Scalable Algorithms for Updating Large-Scale 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: Information Technology

Under the Supervision of Dr. Sanjukta Bhowmick

Omaha, Nebraska

November, 2020

Supervisory Committee:

Dr. Sanjukta Bhowmick

Dr. Yuliya Lierler

Dr. Kathryn Cooper

Dr. Mahantesh Halappanavar

Scalable Algorithms for Updating Large-Scale Dynamic Networks

Sriram Srinivasan, (Ph.D.)

University of Nebraska, 2020

Advisor: Dr. Sanjukta Bhowmick

The growth of social media and data in various domains increased the interest in 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, Google's Page Rank algorithm, and iSpan algorithm for detecting strongly connected components are designed and optimized for static networks. For the networks that change with time, i.e. the dynamic networks (such as social networks, biological networks, or temporal networks) the above-mentioned approaches can only be utilized if they are computed from scratch each time. Performing a 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 GPU which updates network properties such as Connected Components (CC), Minimum Spanning Tree (MST), Single Source Shortest Path (SSSP), Page Rank (PR), 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(CPU), and Gunrock (GPU). The shared memory implementation processes over 50 million updates for updating SSSP on a real-world network in under 300 seconds (GPU).

This dissertation also provides a novel shared memory implementation of detecting, overlapping, and non-overlapping communities on static networks using Permanence. Detecting communities on large scale networks is a fundamental operation in various domains. Detecting correct communities is a challenging problem due to the limitations of the metric such as the state of the art metric modularity since it suffers from the resolution limit. This dissertation is the first attempt to implement shared memory overlapping and non-overlapping communities using permanence. The key attributes of this implementation are the accuracy of the communities when compared to the ground truth and achieve speed up to 10 x times when compared to its sequential implementation.

The dissertation concludes with a summarization of the contributions and their improvement in large-scale network analytics and a discussion about future work in 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.

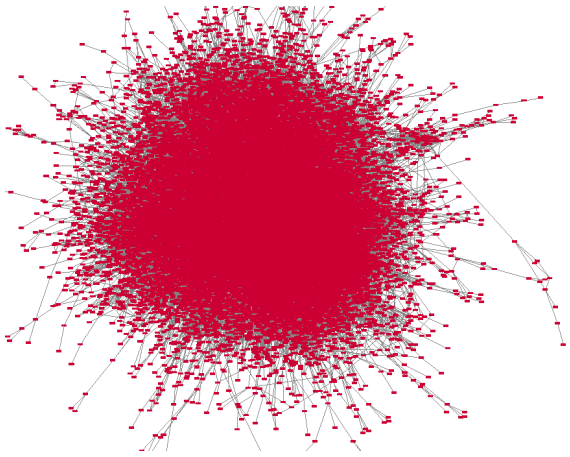


Figure 1.1: Citation Network

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.

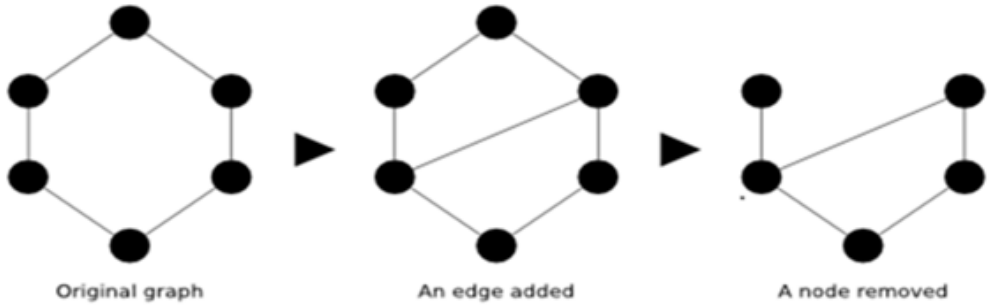


Figure 1.2: Dynamic Network, given a original network at time T_0, later a edge is added, and a node is removed from the original network.

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** In this chapter author proposes a GPU implementation to update the Single Source Shortest Path (SSSP). The chapter includes the algorithm, and scalability results for real-world and synthetic networks. Along with the scalability results, there is a comparison provided for speedup between proposed GPU implementation and parallel implementation presented in chapter 4.

- **Chapter 6**

  Here the author presents a shared memory implementation to detect overlapping communities in large scale networks using Permanence. The chapter includes detailed discussion about the proposed algorithm, scalability results, and community quality comparison with ground truth.

- **Chapter 7**

  This chapter introduces a shared memory implementation of updating Page Rank for dynamic networks. The chapter presents the algorithm, scalability results, and discussion about the results. As the proposed approach is expensive, a novel adaptive approach is introduced towards the end of the chapter which allows to run the state-of the art Page Rank recomputing approach and the proposed dynamic algorithm based on the set of changed edges.

- **Chapter 8**

  Here the author presents a shared memory implementation to detect Strongly Connected Components (SCC's) in large scale networks. The chapter includes detaled discussion about the proposed algorithm, scalability results, and the limitations the proposed implementation for detecting SCC's.

- **Chapter 9**

  This chapter provides a detailed overview of the future work which author proposes to extend this 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/Updating 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/Updating 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/Updating 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.

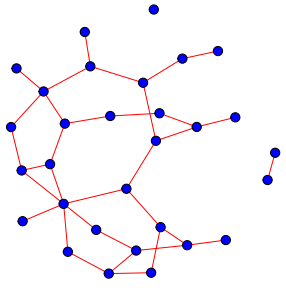


Figure 3.1: Example of Connected Components

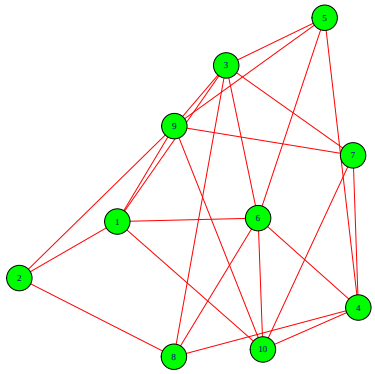


Figure 3.2: Toy Network

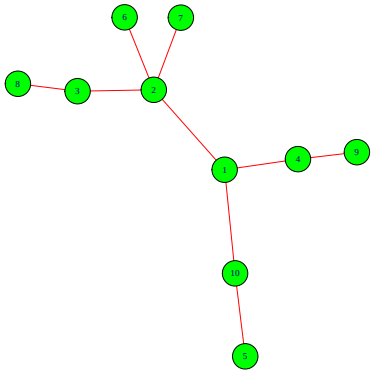


Figure 3.3: Example of Minimum Spanning Tree (MST) of the above Toy Network Figure 3.2

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                                    */
for i ← 0 to |V| do
    Root[i] = i
end
/* Set List of Output Edges to Null                                   */
E_x ← ∅
/* Check for all Edges                                                */
for i ← 0 to |E| do
    this_edge ← E[i]
    v and u are endpoints of this_edge
    /* Find the root of v                                             */
    root_v = v
    while Root[root_v] ≠ root_v do
        root_v = Root[root_v]
    end
    /* Find the root of u                                             */
    root_u = u
    while Root[root_u] ≠ root_u do
        root_u = Root[root_u]
    end
    /* If the endpoints are in separate components, join them         */
    if root_v ≠ root_u then
        /* Add to Key Edges                                           */
        E_x ← E_x ∪ this_edge
        min ← min(root_v, root_u)
        Root[root_v] ← min
        Root[root_u] ← min
    end
end
```

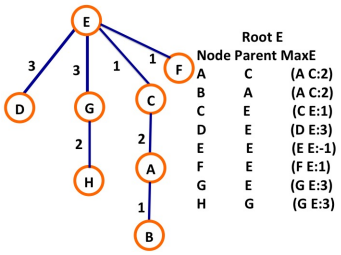


Figure 3.4: Rooted Tree

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.

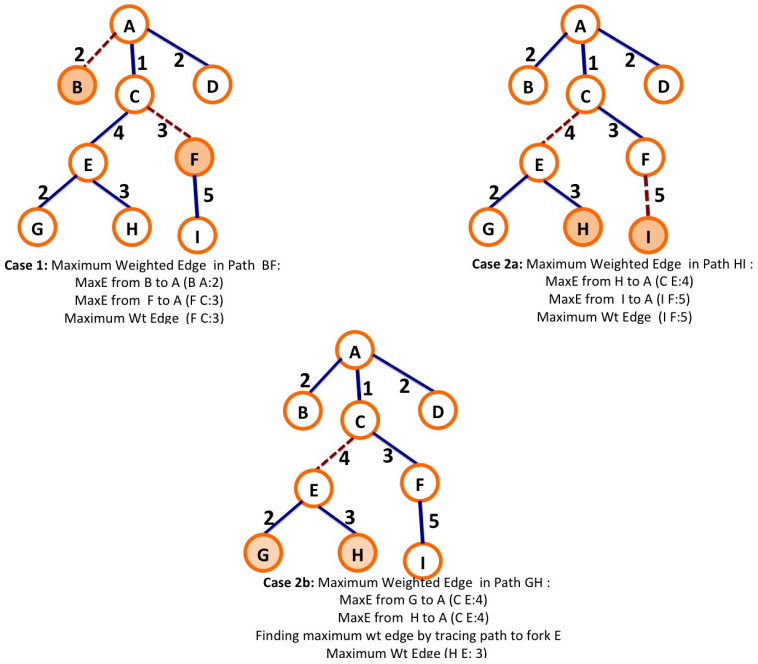


Figure 3.5: Three cases by which to find the maximum weighted edge in a path. The colored nodes are the endpoints of the path. The dashed red lines are the maximum weighted edge from the nodes to the root A.

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.

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

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

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

Algorithm 4: Step 2: Classifying the Changed Edges

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

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

```
1  Function Process_Status(E_x, E_r, CE, Status, Marked, RootedT)
       Input  : Changed Edge Set, CE; Status of Edges, Status; Marking Edges to be
                Replaced, Marked.
       Output: Set E_x of Key Edges; Set E_R of Changed Edges; Rooted Tree, RootedT
2      for i = 0 to |CE| do in parallel
           /* Get Edge and its Status                                          */
3          E ← CE[i]
4          S ← Status[i]
           /* Deleting Key Edge                                                */
5          if S = DEL then
6              Delete Edge E from Appropriate Edge Set
7              ** What does "Appropriate" mean here ** Assign Weight of Edge E to
               INF, i.e., very high value to mark it as deleted
8          end
           /* Edge into Remainder Edges                                        */
9          if S = NONE then
10             Add Edge E to the Remainder Edge Set
11         end
           /* Edge into Disconnected Tree                                      */
12         if S = INS then
13             Add Edge E to the Key Edge Set
14         end
           /* Replacing Edge from Key Edge                                     */
15         if S = RPL then
               /* Find Edge to be Replaced                                     */
16             E_rpl ← Marked[i]
               /* Replacing Edge Matches Current Inserting Edge                */
17             if E_rpl.Rpl_Id = i then
18                 Add Edge ** Which Edge ? ** to the Key Edge Set, E_x
19                 Set Weight of E_rpl to −1 to mark it as deleted
20             end
21             else
                   /* Replacing Edge Does Bot Match                            */
22                 Add E to set of new changed edges
23             end
24         end
25     end
26     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 5.1 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) [57] 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

