the cost of repair exceeds the benefit of not recomputing the tree from scratch.

Table 3.2: 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

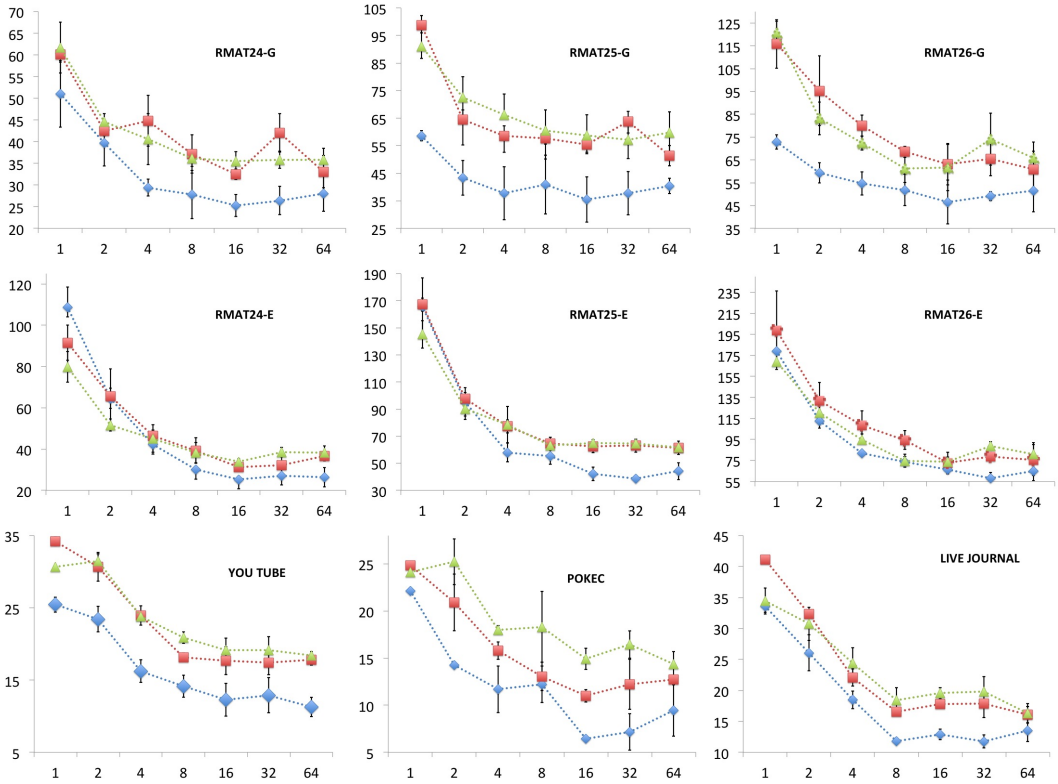

Figure 3.6: Scalability Results for Updating Networks. Top: Networks with scale-free degree distribution of order of 2^{24}, 2^{25}, 2^{26} vertices. Middle: Random Networks with normal degree distribution of order of 2^{24}, 2^{25}, 2^{26} vertices. Bottom: Real-world Networks, left to right (YouTube, Pokec, LiveJournal). Blue 100% insertions, Red 75% insertions, and Green 50% 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).

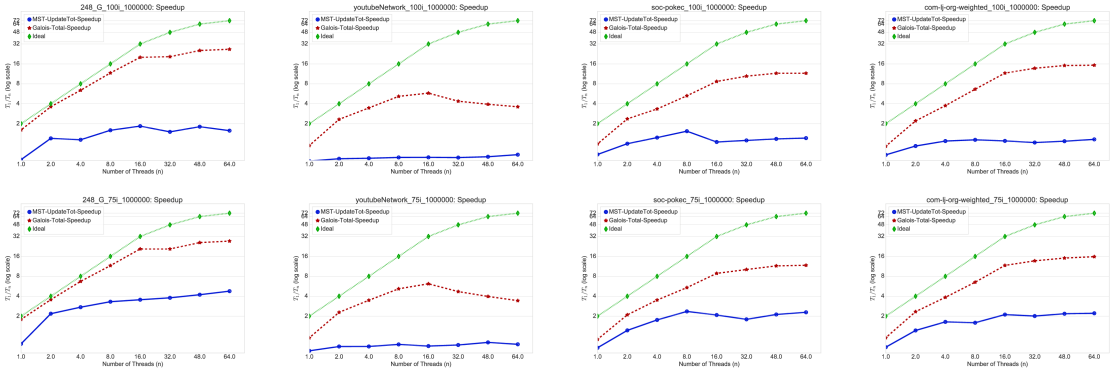

Figure 3.7: Parallel speedup (log scale) for computing the MST of RMAT-24G and three real-world networks (described in Sec **??**). Speedup was computed based on one batch update, with batch size 1,000,000 edges. Top: 100% insertions. Bottom: 75% insertions.

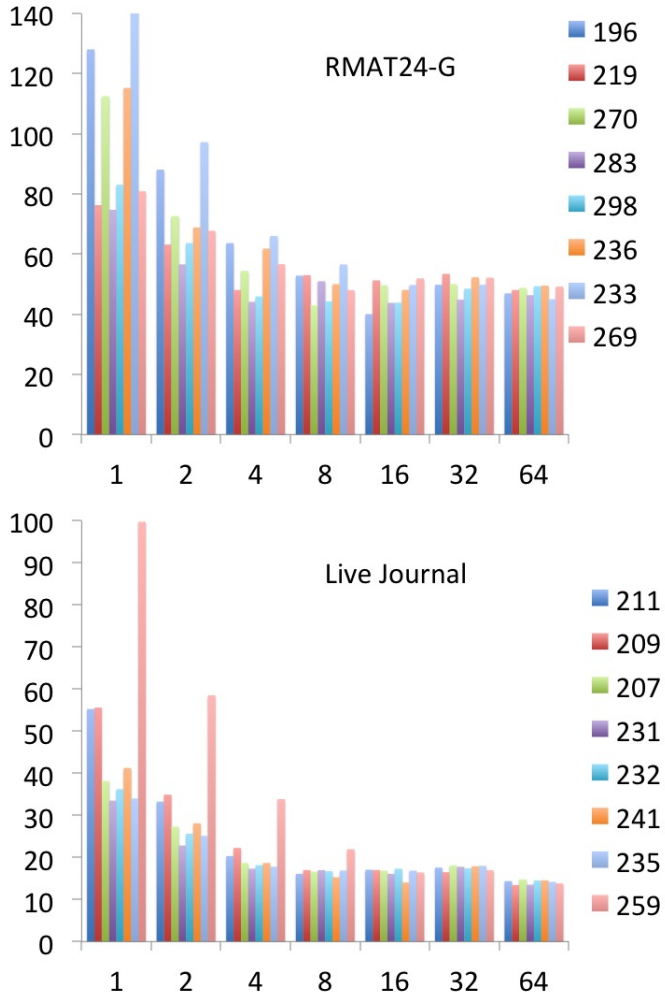

Figure 3.8: Variations of updating time based on the choice of root. Left: RMAT24-G. Right: LiveJournal. The Y-axis gives the time in seconds. The X-axis gives the number of threads. Each colored bar represents a tree of different height. The heights are given in the legend. The results are for 100% 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.