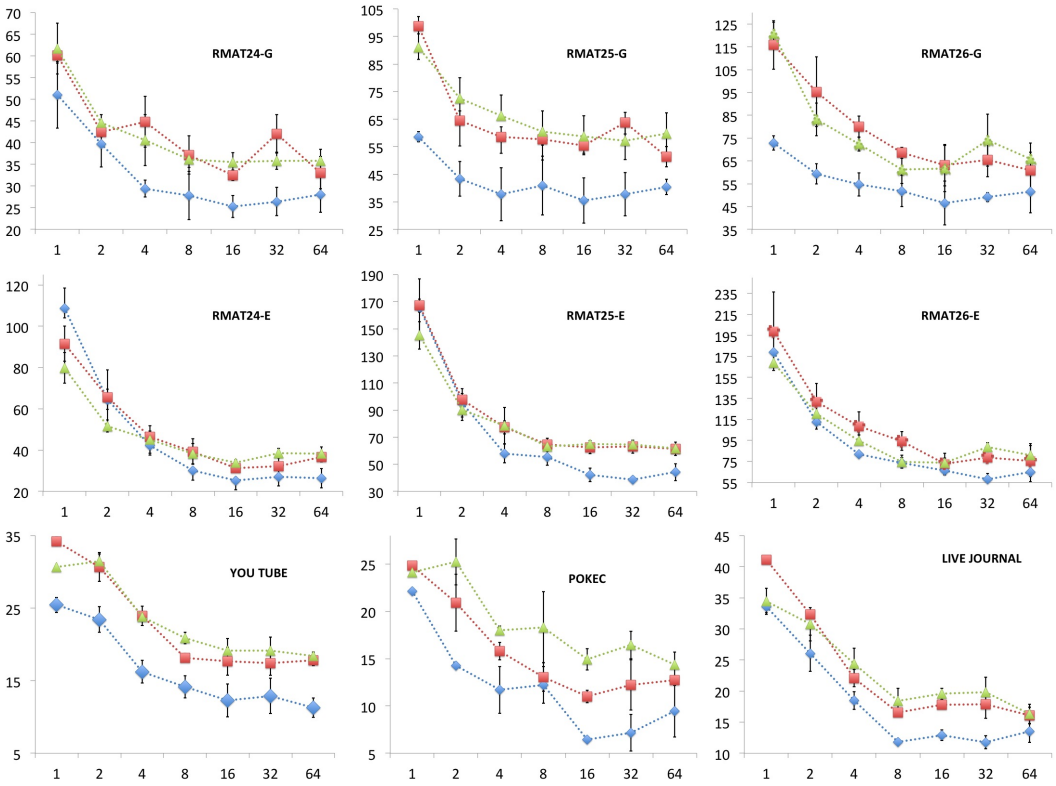


Figure 3.6: Scalability Results for Updating Networks. Top: Networks with scale-free degree distribution of order of 2^{24}, 2^{25}, 2^{26} vertices. Middle: Random Networks with normal degree distribution of order of 2^{24}, 2^{25}, 2^{26} vertices. Bottom: Real-world Networks, left to right (YouTube, 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).

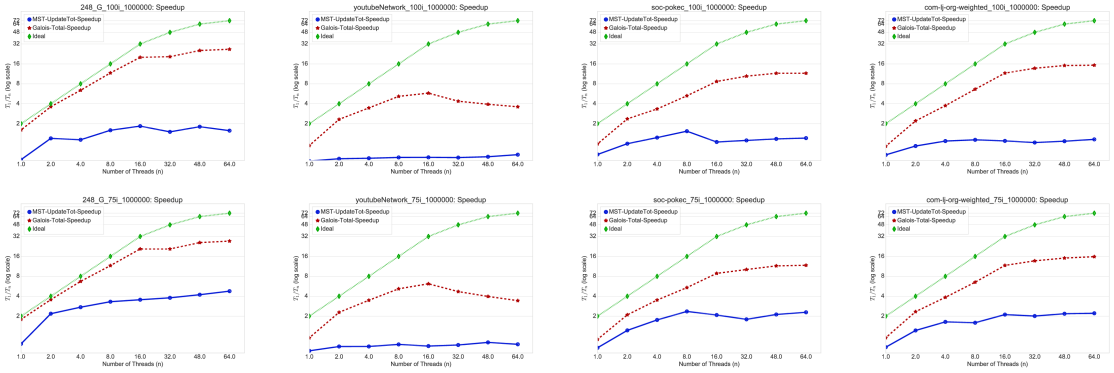


Figure 3.7: Parallel speedup (log scale) for computing the MST of RMAT-24G and three real-world networks (described in Sec **??**). Speedup was computed based on one batch update, with batch size 1,000,000 edges. Top: 100% insertions. Bottom: 75% insertions.

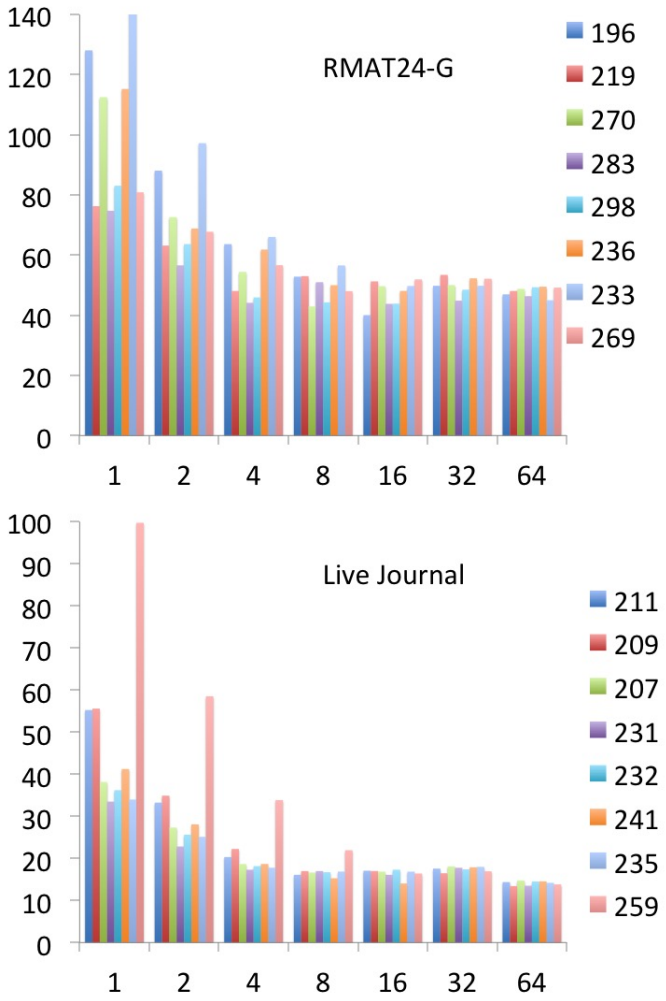


Figure 3.8: Variations of updating time based on the choice of root. Left: RMAT24-G. Right: LiveJournal. The Y-axis gives the time in seconds. The X-axis gives the number of threads. Each colored bar represents a tree of different height. The heights are given in the legend. The results are for 100%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

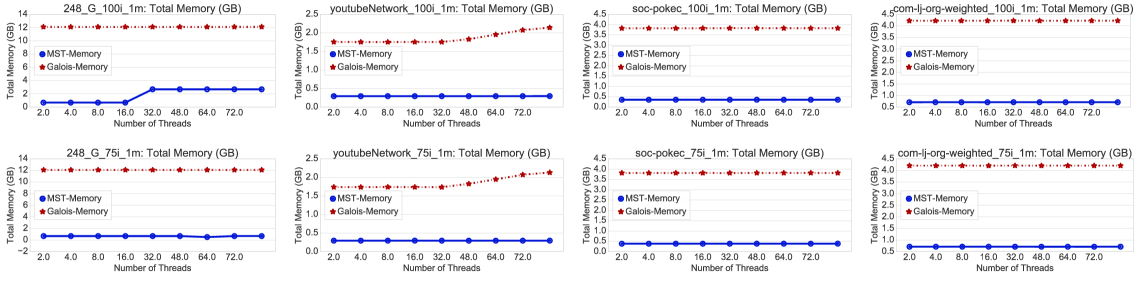


Figure 3.9: Total memory use for computing the minimum spanning tree of RMAT-24G and three real-world networks (described in Sec **??**). Top: 100% insertions. Bottom: 75% insertions.

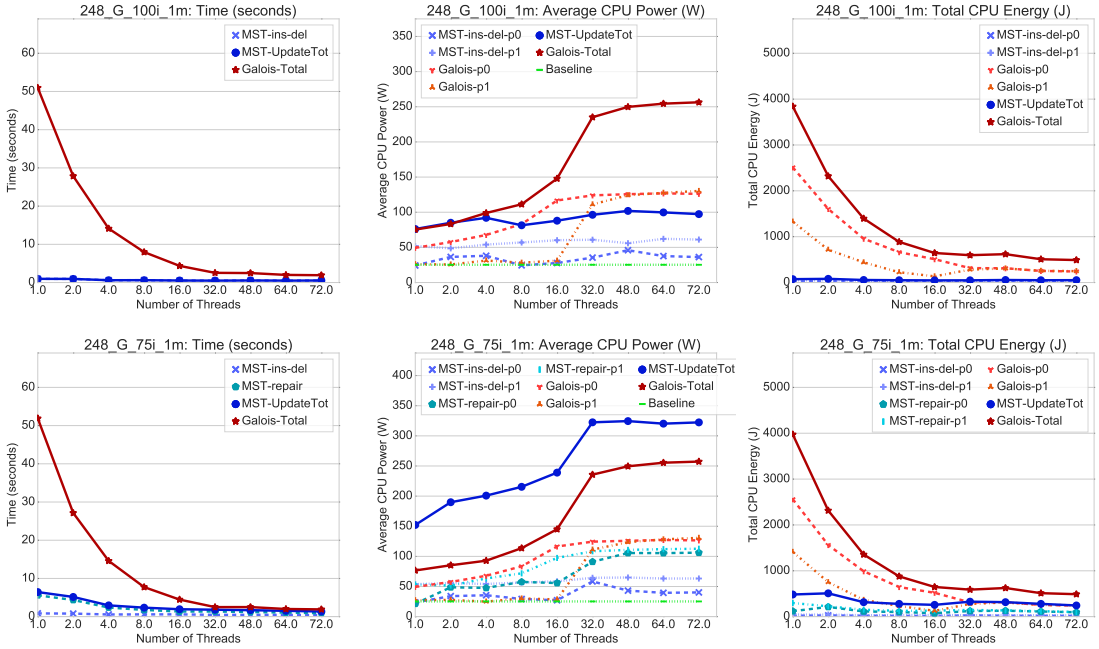


Figure 3.10: Comparison of per-socket and total power and energy measurements: a single batch update (blues), broken into "Insertion and Deletion and "Repair," compared with recomputing using Galois (reds) of the minimum weighted spanning tree. Top: 100% insertions Bottom: 75% insertions. Left: runtime.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

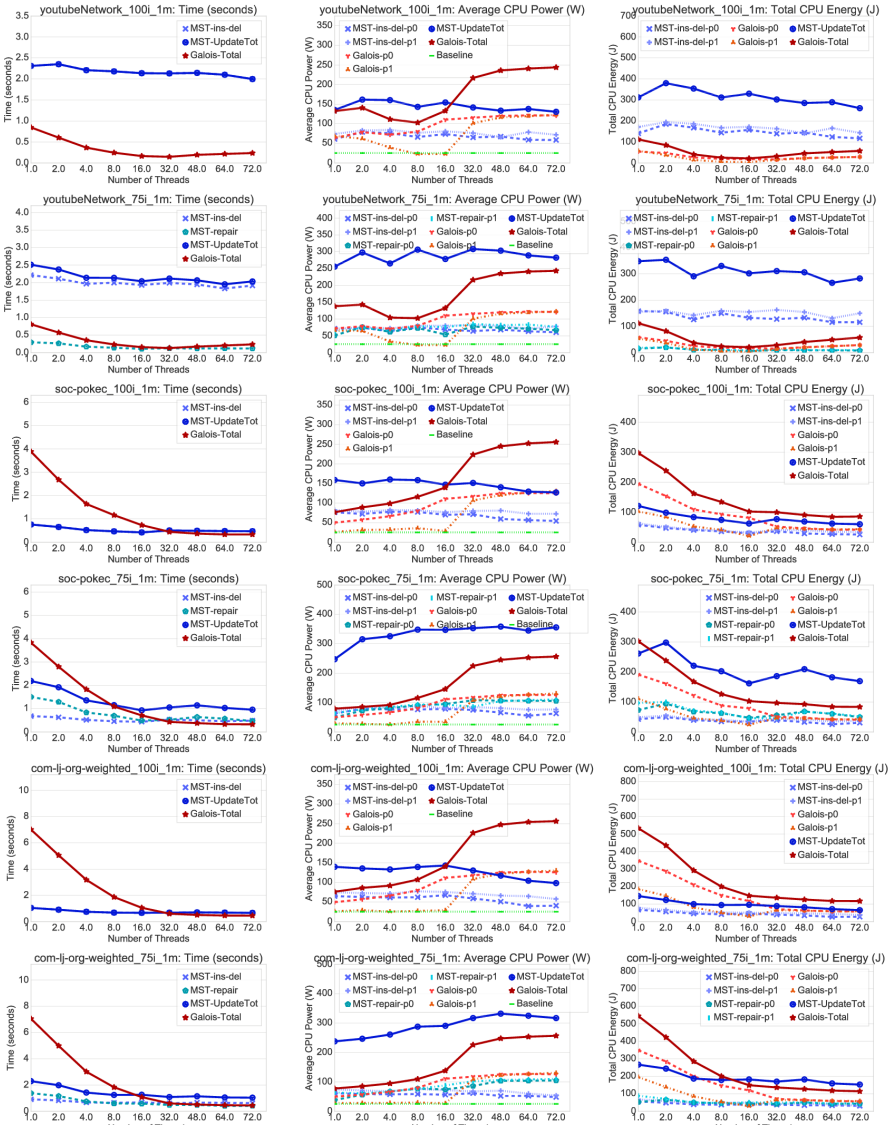


Figure 3.11: Real-world network comparison of per-socket and total power measurements: a single batch update (blues), broken into "Insertion and Deletion" and "Repair," compared with recomputing using Galois (reds) of the minimum weight spanning tree. Top two rows: YouTube (1.1M nodes, 3.0M edges). Middle 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 [58], and DSMR [38]. As mentioned in the introduction that these algorithms are good only for static networks, extending them to the dynamic

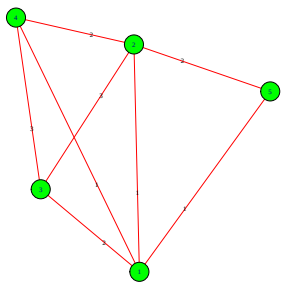


Figure 4.1: Sample Toy-Network

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 16 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 18): 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 16, 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 $Parent_u$ [1] Updating_per_Change$E$, G, T, $Dist$, $Parent$ Initialize Updated Distance and Parent
for $v \in V$ **do**
  $Dist_u[v] \leftarrow Dist[v]$ $Parent_u[v] \leftarrow Parent[v]$
end
Find the affected vertex, x
if $Dist_u[a] > Dist_u[b]$ **then**
  $x \leftarrow a$, $y \leftarrow b$
end
else
  $x \leftarrow b$, $y \leftarrow a$
end
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**
  v=$PQ.top()$ $PQ.dequeue()$
  Process_Vertex(v,G, T, $Dist_u$, $Parent_u$)
end
Process_Vertex$v$, G, T, $Dist_u$, $Parent_u$
for n *where* n *is neighbor of* v *in* G **do**
  if $Dist_u[v] = INF$ *AND* $Parent_u[n] = v$ **then**
    $Dist_u[n] = INF$ $PQ.enqueue(n)$
    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
    $Dist_u[v] > Dist_u[n]+W(v, n)$ $Dist_u[v] \leftarrow Dist_u[n] + W(v, n)$ $PQ.enqueue(v)$
    $Parent_u[v] \leftarrow n$
  end
end

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]
Updating_Batch_ChangeCE, G, T, $Dist$, $Parent$ Initialize Updated Distance and Parent
for $v \in V$ **do**
    /* in parallel */
    $Dist_u[v] \leftarrow Dist[v]$ $Parent_u[v] \leftarrow Parent[v]$ $Affected[v] \leftarrow False$
end
Find the affected vertex, x **if** $Dist_u[a] > Dist_u[b]$ **then**
    $x \leftarrow a$, $y \leftarrow b$
end
else
end
$x \leftarrow b$, $y \leftarrow a$
for *Each edge* $E(a, b) \in CE$ **do**
end
if *E to be inserted* **then**
    If E inserted, update $Dist$ and $Parent$ **if** $Dist_u[x] > Dist_u[y] + W(a, b)$ **then**
        $Dist_u[x] = Dist_u[y] + W(a, b)$ $Parent_u[x] = y$ Mark E as inserted to SSSP Tree $Affected[x] \leftarrow True$
    end
    else
        E to be deleted
    end
    Remove $E(a, b)$ from G Check if Edge in SSSP tree
    if $Parent[a] = b$ OR $Parent[b] = a$ **then**
        $Dist_u[x] = INF$ $Affected[x] \leftarrow True$
    end
end
Lowest Value Edge gets Inserted $Change \leftarrow True$ **while** $Change$ **do**
    $Change \leftarrow False$ **for** *Each edge* $E(a, b) \in CE$ **do**
        /* Process in parallel */
        if *E marked to be inserted to SSSP* **then**
            Find the affected vertex, x
            if $Dist_u[a] > Dist_u[b]$ **then**
                $x \leftarrow a$, $y \leftarrow b$ **else**
                end
                $x \leftarrow b$, $y \leftarrow a$
            end
            Check replaces higher weighted edge
            if $Dist_u[x] > Dist_u[y] + W(a, b)$ **then**
                $Dist_u[x] = Dist_u[y] + W(a, b)$ $Parent_u[x] = b$ $Change \leftarrow True$ $Affected[x] \leftarrow True$
            end
        end
    end
end

Algorithm 8: Step 2: Updating Affected Vertices in Parallel

```
1  Input: Weighted Graph G(V, E), Dist, Parent, Affected, SSSP Tree T. Weight of edge
     (u, v) is W(u, v).
2  Output: Updated Dist_u, Parent_u and SSSP Tree T_u based on structure of Parent_u [1]
3  Process_Vertex_Parallelv, G, T, Dist_u, Parent_u Change ← True
4  while Change do
5      Change ← False for v ∈ V do
           /* Perfoms in parallel                                              */
6          if Affected[v] = False then
7              Skip the vertex
8          end
9          Affected[v] ← False
10         for n where n is neighbor of v in G do
11             if Dist_u[v]=INF & Parent_u[n] = u then
12             end
13             Dist_u[n] = INF Affected[n] ← True
14             else
15             end
16             Dist_u[n] > Dist_u[v]+W(v, n)  Dist_u[n] ← Dist_u[v] + W(v, n)
               Affected[n] ← True Parent_u[n] ← v else
17             end
18             Dist_u[v] > Dist_u[n]+W(v, n) Dist_u[v] ← Dist_u[n] + W(v, n) Affected[v] ← True
               Parent_u[v] ← n
19         end
20     end
21 end
```

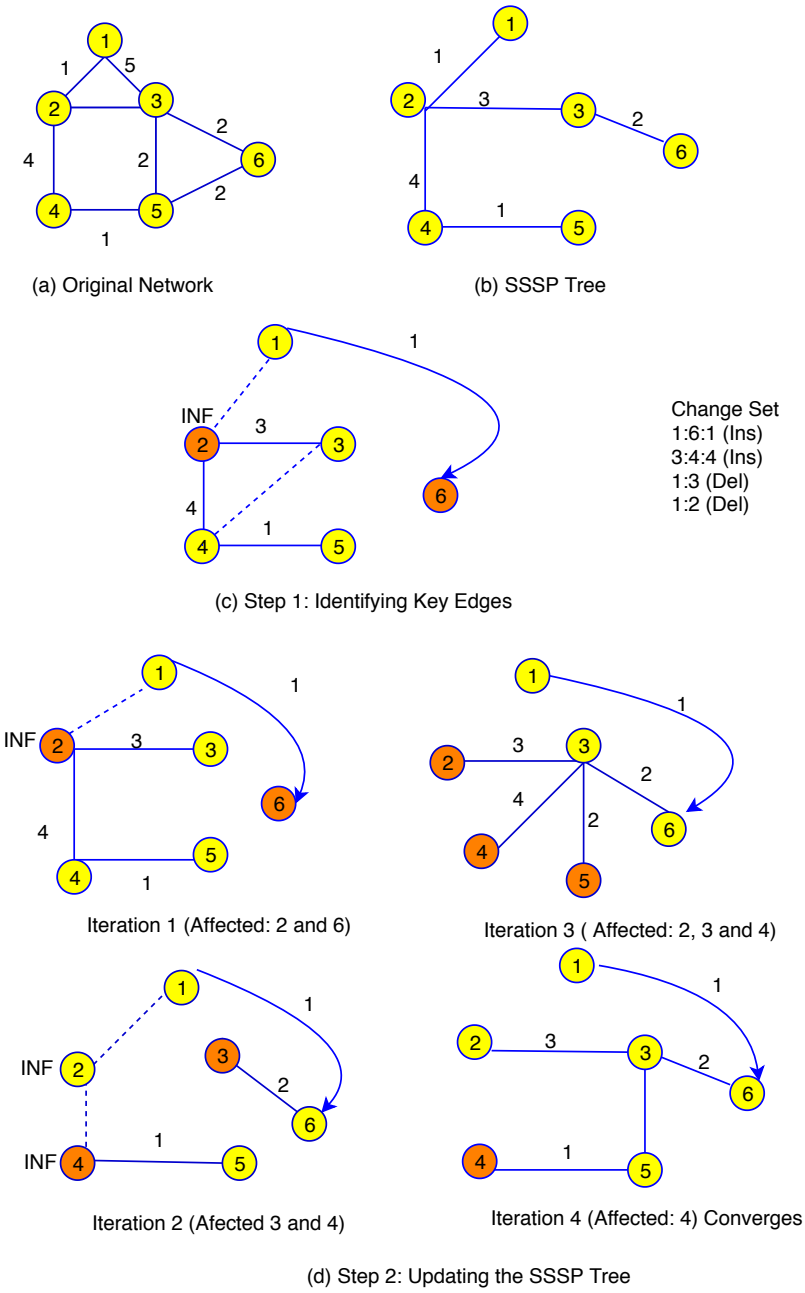


Figure 4.2: SSSP Algorithm
[59]

4.3 Experimental Results

Figures 5.3, 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 [59]. 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.