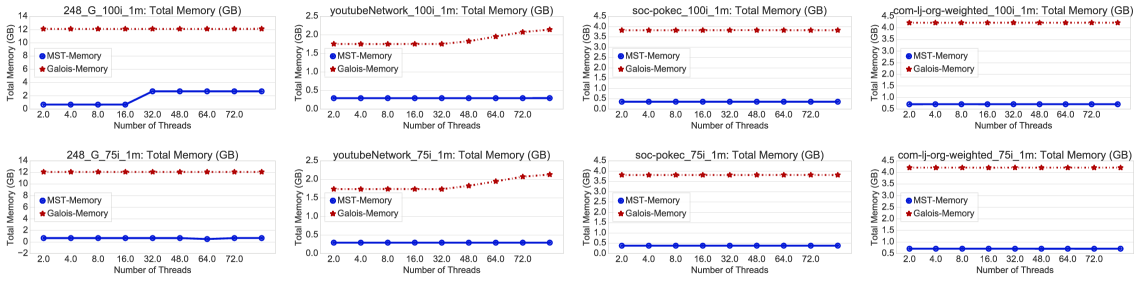

Figure 3.9: Total memory use for computing the minimum spanning tree of RMAT-24G and three real-world networks (described in Sec **??**). Top: 100% insertions. Bottom: 75% insertions.

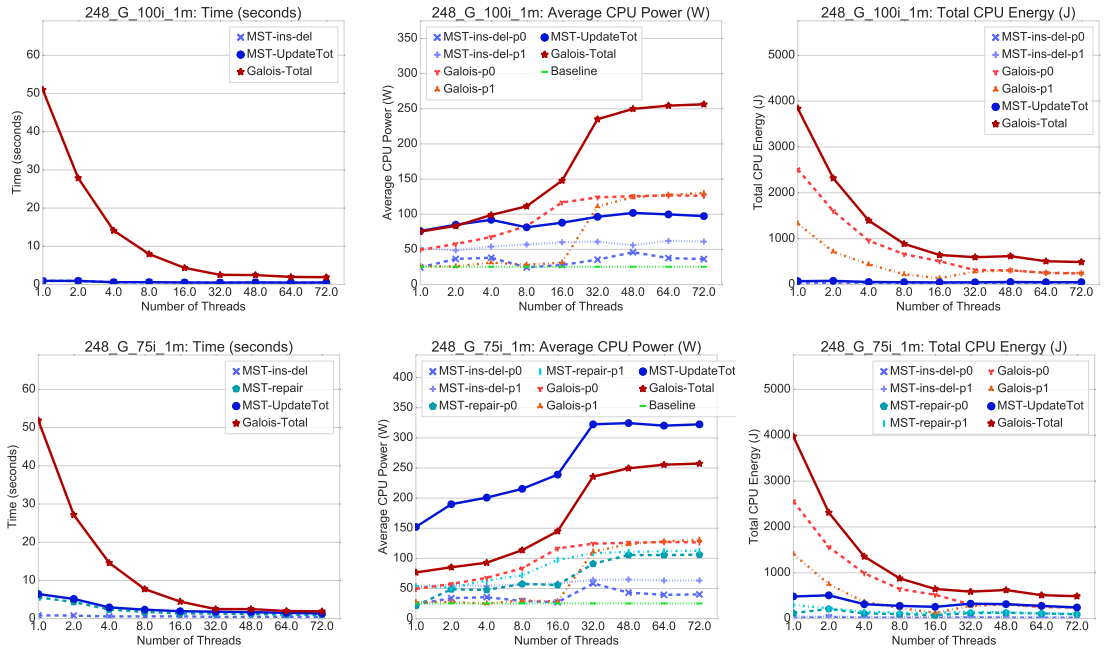

Figure 3.10: Comparison of per-socket and total power and energy measurements: a single batch update (blues), broken into "Insertion and Deletion and "Repair," compared with recomputing using Galois (reds) of the minimum weighted spanning tree. Top: 100% insertions Bottom: 75% insertions. Left: runtime. Center: power measurements. Right: energy measurements.