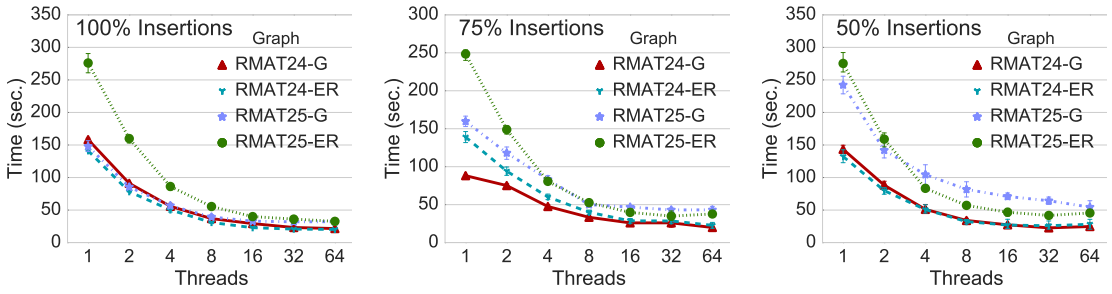


Figure 4.3: Scalability of shared-memory parallel SSSP computation with two RMAT (synthetic) graphs for 50 million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right). [1]

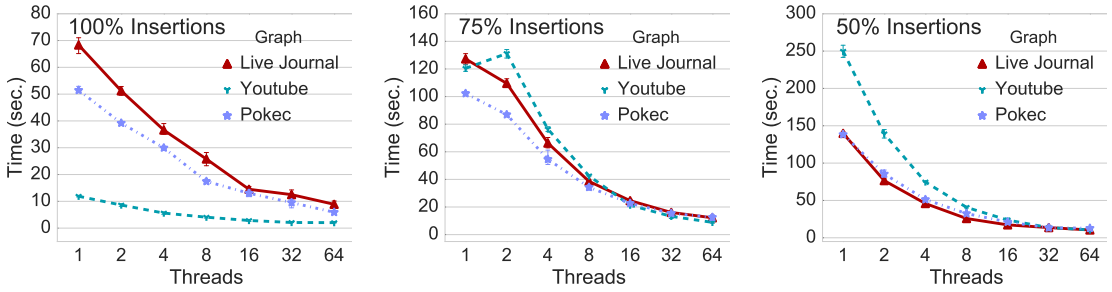


Figure 4.4: Scalability of shared-memory parallel SSSP computation for three real-world graphs, for 50 Million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

4.3.1 Impact of Selecting Source Vertex

For all the experiments demonstrated in 5.3, 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.

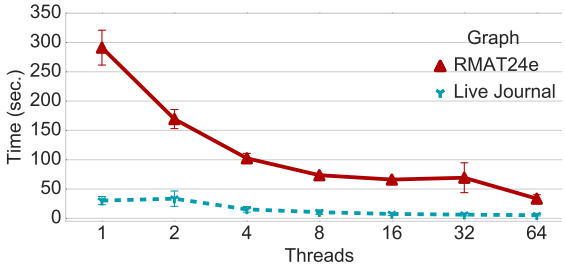


Figure 4.5: Scalability of shared-memory parallel SSSP computation with two RMAT (synthetic) graphs for 50 million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

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.

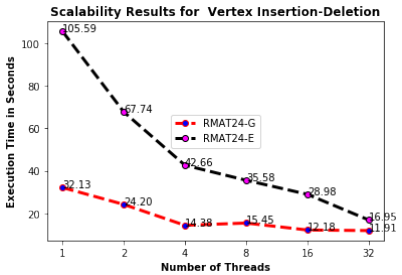


Figure 4.6: Scalability of adding and deleting vertices. RMAT24-ER,G networks with 10 million edges changed, 100 vertices inserted and 50 vertices deleted.

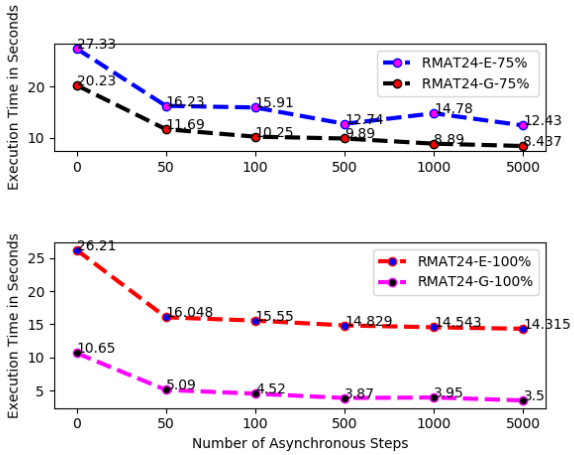


Figure 4.7: Change in execution time with increased levels of asynchrony on 8 threads. X-axis: level of asynchrony. Y-axis: time to update SSSP. Higher asynchronous levels lead to lower time.

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.

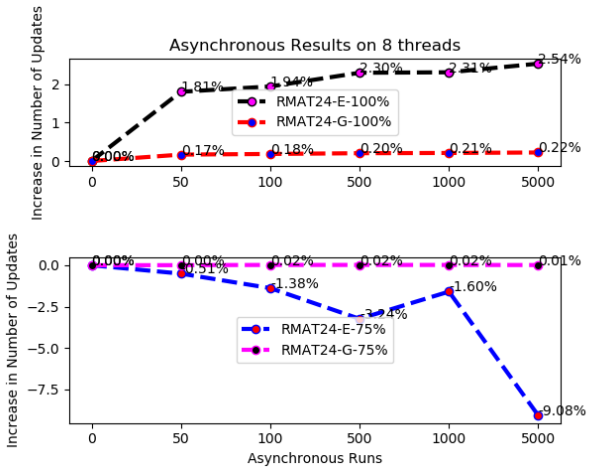


Figure 4.8: Percentage change in number of updates of vertex values with respect to updates for the synchronous case (set at 0) on 8 threads. X-axis: level of asynchrony. Y-axis: percentage change in updates.Asynchrony, in general, leads to more updates of the vertices.

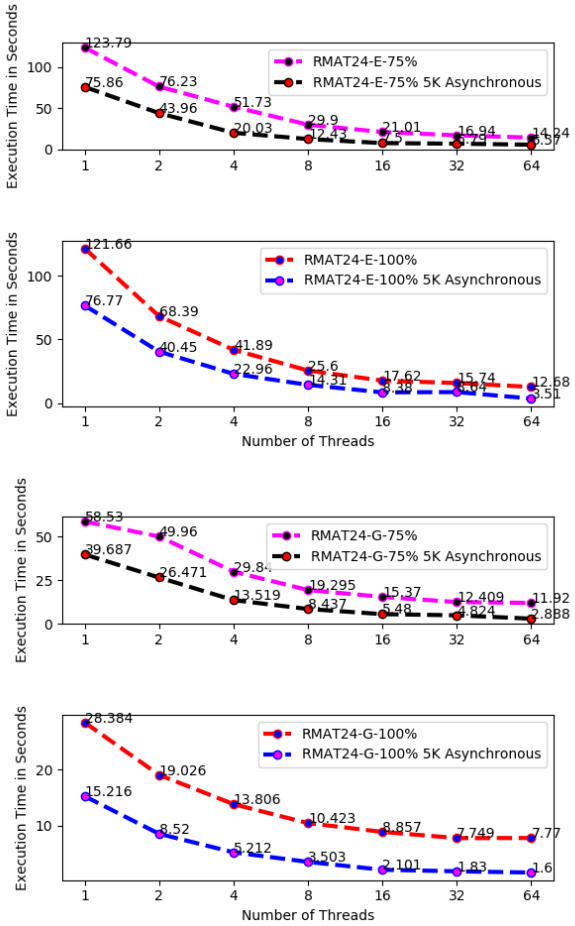


Figure 4.9: Comparison of scalability of synchronous and asynchronous (level 5000) updates. Asynchronous updates are faster and equally scalable.

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 [60] 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 5.2 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.

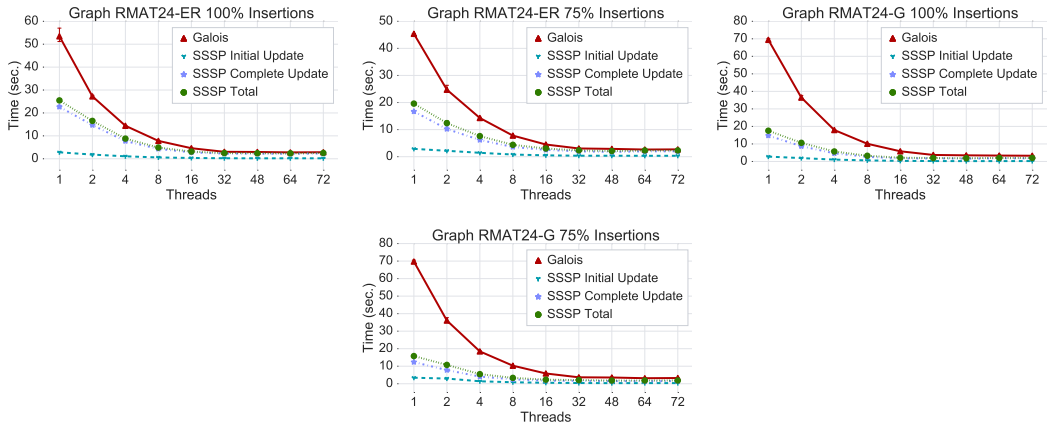


Figure 4.10: Scalability of shared-memory parallel SSSP computation with two RMAT (synthetic) graphs for 50 million edge changes consisting of 100%, 75%, and 50% insertions (left, center, right).

4.5 Correctness of the Algorithm

Consider a graph P, with positive edge weights, that is modified to graph P' due to changes in its structure. now the goal is to prove that the proposed parallel updating algorithm, as given by Algorithms 7 and 18, will produce a valid SSSP tree for P_u.

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

Proof. To prove this, at first assume a graph where the parent vertices point to their children. Now consider a path in this graph between any two vertices p and q. The path goes from vertex p to vertex q. This means that p is an ancestor of q. As per the algorithm for creating the original SSSP tree (Algorithm **??**), and the algorithms for parallel update (Algorithms 7 and 18), a vertex m is

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

assigned as a parent of vertex u only if $Dist[m] < Dist[u]$, therefore transitively the distance of an ancestor vertex will be less than its descendants. Thus, $Dist[p] < Dist[q]$.

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

The tree obtained by our parallel algorithm will be a valid SSSP tree for T_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 p for which the distance of p from the source in T_{alg} is greater than the distance of p from the source vertex in T_u.

Consider a subset of vertices, O, such that all vertices in O have the same distance in T_u and T_{alg}. This means that $\forall vertices \in O$, $Dist_{T_u}[v] = Dist_{T_{alg}}[vertices]$. Clearly, such a set O 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 O. 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, 18, 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 18, 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

GPU Algorithm to Update Single Source Shortest Path

In chapter 4, the author discussed the shared memory implementation of updating SSSP. The experiments did show good scalability, in this chapter author provides an overview of GPU architecture, followed by an optimized GPU SSSP update algorithm, also presents scalability results, and compares the performance with the parallel CPU implementation discussed in chapter 4.

5.1 GPU Architecture

In this dissertation all implementation of GPU based algorithm is on NVIDIA's CUDA (Compute Unified Device Architecture) [61]. CUDA toolkit provides a parallel programming model where you can run single Instructions on Multiple Threads (SIMT). NVIDIA GPU consists of a scalable array SMs (Streaming Multiprocessor) each of which is capable of running a large number of threads. The modern GPU SM supports a block of a maximum of 1024 threads. SIMT architecture provides instruction-level parallelism and hardware multi-threading.

In the CUDA programming model, any code which is meant to run on CPU is called host code, and the GPU executable is called device code or kernel function. The grid architecture of thread consists of blocks, each block in the grid consist of the same number of threads. The graph data for the proposed implementation is stored in the adjacency list, also converting the adjacency list into a row-majored 1D array [62].