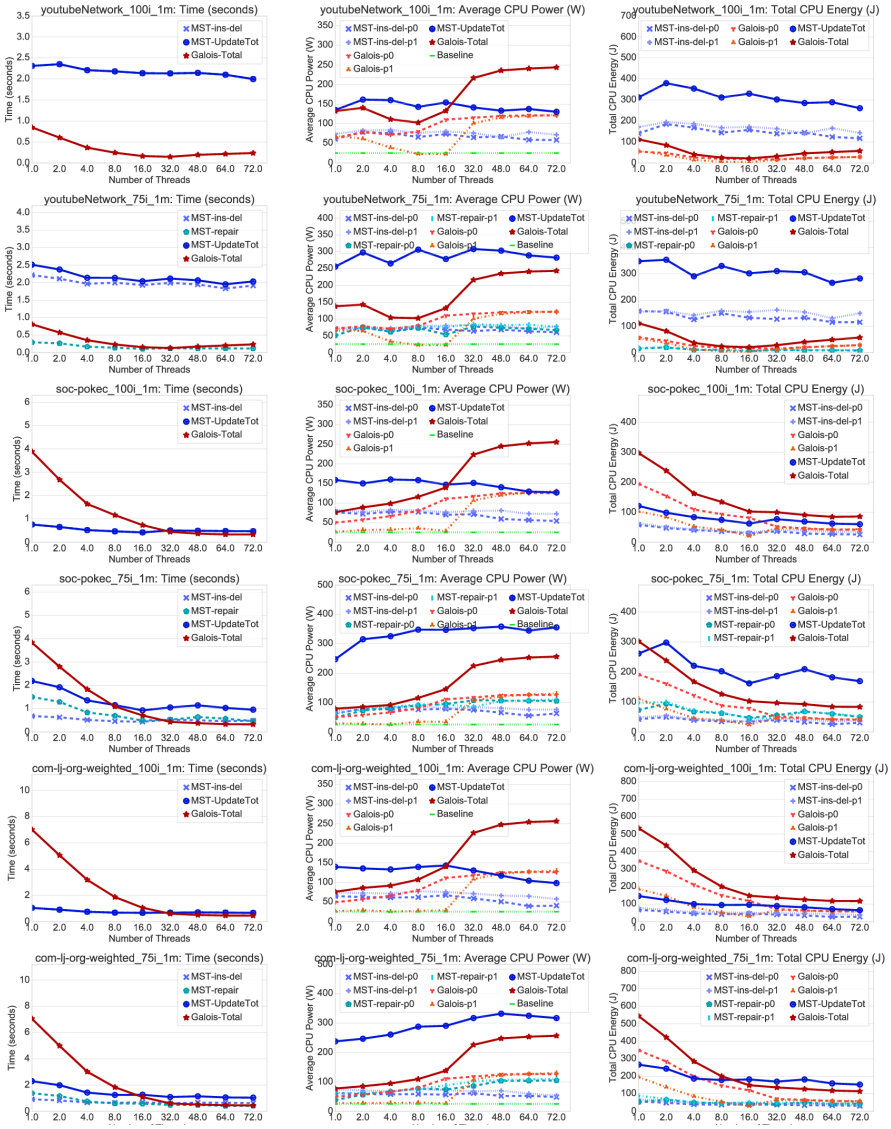

Figure 3.11: Real-world network comparison of per-socket and total power measurements: a single batch update (blues), broken into "Insertion and Deletion" and "Repair," compared with recomputing using Galois (reds) of the minimum weight spanning tree. Top two rows: YouTube (1.1M nodes, 3.0M edges). Middle two rows: Pokec (1.6M nodes, 30.6M edges). Bottom two rows: LiveJournal (4.8M nodes, 68.9M edges). For each network, the first and second rows represent 100% and 75% insertions, respectively.

3.3.5 Correctness of the Algorithm

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.

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.

□

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}.

□

Chapter 4

Scalable Algorithm to Update Single Source Shortest Path

Detecting the single-source shortest path (SSSP) is a classical graph theory problem, and has many practical applications such as calculating the optimal route in maps, internet routing, path planning for robots, and centrality analysis in the complex network. Given a network and a source vertex, the idea is to identify the shortest path from the source to the remaining vertices in the network.

In figure assuming **"1"** is the source vertex, an SSSP tree is constructed in the figure which gives the shortest path to the vertices **"2", "3", "4", "5"**. There exist many parallel algorithms for calculating SSSP, the popular once are Delta-stepping [57], and DSMR [38]. As mentioned in the introduction that these algorithms are good only for static networks, extending them to the dynamic

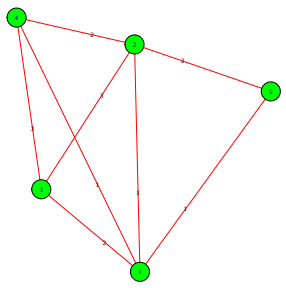

Figure 4.1: Sample Toy-Network

networks requires re-computation from scratch. In this chapter, the author extends the algorithm for updating the MST from the previous chapter to update SSSP, provide the necessary proof for correctness and the scalability results.

4.1 The dynamic graph problem for updating SSSP

Given a graph G =(V, E), source vertex **v**, SSSP tree **s**, a batch of edge insertions and deletions, update the SSSP tree.

4.2 SSSP Algorithm

Now the author presents a novel parallel algorithm, the key idea is to identify the potential changes in the network caused by the addition/deletion of a vertex/edge. In general when the updates are processed it only affects the subnetwork, not the entire network if the computation is restricted to the subnetwork then the computation time is significantly reduced. The algorithm consists of a two-step process, the first step is to identify the set of vertices in the network is affected by the changes, and the next step is to update the SSSP. Both the steps are done in parallel, the important advantage of breaking the operation in two steps is to improve the scalability and reducing redundant computations.

There are two ways that a batch of changed edges can be processed in parallel, the first approach is to distribute the work across the graph, each thread is assigned a subgraph. Each subgraph is updated in parallel, as per the changes that affect that subgraph. This approach has limitations such as load imbalance since in the real-world graphs changed edges may not be equally distributed across the network. There are chances that one subgraph might be processing more edges than the other. The second approach is to distribute the set of changed edges, then for each changed edge algorithm 6 is executed in parallel. There are more scalable approaches, however, there are couple of challenges such as -

- Load Imbalance
- Locking Mechanism to avoid a race condition
- Redundant Computations

To address the above mentioned challenges, a two step approach is proposed.Figure 4.2 gives an example of how the algorithm works-

- Step 1. Identify changed edges affecting SSSP (Algorithm 7), all changed edges are processed in parallel, next filter the once which will affect the SSSP. All edge weights are assumed to be positive. Edge (a,b) is marked for insertion if and only the distance of b is reduced as shown in the example. Considering if it happens then a is marked as the parent of b and the distance of b is updated. Now considering the edge (a,b) is marked for deletion, if the edge is part of the key edges in the SSSP tree, then the distance of b is changed to INF (given a high value) to mark that edge containing b to the tree is deleted. Both cases vertex b is marked as affected. In the case of insertion, there is an additional iteration if a vertex has multiple associated edges with it, the update is performed with the edges with the lowest weight.

- Step 2. Update the subgraphs of the affected vertices (Algorithm 8): Now that the edges which affect the SSSP are identified, subgraphs leading from the affected vertices are updated. Each vertex in the network is associated with a boolean value effect which is updated to true if the distance of the vertex has changed. Each iteration the function $Process\_Vertex\_Parallel$ is applied to all the affected vertices in parallel. The iterations started converging when there exist no vertices to update. The example is shown in the figure 4.2 at the first iteration the nodes are 1

  $Process\_Vertex\_Parallel$ is similar to algorithm 6, however instead of pushing vertices to the priority queue, which processes sequentially, they are marked and processed concurrently.

Algorithm 6 Updating SSSP for a Single Change

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

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

procedure UPDATING_PER_CHANGE(E, G, T, $Dist$, $Parent$)

▷ Initialize Updated Distance and Parent

for $v \in V$ **do**

end

$Dist_u[v] \leftarrow Dist[v]$

$Parent_u[v] \leftarrow Parent[v]$

▷ Find the affected vertex, x **if** $Dist_u[a] > Dist_u[b]$ **then**

$x \leftarrow a$, $y \leftarrow b$

end

else

end

$x \leftarrow b$, $y \leftarrow a$

▷ Initialize Priority Queue PQ and update $Dist_u[x]$

$PQ \leftarrow x$

if E *is inserted* **then**

$Dist_u[x] = Dist_u[y] + W(a, b)$

end

if E *is deleted* **then**

$Dist_u[x] = INF$

end

▷ Update the subgraph affected by x **while** *(PQ not empty)* **do**

end

v=$PQ.top()$

$PQ.dequeue()$

Process_Vertex(v,G, T, $Dist_u$, $Parent_u$)

procedure PROCESS_VERTEX(v, G, T, $Dist_u$, $Parent_u$) **for** n *where* n *is neighbor of* v *in* G **do**

end

$Dist_u[v] = INF$ AND $Parent_u[n] = v$

$Dist_u[n] = INF$

$PQ.enqueue(n)$ **else**

end

$Dist_u[n] > Dist_u[v]$+$W(v, n)$

$Dist_u[n] \leftarrow Dist_u[v] + W(v, n)$

$PQ.enqueue(n)$

$Parent_u[n] \leftarrow v$ **else**

end

$Dist_u[v] > Dist_u[n]$+$W(v, n)$

$Dist_u[v] \leftarrow Dist_u[n] + W(v, n)$

$PQ.enqueue(v)$

$Parent_u[v] \leftarrow n$

Algorithm 7 Step1: Processing Changed Edges in Parallel

Input: Graph $G(V, E)$, $Dist$, $Parent$, SSSP Tree T, Changed Edges CE. Weight of edge (v, u) is $W(v, u)$.
Output: Updated $Dist_u$, $Parent_u$ and SSSP Tree T_u based on structure of $Parent_u$

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

Algorithm 8 Step 2: Updating Affected Vertices in Parallel

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

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

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

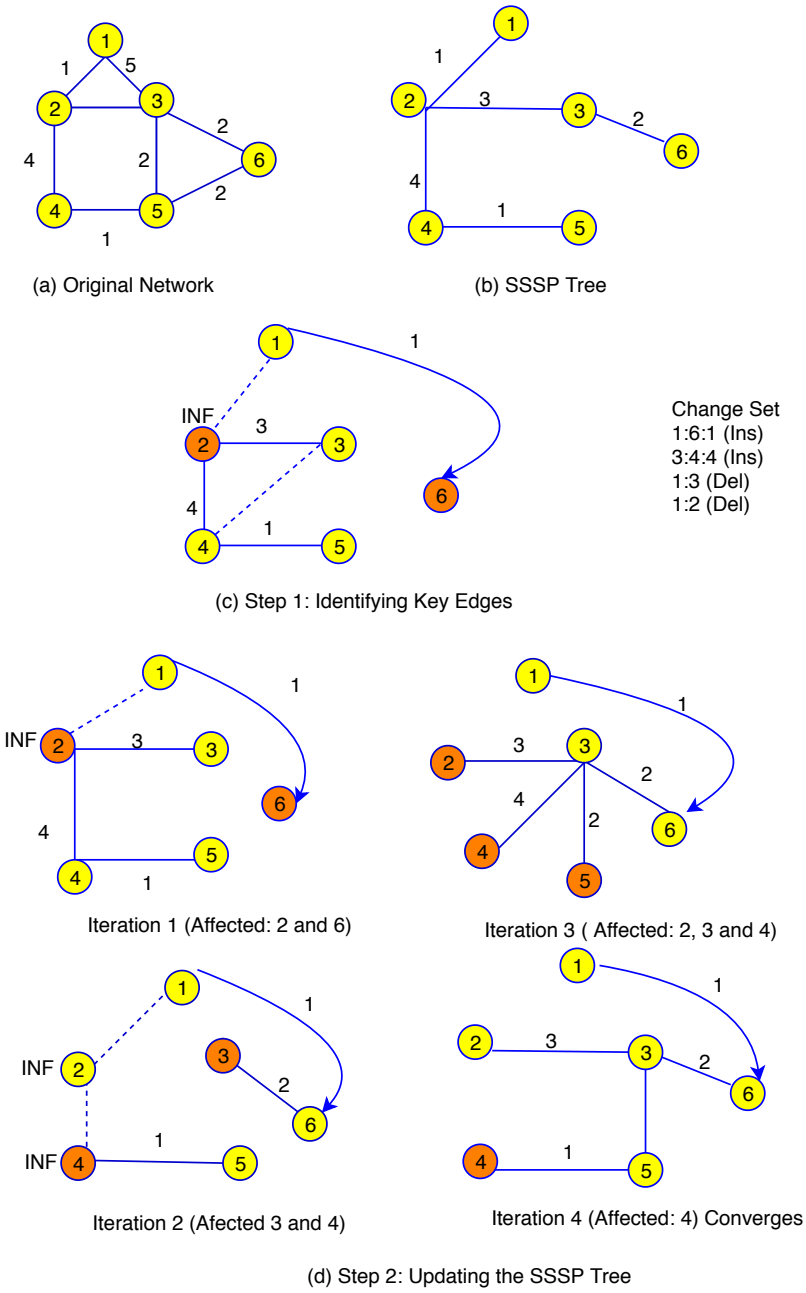

Figure 4.2: SSSP Algorithm
[58]

4.3 Experimental Results

Figures 4.3, and 4.4 show the parallel scaling of the proposed algorithm on the two synthetic and three real-world networks. The time for updating the shortest path with a 50-million-edge update which consist of different fractions of edge insertions and deletions.x% insertions means that the dataset contains the x% edge insertions and $(100-x)$% edge deletions [58]. For all the experiments in this dissertation time is used as a measurement instead of TEPS (Traversed Edges Per Second). The reason for not using TEPS is because it is not the correct metric for measuring performance for dynamic networks, ideal goal of this dissertation is to traverse less edges.