5.2 GPU Algorithm to Update SSSP on Dynamic Networks

Efficiently handling the changed set of edges requires a good distribution strategy to achieve proper load balancing, which will play an important role in performance. There are two ways to distribute changed edges one is each thread is assigned to a subgraph, this might lead to improper load balancing. The second approach is each change edges are assigned to a thread and all of them are processed in parallel. The second approach gives a better load-balancing and also ensures there are no idle threads. For the proposed implementation the second approach which is assigning changed edges per thread is used.

5.2.1 Challenges Creating Efficient GPU Algorithm

- Synchronization and Locking: To avoid race condition atomic operations in GPU can be used. However, atomic operations can make the implementation slower or update operations sequentially. To avoid using locks, the other alternative is running the implementation for multiple iterations and checking during each iteration for correctness. Running the update algorithm for multiple iterations can add additional overhead.

- Loop Formation: While deleting edges or inserting edges in parallel there are chances of loop formation that can lead to providing inconsistent results.

5.2.2 Algorithm

- Input: input is a graph G(V, E), List of Changed Edges CE, SSSP Tree which holds the shortest distance for all vertices for the input graph, and the parent vertex for each vertex which is the vertex whose edge weight is used in final to calculate the SSSP.

- Output:- Output is the updated SSSP tree which holds the updated SSSP for all vertices, along with the parent vertex after processing the changed edges.

The proposed implementation has 4 steps, Algorithm 9 is the driver program which calls other algorithms. In Algorithm 9, $Affected_{del}$, and $Affected_{all}$ are initialized, $Affected_{del}$ is the set of vertices which are affected because of deletion, $Affected_{all}$ is the set of the vertices which are affected while processing the changed edges.

In Algorithm 10 all the edges which are identified for deletion are processed in parallel and the affected vertices are marked. If an edge $E(a, b)$ is marked for deletion and if it is part of the original

Algorithm 9: Parallel SSSP Update

Data: Graph $G(V, E)$, SSSP tree T represented by $Dist$ and $Parent$, Changed edges CE, Weight of edge (u, v) denoted as $W(u, v)$

Result: Updated distances of nodes from source s stored as $Dist_u$, Updated $Parent_u$, Updated SSSP tree T_u based on structure of $Parent_u$

1 **Function** `UpdateSSSP(`G, $Dist$, $Parent$, CE, T`):`
2 Initialize two arrays of size $|V|$, named $Affected_{del}$ and $Affected_{all}$ with boolean value $false$
3 ProcessDelEdge($CE, Dist, Parent, Affected_{del}, G, T$)
4 ProcessInsEdge($CE, Dist, Parent, EdgeWt, Affected_{all}, G, T$)
5 Initialize an empty set Arr
6 Filter($Arr, Affected_{del}$)
7 **while** *Arr is not empty* **do**
8 UpdateNeighbors_del($G, Dist, Parent, Affected_{del}, Affected_{all}$)
9 Filter($Arr, Affected_{del}$)
10 **end**
11 Filter($Arr, Affected_{all}$)
12 **while** *Arr is not empty* **do**
13 UpdateNeighbors_all($G, Dist, Parent, Affected_{all}, Change$)
14 Filter($Arr, Affected_{all}$)
15 **end**
16 **return**
17 **Function** `Filter(`$Arr, Affected$`):`
18 Delete all previous elements from Arr
19 In parallel store all vertex $u \in V$ in Arr if $Affected[u] = true$
20 **return**

SSSP tree, then the vertices in the edges are marked as affected and deleted from the Graph. If the above edges are not part of the SSSP tree, then the edges can be deleted from the graph.

Algorithm 10: Function: ProcessDelEdges

```
1  Function ProcessDelEdges(CE, Dist, Parent,
2  Affected_del, G, T):
3      for each edge e(a, b) ∈ CE to be deleted, in parallel do
4          if e(a, b) ∈ T then
5              if Parent[a] = b then
6                  x ← b, y ← a
7              end
8              else
9                  x ← a, y ← b
10             end
11             Remove e(x, y) from T
12             Change Dist[y] to infinity
13             Mark flag Affected_del[y] and Affected_all[y] as true
14         end
15         Remove e(a, b) from G
16     end
17 return
```

Algorithm 11 processes all the changed edges in parallel which are marked for insertion. The changed edge $E(a, b)$ will only affect the SSSP tree if and only if one of the vertices in the distance of the changed edge is reduced from the source vertex. In the above case if the distance of a vertex a or b from a given source is reduced then they are marked as affected. An effective edge insertion will also update the *Parent* of an affected node.

The vertices which are affected due to deletion are filtered out by checking the $Affected_{del}$ flag (Algorithm 9, line 19) inside the Filter function. These vertices are processed in parallel and their child nodes (i.e. the nodes whose parents are marked as affected) are marked affected and their distance is updated as infinity. The filter function is repeated until all the subtrees of the affected nodes are marked.

Finally, in the last step, all the affected nodes are identified and processed in parallel. The Algorithm 13 connects the disconnected vertices using the available edges from the graph and updates the distance of the affected node to a different subtree which offers minimum distance to connect to the source vertex.

5.2.3 Experimental Result

In this section, experimental results are present for the proposed GPU SSSP implementation. The SSSP algorithm was implemented using CUDA C++. All experiments were run on NVIDIA Tesla

Algorithm 11: Function: Process Inserted Edges

```
1  Function ProcessInsEdges(CE, Dist, Parent, Affected
2  , G, T):
3      for each egde e(a, b) ∈ CE in parallel do
4          if Dist[a] > Dist[b] then
5              x ← b, y ← a
6          end
7          else
8              x ← a, y ← b
9          end
10         if Parent[x] = y then
11             skip the edge and add it to original graph edge list
12         end
13         if Dist[y] = ∞ ∧ Affected_del[y] = true then
14             skip the edge and add it to original graph edge list
15         end
16         if Dist[y] > Dist[x] + W(x, y) then
17             Dist[y] ← Dist[x] + W(x, y)
18             Parent[y] ← x
19             Affected_all[y] as true
20         end
21     end
22 return
23
```

Algorithm 12: Function: UpdateNeighbors_del

```
1  Function UpdateNeighbors_del(G, Dist, Parent,
2  Affected_del, Affected_all, Change):
3      for each vertex v ∈ Arr in parallel do
4          Mark Affected_del[v] as false
5          for all vertex c, where c is Child vertex of v in the SSSP tree T do
6              Set Dist[c] as infinity
7              Mark Affected_del[c] and Affected_all[c] as true
8          end
9      end
10 return
```

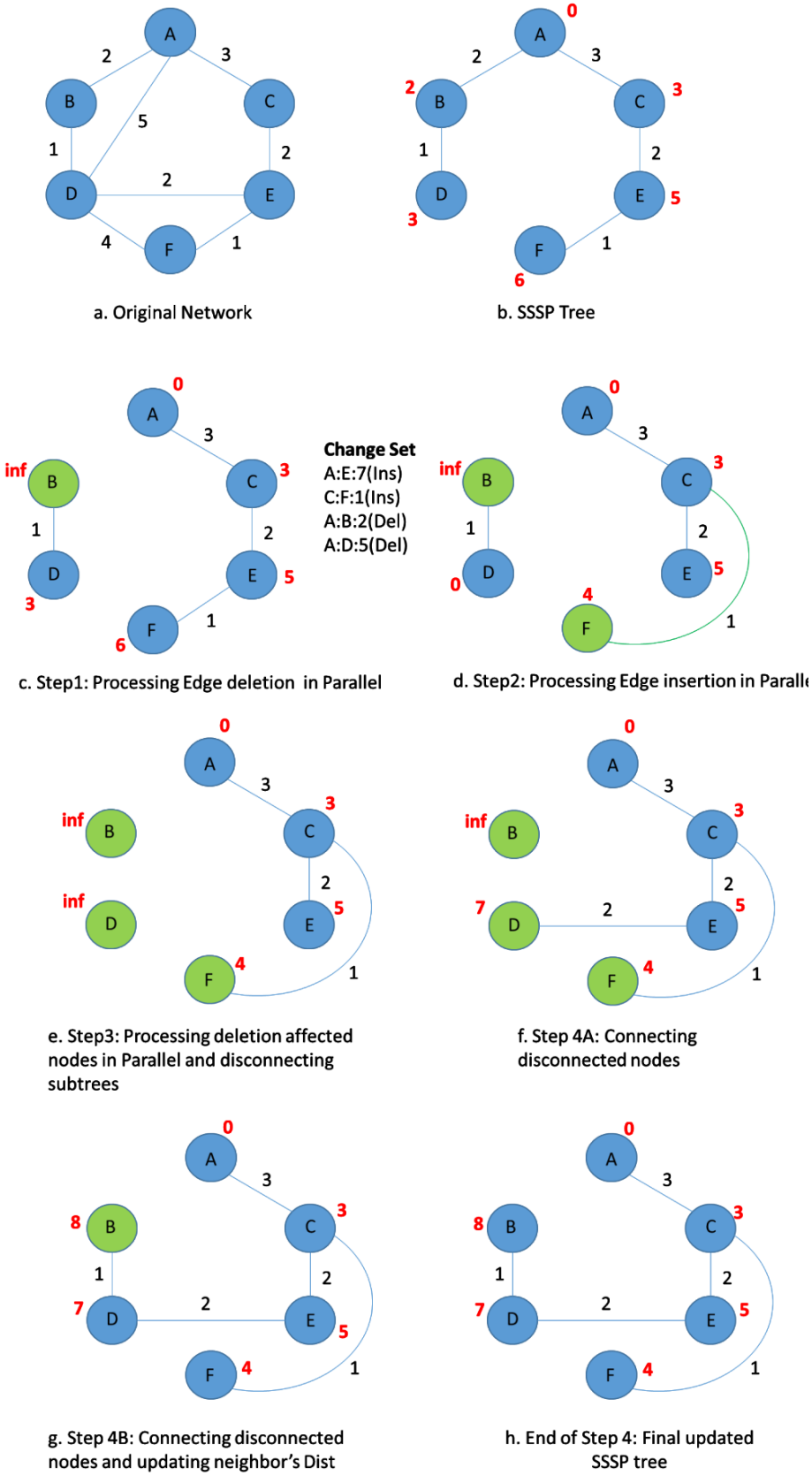


Figure 5.1: Algorithm steps using a toy network to show how the proposed implementation updates SSSP

Algorithm 13: Function: UpdateNeighbors_all

```
1  Function UpdateNeighbors_all(G, Dist, Parent,
2  Affected, Change):
3      for each vertex v ∈ Arr in parallel do
4          Affected_all[v] ← false
5          for vertex n, where n ∈ V and n is neighbor of v do
6              if Dist[n] > Dist[v] + W(v, n) then
7                  Dist[n] ← Dist[v] + W(v, n)
8                  Parent[n] ← v
9                  Mark Affectedall[n] as true
10             end
11             else if Dist[v] > Dist[n] + W(n, v) then
12                 Dist[v] ← Dist[n] + W(n, v)
13                 Parent[v] ← n
14                 Mark Affectedall[v] as true
15             end
16         end
17     end
18 return
19
```

V100 GPU with 32GB memory and Intel (R) Xeon (R) Gold 6248 CPU at 2.50 GHz.

To study the scalability of the GPU algorithm 4 large real-world networks and synthetic networks were chosen. For the synthetic network, the R-MAT graph generator was used and the network was generated with probability (a=0.45, b=c=0.15, and d=0.25). The probability used for generating the R-MAT network is known to have scale-free degree distribution and is labeled as G. The synthetic network used has around 16M nodes and 270M edges. Table 5.1 gives the specs of real-world networks taken from [16] [63].

Table 5.1: Real-world networks used in this experiments

Name	Num. of Vertices	Num. of Edges
Baidu	2,141,301	26,493,629
Flickr	2,302,926	40,989,574
Orkut	2,997,166	106,349,209
Hollywood-2009	1,069,112	2,9142,494

Scalability Analysis

To study the performance of the proposed CUDA implementation, the time for updating SSSP on 1-million, 5-million, 10million, 25-million, and 50-million changed edges are performed. The changed edges consist of different fractions of edges deletions and insertions. The edge insertion percentage means if the changed edges contain x % insertion then there exist (100-x)% edge deletions. All-time

reported are in milliseconds. Figure5.2, and 5.3 are the time taken by our proposed implementation to update SSSP. The performance of 25-Million and 50-Million changed edges are performed for one large real-world network Soc-Orkut and synthetic Network (RMAT-24-G). The time reported in Figure 5.2, and 5.3 show that the proposed implementation does well for 100 %, 95 %, 90 %, 75 % , and 50 % edge insertions. Beyond 50% edge insertion as the number of deletion increases, the proposed implementation takes more time to update. The edges marked for deletion are at least traversed twice to disconnect and connect it back with the SSSP Tree structure. For 1-Million, 5-Million, and 10 -Million changed edges the edge insertion percentage doesn't show a significant difference. In the case of 25-Million, and 50-Million for (100 % edge insertion, 0 % deletion) and (100 % deletion and 0 % insertion) the time to update SSSP shows significant difference.

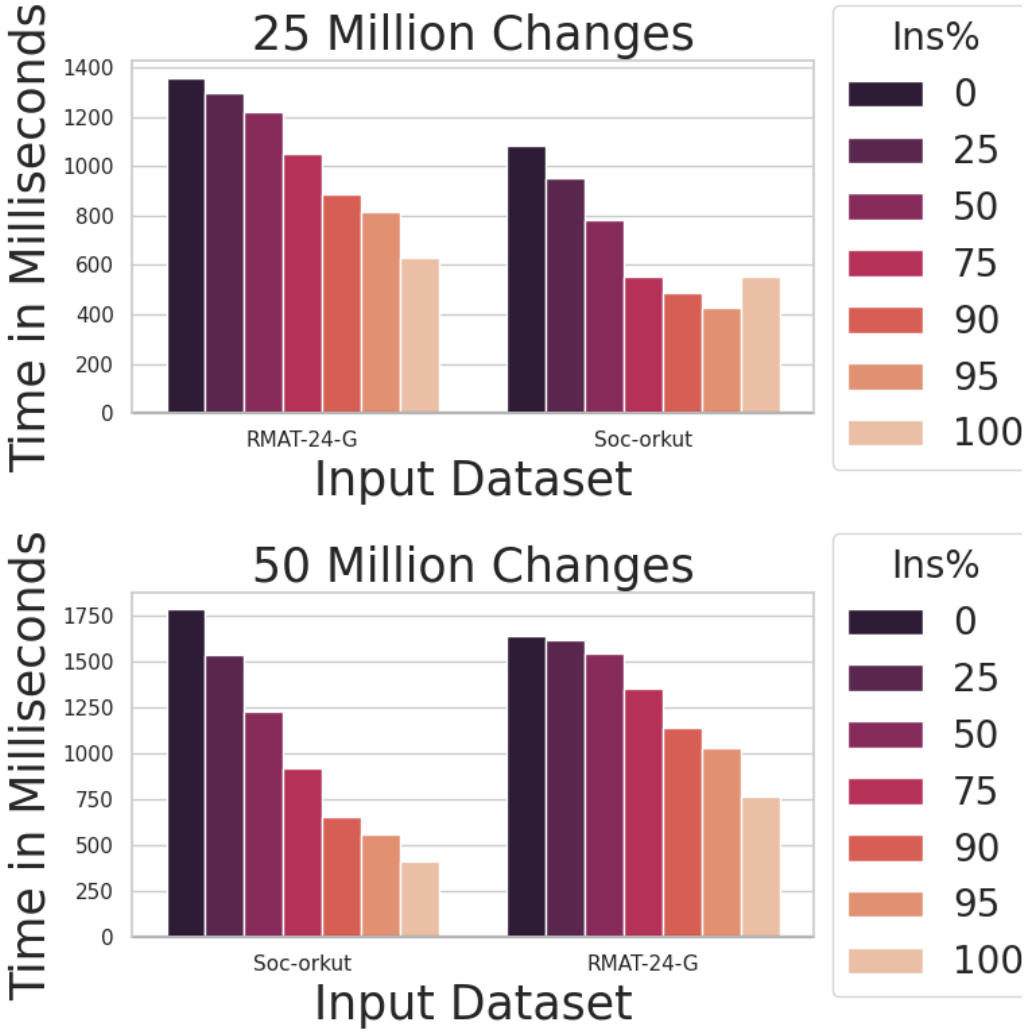


Figure 5.2: Execution time of proposed GPU implementation for networks discussed in Table 5.1 with 25M and 50M changed edges consisting of p = 100%, 75%, 50%, 25%, and 0% insertions and (100-p)%deletions).

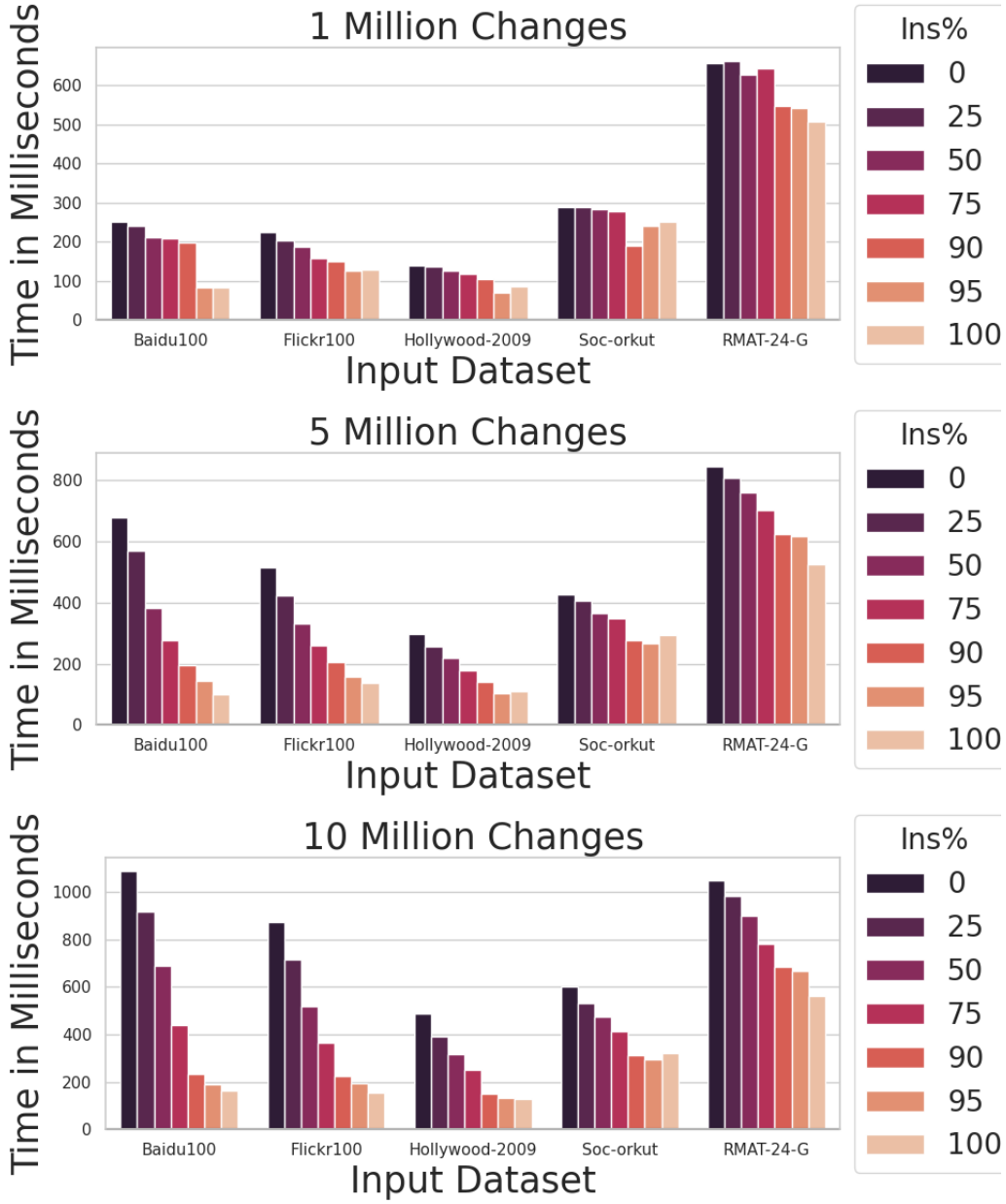


Figure 5.3: Execution time of proposed GPU implementation for networks discussed in Table 5.1 with 1M, 5M, and 10M changed edges consisting of p = 100%, 75%, 50%, 25%, and 0% insertions and (100-p)% deletions).

The speedup of the GPU algorithm relative to the sequential CPU implementation [59] is shown in Figure 5.4, and Figure 5.5. Two networks Soc-Orkut and RMAT24-G with different combinations of edge insertion percentage ran on CPU with no parallelism. The speedup ranges from 1 to 12 for RMAT24-G, with an average of 5.17. The speedup for Soc-Orkut real-world networks ranges from 1 to 14 with an average of 4.17.

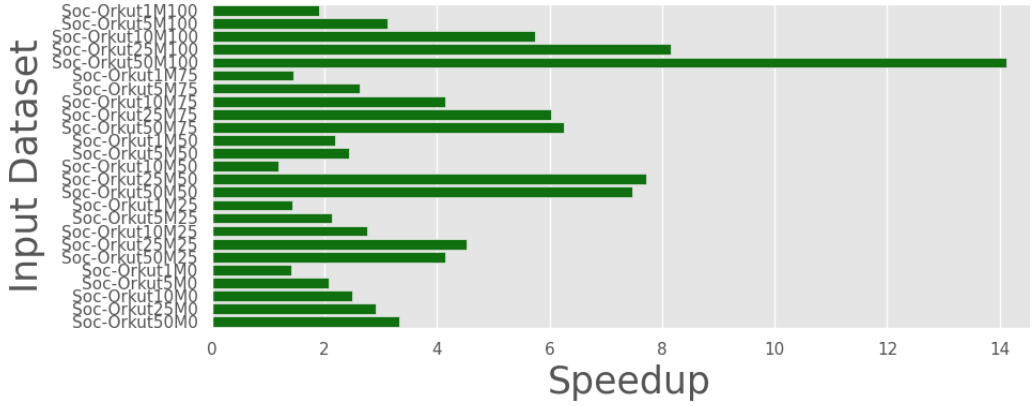


Figure 5.4: Speedup comparison of GPU implementation vs sequential algorithm on the real-world network Soc-Orkut.