The results demonstrate that the proposed approach is scalable for almost all the networks used in this dissertation, however, the scalability decreases as the number of threads increases, the reason could be amount of work decreases as thread shrinks. In general it is observed that the random (ER) graphs are more scalabale when compared to the scale free graphs.

In the case of real-world graphs, time for updating SSSP increases for processing deletions as compared to the synthetic graphs. In general the real-world graphs selected for this dissertation are smaller and the percentage of affected vertices are higher.

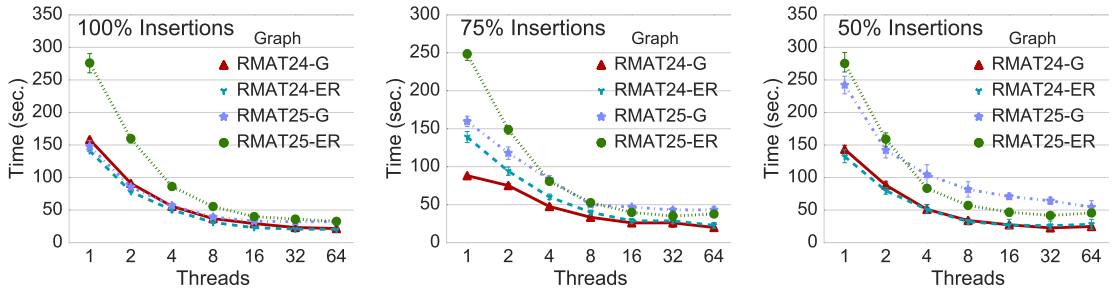

Figure 4.3: Scalability of shared-memory parallel SSSP computation with two RMAT (synthetic) graphs for 50 million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right). [1]

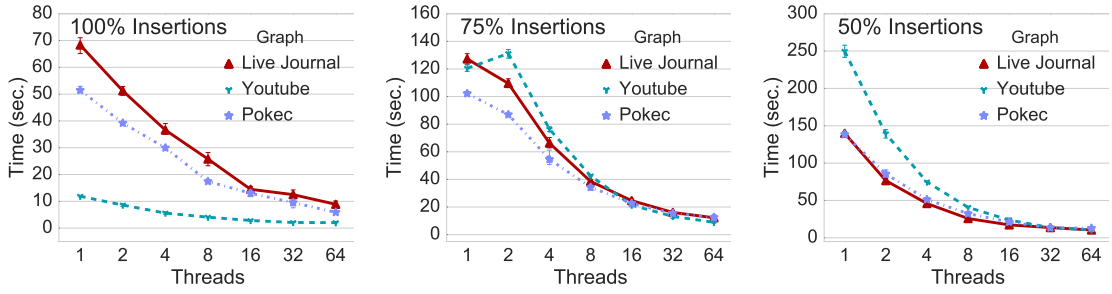

Figure 4.4: Scalability of shared-memory parallel SSSP computation for three real-world graphs, for 50 Million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

4.3.1 Impact of Selecting Source Vertex

For all the experiments demonstrated in 4.3, and 4.4 initial vertex 0 is considered as a source vertex. In order to study the impact of selection of source vertex on scalability, two other source vertices are selected at random from the two of the graphs and then the computation time for updating 100% insertion is measured. Figure 4.5, gives the scalability of one real-world graph and synthetic graph with different source vertex. The timings were almost identical when compared to the conventional source vertex 0 for the same network. The selection of source vertex does not signifcantly affect the scalability.

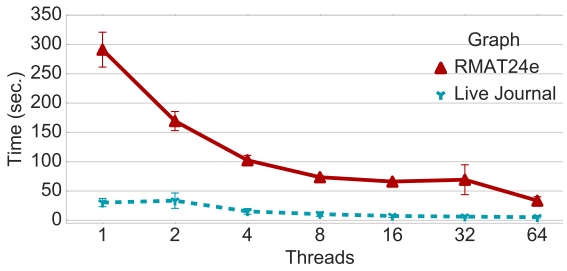

Figure 4.5: Scalability of shared-memory parallel SSSP computation with two RMAT (synthetic) graphs for 50 million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

4.3.2 Addition and Deletion of Vertices

The proposed approach can be extended to addition and deletion of vertices. In order to extend the data structure which is an adjacency list, maintains a buffer space for adding new vertices. Once a new vertex is added, edges are inserted in the row of the adjacency list. For vertex or edge deletion, software marks them for deletion performing a soft delete. Figure 4.6 demonstrates strong scaling results for adding and deleting vertices from RMAT24, ER and G. When compared to only edge changes the scalability is better for random graphs, than the scale free.

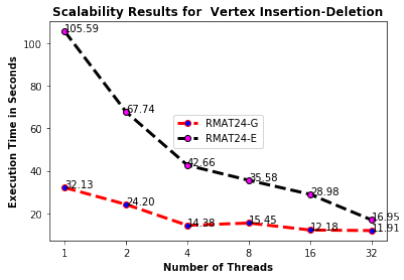

Figure 4.6: Scalability of adding and deleting vertices. RMAT24-ER,G networks with 10 million edges changed, 100 vertices inserted and 50 vertices deleted.

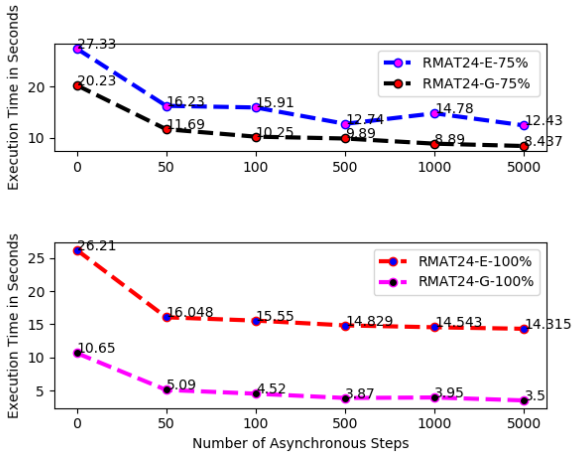

Figure 4.7: Change in execution time with increased levels of asynchrony on 8 threads. X-axis: level of asynchrony. Y-axis: time to update SSSP. Higher asynchronous levels lead to lower time.