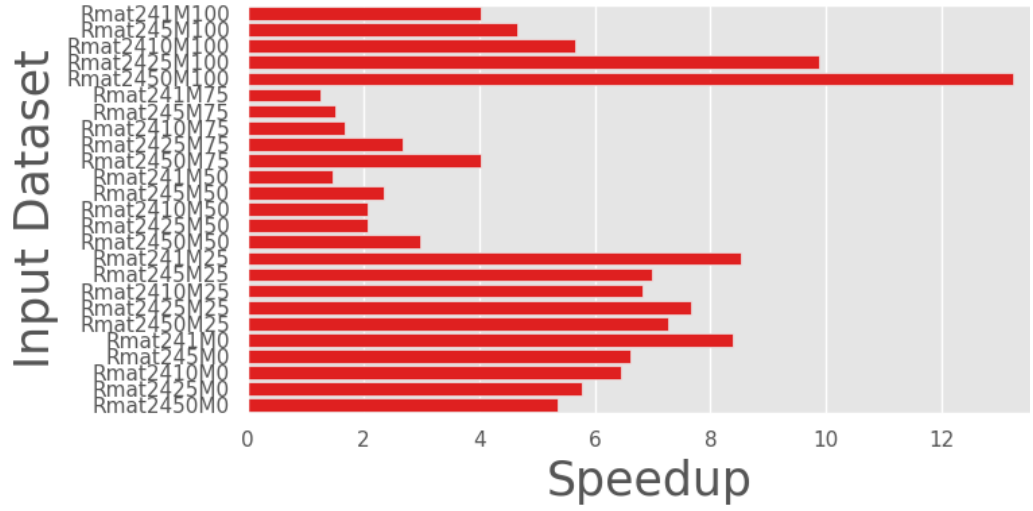


Figure 5.5: Speedup comparison of GPU implementation vs sequential algorithm on the synthetic network RMAT24-G.

The speedup of GPU proposed algorithm relative to the parallel CPU implementation (shared memory) running on 18 threads is shown in Figure 5.2.3, and 5.7. The speedup ranges from 0 to 7 for the Soc-Orkut, on a few occasions, it is observed that CPU implementation does well. For the RMAT24-G network, GPU implementation with 1M changed edges, and 100 % edges marked for deletion the speedup is up to 238.

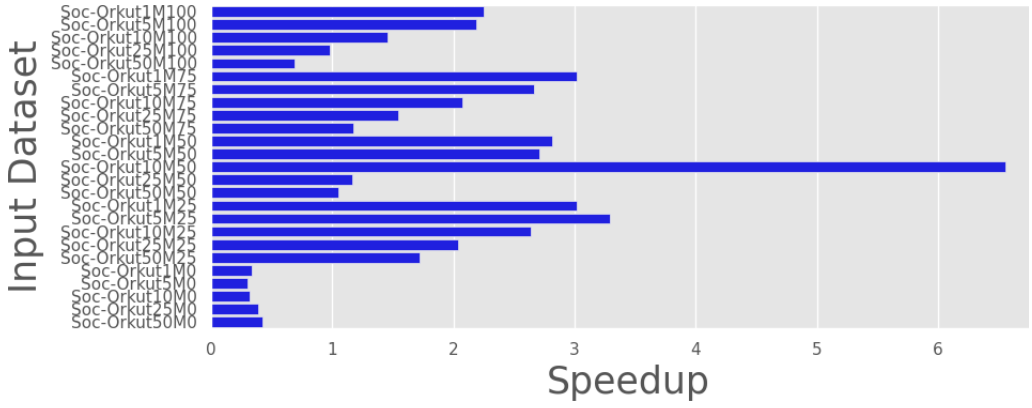


Figure 5.6: Speedup comparison of GPU implementation vs sequential algorithm on the real-world network Soc-Orkut.

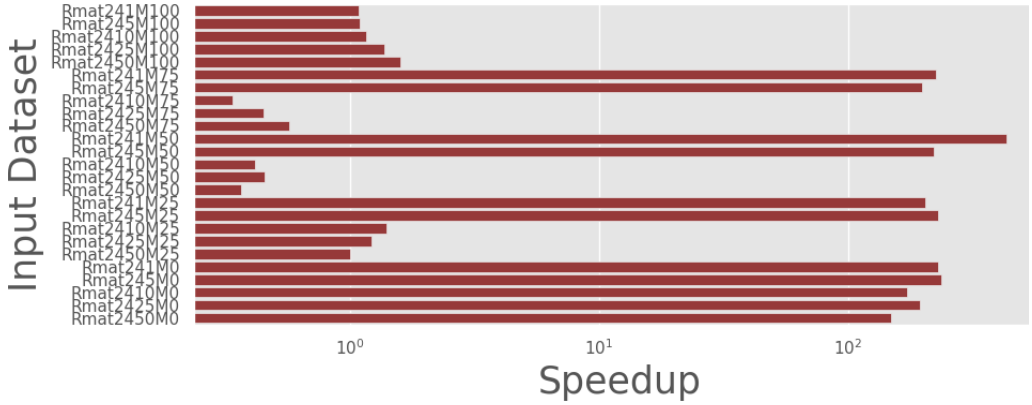


Figure 5.7: Speedup comparison of GPU implementation vs sequential algorithm on the synthetic network RMAT24-G.

Gunrock Comparison

The proposed GPU implementation is compared with other available GPU implementation such as Gunrock [64] and Ingole et al. [65]. The Ingole et al. implementation uses JavaScript WebGL to update SSSP on dynamic networks, however, as per the implementation notes, it is not suitable if the edge updates are greater than 10 %. For the networks and changed edges generated for the Ingole et al. implementation is not suitable. The Gunrock implementation is a widely used tool for graph analytics on GPUs. Among all available GPU graph algorithm, Gunrock implementation shows good speedup. Figure 5.8 shows the speedup of the proposed GPU implementation compared with the Gunrock implementation. The Gunrock implementation only allows static network analytics, so to use the networks and changed edges generated for the experiments, the changed edges were added to the original network. The time to create the input set for Gunrock is not considered. The comparison experiment is performed 6 times and the average time was considered and measured

improvement.

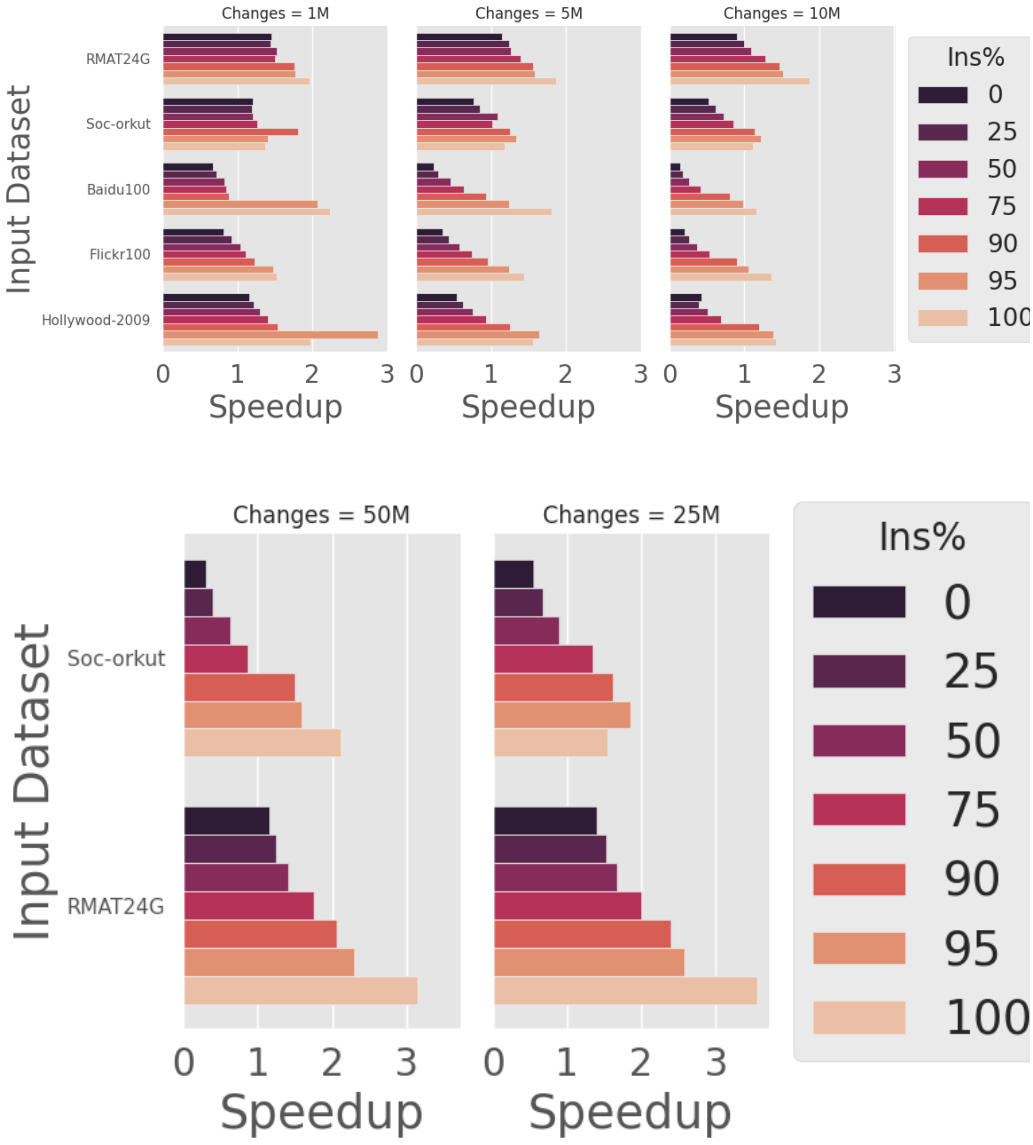


Figure 5.8: Comparison between our GPU implementation of the SSSP update algorithm for dynamic networks, and the Gunrock implementation which computes SSSP from scratch on static networks. X-axis is the speedup, Y-axis is the graph input. The speedup is measured for all networks with 25M and 50M changed edges for 0%, 25%, 50%, 75%, and 100% edge insertion (the rest of the edge changes are deletions

Figure 5.8 shows that in most cases the proposed implementation shows improvement and speedup for 50 % or more edge insertions. In the case where the majority of changes are deletion and changed edges, deletions are above 50 % speedup is good for 1M, 5M, and 10M. For 25M, and 50M changed edges, where the majority of changed edges are marked for deletion speedup of the proposed implementation is low. Based on the observation if the amount of changed edges is greater than 25 % of the network size and the majority of the edges marked for deletion which is more than 50 % it is better to recompute from scratch rather than updating. In the cases where the majority of changed edges are marked for insertion 50 % or above for all combinations including 25M and 50M, the proposed implementation performs well when compared to the Gunrock. For Soc-Orkut speedup is above 1 for 75 %, 90 % , 95 % and 100 %. For both RMAT-24G, and Soc-Orkut anything less than 75% edge insertion, recomputing does well for 25M and 50M. Table 5.2.3 shows the time taken by Gunrock, proposed SSSP update, and improvement of the proposed implementation.

Table 5.2: Improvement of our update algorithm over the Gunrock static algorithm for a 1M, 5M, 10M, 25M, and 50M edge update on the RMAT24 and Soc-orkut graphs. All time are reported in milliseconds.