4.3.3 Increasing Asynchrony

In this section effect of asynchrony is studied. The level of asynchrony is defined as one minus the length of path traversed before synchronization at the for loop. Level 0 means only neighbors are processed, Level 1 means that distance 2 neighbors are processed. RMAT24 G and ER graphs with 10 M edges with 75 % and 100 % insertions. The level of asynchrony for the experiments are varied by 0, 50, 100, 500 and 1K and 5K. Figure 4.7 shows the execution time with increasing asynchrony for 8 threads. Time decreases sharply as the level of asynchrony increases from 0 to 50. Figure 4.8 compares the percentage of updates, change with increasing the level of asynchrony with respect to the updates for the synchronous method on 8 threads. For all the cases the number of updates increase, which indicates the increase of redudant computation.

Figure 4.9 gives the scalability of the synchronous update to asynchrony of 50k. Asynchronous around level 50 k is scalable and requires less time. The results demonstrates that the tuning the level of asynchrony can lead to faster and scalable update of the SSSP tree.

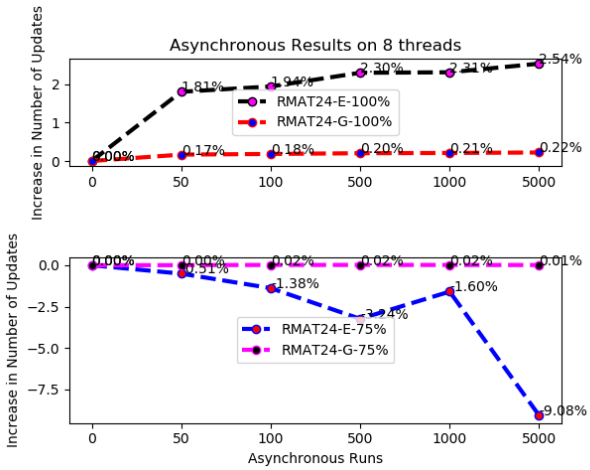

Figure 4.8: Percentage change in number of updates of vertex values with respect to updates for the synchronous case (set at 0) on 8 threads. X-axis: level of asynchrony. Y-axis: percentage change in updates.Asynchrony, in general, leads to more updates of the vertices.

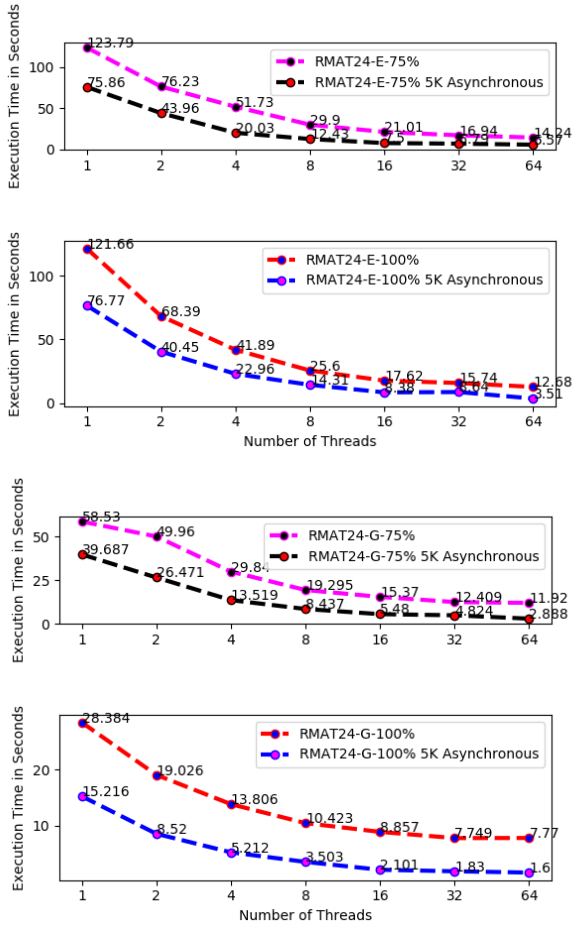

Figure 4.9: Comparison of scalability of synchronous and asynchronous (level 5000) updates. Asynchronous updates are faster and equally scalable.

4.4 Comparison with Galois

In order to compare the performance of the proposed approach, author compares the time taken to recompute the shortest path from scratch using Galois 2.2.1. Galois is very fast compared to other parallel network packages. Galois uses iterative Δ-stepping [59] to compute SSSP with non-negative edge weights. Figure 4.10 shows the execution time of the proposed approach with comparison to Galois- static on the RMAT24-G and RMAT24-ER graphs. Table 4.1 summarizes the improvement of proposed approach over Galois.The improvement was computed by averaging the times from four experiments for each parameter. In all cases, the proposed approach is faster than the recomputation.

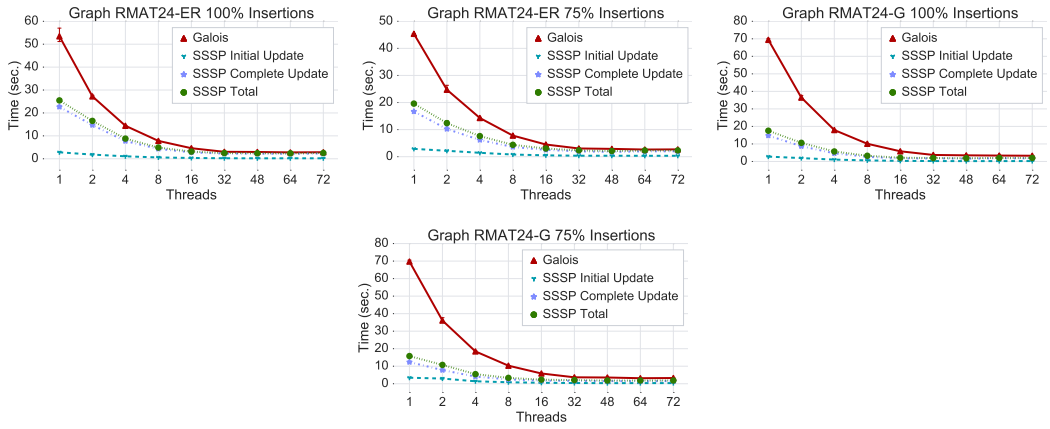

Figure 4.10: Scalability of shared-memory parallel SSSP computation with two RMAT (synthetic) graphs for 50 million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

4.5 Correctness of the Algorithm

Consider a graph G, with positive edge weights, that is modified to graph G_u due to changes in its structure. We will prove that our parallel updating algorithm, as given by Algorithms 7 and 8, will produce a valid SSSP tree for G_u.

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

Proof. To prove this, we first create a directed graph where the parent vertices point to their children. Now consider a path in this graph between any two vertices a and b. The path goes from vertex a to vertex b. This means that a is an ancestor of b. As per the algorithm for creating the original SSSP tree (Algorithm **??**), and the algorithms for parallel update (Algorithms 7 and 8), a vertex

Table 4.1: Improvement of update algorithm over Galois's static algorithm for a 50 million-edge update with 100% and 75% insertions on the RMAT24-ER and RMAT24-G graphs.

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

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. □

The tree obtained by our parallel algorithm will be a valid SSSP tree for G_u.

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

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

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

Consider the case where b=c. We know that the $Dist_{T_u}[b] = Dist_{T_{alg}}[b]$. Also, per Algorithms **??**, 7, 8, the distance of a child node is the distance of its parent plus the weight of the connecting edge. Therefore, $Dist_{T_{alg}}[a] = Dist_{T_{alg}}[b] + W(a,b) = Dist_{T_u}[b] + W(a,b) = Dist_{T_u}[a]$.

Now consider when $b \neq c$. Since the edge (b,a) exists in T_u, it also exists in G_u. Since $Dist_{T_u}[v] \neq$

$Dist_{T_{alg}}[v]$, the distance of a was updated either in T_u or in T_{alg}, or in both, from the original SSSP tree. Any of these cases imply that a was part of an affected subgraph. Therefore, at some iteration(s) in Step 2, a was marked as an affected vertex.

Because the edge (b, a) exists in G and a is an affected vertex, in Step 2 (Algorithm 8, lines 13:21), the current distance of a would have been compared with $Dist_{T_{alg}}[b] + W(a, b)$. Since this is the lowest distance of a according to the known SSSP tree T_u, either the current distance would have been updated to the value of $Dist_{T_{alg}}[b] + W(a, b)$ or its was already equal to the value. Therefore, $Dist_{T_u}[a] = Dist_{T_{alg}}[a]$. □

Chapter 5

Dissertation Proposal

5.1 Proposed Work

The expected contributions of the dissertation research are:

1. *Updating Strongly Connected Components:* The shared memory implementation for updating SSSP, graph connectivity, and Minimum Spanning Tree (MST) on dynamic networks have shown strong scalability. Based on the results mentioned in the previous section, author proposes to extend the existing software and update few well known network properties. One of the network properties of interest is Strongly Connected Components (SCC).

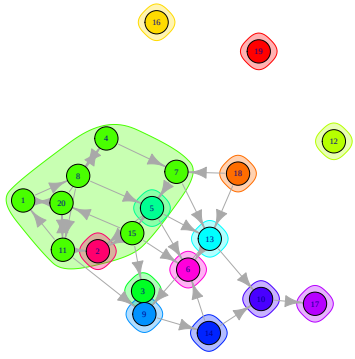

Figure 5.1: Strongly Connected Components

Figure 5.1 is an example of Strongly Connected Components (SCC). SCC is a group of vertices,

if there is a path between all pairs of vertices. In the above example there is one SCC. The group of vertices which are colored as green and clustered together is the SCC. There are many shared memory implementations of finding SCC such as [56], [6]. These implementations are not focused on dynamic networks. The author proposes to extend the proposed software in this dissertation to update the SCC on the dynamic networks.

2. *Shared Memory Implementation of Overlapping Community Detection:*

Networks are said to exhibit community structure, if the nodes can be grouped such that group is densely connected internally and sparsely connected externally with other groups. Real World networks exhibit community structure.

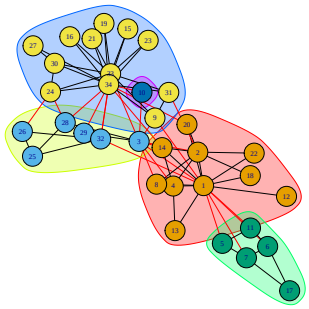

Figure 5.2: Singleton Communities

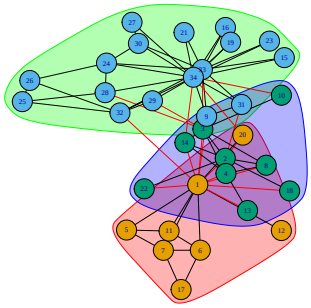

Figure 5.3: Overlapping Communities

Figure 5.2 shows an example for singleton communities. In singleton communities each node in the network belongs to only one community. Figure 5.3 is an example for overlapping

communities. There are chances that a given node in the network can belong to the multiple communities. There are couple of shared memory and distributed implementation of detecting singleton communities [60,61]. In the case of overlapping communities there exist a distributed memory implementation of SLPA [62]. In earlier research [60] vertex-centric metric called "permanence" was proposed for overlapping communities. Permanence ranges from -1 to 1, where -1 means vertex is assigned to a wrong community, however +1 means vertex is assigned to correct community. This is a quantitative measure and gives a idea that how much a vertex belongs to its community. GenPerm proposed in [63] is designed for overlapping community analysis. The author proposes to study the performance such as the scalability, and the community structure of shared memory implementation of overlapping communities using a metric GenPerm [63].

3. *Extending the Dynamic Network Algorithms to GPUs:*

   Shared memory implementation of MST, connected components, and SSSP show limited scalability on few networks. Graphical Processing Units (GPUs) provide more computing power and memory bandwidth than CPUs [64]. GPUs can be a good candidate for updating the networks properties of dynamic network. Recently there has been plethora of graph algorithms that have been implemented on GPUs such as [64] . Sha et al. [65] has an implementation of Dynamic Graph analytics on GPUs, however there software allows only three network properties such as BFS, connected component and PageRank. The author proposes to extend connected components on GPUs, study the scalability, and compare the performance with sha et al. [65] .

Bibliography

[1] 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 *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pp. 245–254, Dec 2018.

[2] S. Roy, M. Xue, and S. K. Das, "Security and discoverability of spread dynamics in cyber-physical networks," *IEEE Transaction on Parallel and Distributed Systems*, vol. 23(9), p. 16941707, 2012.

[3] F. Restuccia and S. K. Das, "Fides: A trust-based framework for secure user incentivization in participatory sensing," *Proceedings of 15th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2014.

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

[5] S. J. Plimpton and T. Shead, "Streaming data analytics via message passing with application to graph algorithms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 8, pp. 2687 – 2698, 2014.

[6] Y. Ji, H. Liu, and H. H. Huang, "ispan: parallel identification of strongly connected components with spanning trees," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, p. 58, IEEE Press, 2018.

[7] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–2, Sept 2013.

[8] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 135–146, 2010.

[9] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "GoFFish: A Sub-Graph Centric Framework for Large-Scale Graph Analytics," *arXiv:1311.5949 [cs]*, vol. 8632, pp. 451–462, 2014. arXiv: 1311.5949.

[10] G. Cattaneo, P. Faruolo, U. F. Petrillo, and G. Italiano, "Maintaining dynamic minimum spanning trees: An experimental study," *Discrete Applied Mathematics*, vol. 158, pp. 404–425, Mar. 2010.

[11] T. Hayashi, T. Akiba, and Y. Yoshida, "Fully dynamic betweenness centrality maintenance on massive networks," vol. 9, no. 2, pp. 48–59.

[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI 14, (Broomfield, CO, USA), pp. 599–613, 2014.

[13] D. Gregor, N. Edmonds, B. Barret, and A. Lumsdaine, "The parallel boost graph library," 2005. *Available at* `http://www.osl.iu.edu/research/pbgl`.

[14] "Knowledge Discovery Toolkit." `http://kdt.sourceforge.net/`, 2014.

[15] "Apache Incubator Giraph." `http://incubator.apache.org/giraph`. Accessed on 2018-03-37.

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

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

[18] C. Staudt, A. Sazonovs, and H. Meyerhenke, "Networkit: An interactive tool suite for high-performance network analysis," *CoRR*, vol. abs/1403.3005, 2014.

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, (Washington, DC, USA), pp. 229–238, IEEE Computer Society, 2009.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new parallel framework for machine learning," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[21] G. Karypis and V. Kumar, "Metisa software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing ordering of sparse matrices," 01 1997.

[22] K. Devine, B. Hendrickson, E. Boman, M. S. John, and C. Vaughan, "Design of dynamic load-balancing tools for parallel applications," in *Proc. Intl. Conf. on Supercomputing*, (Santa Fe, New Mexico), pp. 110–118, 2000.

[23] C. Chevalier and F. Pellegrini, "PT-SCOTCH: A tool for efficient parallel graph ordering," *Parallel Computing*, vol. 34, no. 6-8, pp. 318–331, 2008.

[24] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *J. Parallel Distrib. Comput.*, vol. 48, pp. 71–95, Jan. 1998.

[25] "Stinger-streaming graph analytics." `http://www.stingergraph.com/`, 2015.

[26] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarra-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, "Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation," *Georgia Institute of Technology, Tech. Rep*, 2009.

[27] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *In SDM*, 2004.

[28] A. Yoo, E. Chow, K. Henderson, W. Mclendon, B. Hendrickson, and mit atalyrek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *In SC 05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 25, IEEE Computer Society, 2005.

[29] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[30] 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.

[31] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, DAC '10, (New York, NY, USA), pp. 52–55, ACM, 2010.

[32] D. A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps)," *J. Parallel Distrib. Comput.*, vol. 65, pp. 994–1006, Sept. 2005.

[33] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, "Fast minimum spanning tree for large graphs on the gpu," in *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, (New York, NY, USA), pp. 167–171, ACM, 2009.

[34] D. Eppstein, Z. Galil, and G. F. Italiano, "Dynamic graph algorithms," in *Algorithms and Theory of Computation Handbook* (M. J. Atallah, ed.), ch. 8, CRC Press, 1999.

[35] S. Das and P. Ferragina, "An erew pram algorithm for updating minimum spanning trees," vol. 9, pp. 111–122, 1999.

[36] S. Srinivasan, S. Bhowmick, and S. Das, "Application of Graph Sparsification in Developing Parallel Algorithms for Updating Connected Components," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 885–891, May 2016.

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

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

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

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

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

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

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

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

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

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

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

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

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

[50] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[51] L. K. Fleischer, B. Hendrickson, and A. Pınar, "On identifying strongly connected components in parallel," in *International Parallel and Distributed Processing Symposium*, pp. 505–511, Springer, 2000.

[52] W. Mclendon Iii, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 901–910, 2005.

[53] J. Barnat, P. Bauch, L. Brim, and M. Ceška, "Computing strongly connected components in parallel on cuda," in *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 544–555, IEEE, 2011.

[54] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 92, ACM, 2013.

[55] X. Chen, C. Chen, J. Shen, J. Fang, T. Tang, C. Yang, and Z. Wang, "Orchestrating parallel detection of strongly connected components on gpus," *Parallel Computing*, vol. 78, pp. 101–114, 2018.

[56] G. M. Slota and K. Madduri, "Parallel color-coding," *Parallel Computing*, vol. 47, pp. 51 – 69, 2015. Graph analysis for scientific discovery.

[57] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[58] DynamicSSSP, "HIPC18 Parallel Dynamic SSSP. Contribute to DynamicSSSP/HIPC18 development by creating an account on GitHub," Dec. 2018. original-date: 2018-03-29T03:04:36Z.

[59] U. Meyer and P. Sanders, "-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114 – 152, 2003.

[60] T. Chakraborty, S. Srinivasan, N. Ganguly, A. Mukherjee, and S. Bhowmick, "Permanence and Community Structure in Complex Networks," *arXiv:1606.01543 [physics]*, June 2016. arXiv: 1606.01543.

[61] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, Oct. 2008. arXiv: 0805.4770.

[62] J. Xie, B. K. Szymanski, and X. Liu, "SLPA: Uncovering Overlapping Communities in Social Networks via A Speaker-listener Interaction Dynamic Process," *arXiv:1109.5720 [physics]*, Sept. 2011. arXiv: 1109.5720.

[63] T. Chakraborty, S. Kumar, N. Ganguly, A. Mukherjee, and S. Bhowmick, "GenPerm: A Unified Method for Detecting Non-overlapping and Overlapping Communities," 2016.

[64] Y. Pan, R. Pearce, and J. D. Owens, "Scalable Breadth-First Search on a GPU Cluster," *arXiv:1803.03922 [cs]*, Mar. 2018. arXiv: 1803.03922.

[65] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating Dynamic Graph Analytics on GPUs," p. 14.