Experiment	Gunrock	SSSP Update	Improv.
RMAT241M_G_100i	1988.49	496.36	4.006
RMAT241M_G_75i	1932.64	705.707	2.73
RMAT245M_G_100i	1962.65	543.533	4.00
RMAT245M_G_75i	1971.45	953.653	2.06
RMAT2410M_G_100i	1962.65	543.533	4.00
RMAT2410M_G_75i	1971.45	953.653	2.06
RMAT2425M_G_100i	1962.65	543.533	4.00
RMAT2425M_G_75i	1971.45	953.653	2.06
RMAT2450M_G_100i	1962.65	543.533	4.00
SOC-Orkut1M_100i	691.84	235.07	2.94
SOC-Orkut1M_75i	697.43	301.26	2.31
SOC-Orkut5M_100i	694.44	331.48	2.09
SOC-Orkut5M_75i	706.38	372.46	1.90
SOC-Orkut10M_100i	713.38	299.73	2.38
SOC-Orkut10M_75i	703.46	436.91	1.61
SOC-Orkut25M_100i	858.44	435.141	1.97
SOC-Orkut25M_75i	703.46	436.91	1.61
SOC-Orkut50M_100i	874.43	505.258	1.73

Chapter 6

Shared Memory Algorithm to Detect Overlapping Communities

The idea of classifying vertices in a network into groups or sets such that the vertices in the group have stronger relations when compared to vertices in the other group. There are various approaches proposed to detect communities such as the modularity metric proposed by Newman and Girvan [66], for more details about different algorithms and metrics for detecting communities see [67]. One of the well known popular methods for detecting communities is the Louvain method [68]. The key idea of the Louvain approach is to find the best combination which gives a higher value of modularity by moving nodes between communities. This iterative greedy approach keeps continuing until it achieves a stable and optimized modularity value, and there are many practical applications such as getting a better insight into social networks [68]. There exist many parallel implementations of the Louvain approach [69], [70] which makes it fast to analyze large networks with billions of nodes. However, due to the shortcomings of the modularity metric, the Louvain approach fails to detect smaller communities. Also, it has been observed that [71] the modularity score doesn't give any insight about the system, for example considering Jazz and Western USA power grid network, the Jazz has a stronger community structure when compared to the power grid, however, the modularity score is high for power grid network. It has been observed that optimizing modularity forces to generate a community structure for a network, even if the community structure doesn't exist.

To overcome the limitations of modularity, Chakraborty et al. [71] [72] proposed a vertex-based metric called Permanence, which is a score assigned to a vertex and it ranges from -1 to 1. The key idea of this approach is to move vertices across the community and see which community holds the

vertices stronger. In general, if a vertex belongs to a community which means it has a stronger pull from its internal neighbors than the external connections from other communities. Permanence also tends to create Singleton communities when a vertex experiences equal pull from all its neighboring communities. There is a lot of interest in developing parallel versions of the community detection algorithms, which has resulted in the implementation of a wide variety of algorithms suitable for different computational environments.

In this chapter, the author presents a shared memory implementation of a new highly scalable shared-memory CPU algorithm using Permanence a vertex-based metric to detect communities. The data structures used in this implementation allows to avoid using locks for synchronization and is capable of extending the proposed algorithm to detect overlapping communities. To summarize the main contributions of this chapter are:

- The first shared memory algorithm to detect communities using Permanence.

6.1 Permanence Method

Assume a graph G with vertex set V, edge set E, Internal connections I(V), $E_{max}(V)$ maximum connections to a single external community,$c_{in}(V)$ is the internal clustering coefficient which is only specific to the community with internal neighbors, and D(V) is the degree of the vertex. The formula of Permanence for a vertex p is :

$Perm(p) = [\frac{I(p)}{E_{max}(P)} \times \frac{1}{D(P)}]$ - $[1 - C_{in}(P)]$ [71]

The permanence formula has two important components, the first component is coined as pull, which is how much pull a vertex experience from its community and its neighboring communities. The E_{max} only takes the maximum external connections to other communities for a given vertex, and avoids considering all other external connections as the one which has maximum influence than others, If E_{max} is 0 it is updated to 1. The next component is to measure how tightly a vertex is attached to its cluster, if the total count of internal neighbors is less than 2 than $C_{in}(P)$ it is set to 0. Figure 6.1 is a toy network, and Permanence computation of Vertex P in the network is computed using the Formula 6.1. In Figure 6.1, using the Formula 6.1 permanence value of P is calculated.

6.1.1 Permanence Calculation for the network represented in Figure 6.1

$D(P) \leftarrow 6$

I(P)$\leftarrow$2

$\mathrm{E}_{max}(P) \leftarrow 2$

$\mathrm{C}_{in}(P) \leftarrow \frac{2}{3}$

$Perm(P) \leftarrow [\frac{2}{2} \times \frac{1}{6}] - [1 - \frac{2}{3}]$

$\mathrm{Perm(P)} \leftarrow [1 \times \frac{1}{5}] - [1]$

$\mathrm{Perm(P)} \leftarrow [\frac{1}{5}] - [1]$

$\mathrm{Perm(P)} \leftarrow -0.8$

6.2 Permanence for Detecting Overlapping Community

The Formula 6.1 was initially designed for non-overlapping community [71], Chakraborty et al. [72] re-engineered the Formula for overlapping communities, The first term in Formula 6.1 $\frac{I(p)}{E_{max}(P)} \times \frac{1}{D(P)}]$, $D(P)$, and E_{max} remain same. The internal connections can be grouped in two sets :

- Connections (non-shared edges) belongs to only one community, for example P, B in Figure 6.2.
- connections (shared edges) belongs to more than one community, for example P, A in Figure 6.2.

The internal connection is calculated as the sum of non-shared edges plus the sum of shared edges normalized by the number of communities over which they are shared [72]. The second term in the Permanence formula which is the calculation of internal connection and clustering coefficient, In this chapter, the same clustering coefficient, however, it is normalized with the effective internal connection of vertex. The correctness proof of permanence metric is discussed in [72], and [71].

The Generalized Permanence formula for a vertex P in Community C as per the Figure 6.2 is: $Perm^c(P)$=$[\frac{I^C(P)}{E_{max}(P)} \times \frac{1}{D(P)}] - [1 - C_{in}(V)] \times \frac{I^C(P)}{I(P)}$ [72]

Where $I^C P$ is the total set of internal edges in community C and $I(P)$ is the internal connections of Vertex P.

6.2.1 Overlapping Permanence Calculation for the network represented in Figure 6.2

$D(P) \leftarrow 8$

$\mathrm{I(P)} \leftarrow 5$

$$Permanence_g^{c_1}(P) \leftarrow \frac{(1+1+\frac{1}{2})}{2\times 8} - (1-\frac{1}{3}) \times \frac{1+1+\frac{1}{2}}{5} = -0.18$$

$$Permanence_g^{c_2}(P) \leftarrow \frac{(1+1+\frac{1}{2})}{2\times 8} - (1-\frac{2}{3}) \times \frac{1+1+\frac{1}{2}}{5} = -0.01$$

$$Permanence_g(P) = Permanence_g^{c_1}(P) + Permanence_g^{c_2}(P) = -0.18 - 0.01 = -0.19$$

6.3 Community Detection by maximizing permanence

The sequential algorithm (pseudo code) for detecting overlapping communities by maximizing permanence is discussed in Algorithm 16. The iterative approach is designed based on optimizing metrics such as the modularity [73]. The sequential algorithm consists of two phases, In **phase 1, which is the initialization** all vertices are initialized and assigned to a community. In this case, all edges are traversed and each of them is assigned to a separate community, in this phase majority of the vertices are assigned to more than one community.

In **phase 2, which is update** in each iteration permanence of permanence of vertex P $P_g^C(P)$ for all its neighboring communities C is computed. Whenever the computation results greater than Zero vertex v is added to the community C. TempComm is created to store which vertex P belongs. CurrentComm has the assigned community for vertex P, in the initial phase, it will be the community id's obtained by seeding. TempComm and CurrentComm is compared if the TempComm is greater then CurrentComm is assigned TempComm. During each iteration vertex P might see new communities added. There is a possibility of detecting singleton communities, if detected the algorithm retains as it is. The algorithm has a termination condition when it satisfies one of the following conditions:

- Reaches max Iteration condition
- There is no improvement in the overall Permanence score when compared to the previous iteration.

6.3.1 Scalability Analysis

In this section, the performance of the shared memory implementation of Permanence to detect overlapping communities is discussed. The scalability experiments are performed on 2-real-world

Algorithm 14: Sequential Algorithm to Detect Overlapping Communities

Input : Undirected and Connected Graph $G(V, E)$
Output: Overlapping Communities, Permanence Score for all vertices, and Overall Permanence Score of the input Graph G

`/* Assign each Edge e in G(V,E) to separate communities */`
`/* Each edge has two vertices */`
$G - Permanence \leftarrow 0.00$
$Permanence_{prev} \leftarrow -1.0$
`/* Stores the overall permanence score of G obtained from previous iteration */`
$iteration \leftarrow 0$
$SumPerm \leftarrow 0$
while $iter < MaxIteration || sumPerm! = Permanence_{prev}$ **do**
  forall $v \in G(v)$ **do**
    `/* CurrentCommunities is the list of Community id's which` v `belongs */`
    `/* Compute` P^{cur}`(v), permanence score for each vertex */`
    if $P^{cur}(v) \leftarrow 1$ **then**
      $SumPerm \leftarrow SumPerm + P^{cur}(v)$
      score 1 means is the highest value and no need to continue further
    end
    else
      $P^{tmp}(v) \leftarrow 0$
    end
    `/* Find Overlapping Permanence of v */`
    `/* CNeigh sets of Neighboring Communities */`
    $TempComm \leftarrow []$
    forall *community c in CNeigh* **do**
      `/* Temporarily assigning v to c */`
      `/* Calculate` $P^c(v)$`, which is the permanence score of vertex v in C */`
      if $P^c(v) > 0$ **then**
        $P^{tmp}(v) \leftarrow P^{tmp}(v) + P^{cur}(v)\ TempComm \leftarrow c$
      end
    end
  end
  if then
    $P^{tmp}(v) > P^{cur}(v)$**end**
    $P^{cur}(v) \leftarrow P^{tmp}(v)$
    $CurrentCommunities \leftarrow TempComm$
    $SumPerm \leftarrow SumPerm + P^{cur}(v)$
    $iter \leftarrow iter + 1$
  end
  $G - Permanence \leftarrow SumPerm/V$
  $G - Permanence$

Algorithm 15: Parallel Permanence Algorithm to Detect Overlapping Communities

Input : The network $G(V, E)$, and number of threads P
Output: Overall Permanence Score, and Community ID for Each vertices in $G(V, E)$

```
   /* Initialize empty set for each feature                                  */
   /* Choose Seeding, default seeding is Assigning each edge in G(V,E) to
      separate communities, other options are degree based seeding, and
      Clustering Coefficient based seeding                                   */
1  iter ← 0.00MaxIteration← 100
2  while iter < MaxIteration||sumPerm! = Permanence_prev do
3  |  G − Permanence ← 0.00
4  |  Permanence_prev ←sumPerm
5  |  sumPerm ← 0.00
   |  /* All nodes are processed in parallel                                */
   |  /* Updates for all nodes in parallel                                  */
6  |  forall v ∈ G(v) do
7  |  |  Permanence_prev ← Permanence_curr[v]
8  |  end
   |  /* Permanence Computation for all nodes happens in parallel          */
9  |  forall v ∈ G(v) do
10 |  |  Compute_Permanence_Overlap(G(V,E), Permanence_curr[v],Permanence_prev[v] )
   |  |  /* Computation and Formula is discussed in Algorithm 14           */
11 |  end
   |  /* Final_Perm is the Permanence value of all vertices, which also depends on the neighbor's permanence value */
   |  /* All nodes are processed in Parallel                               */
12 |  forall v ∈ G(v) do
13 |  |  Final_Perm[v] ← 0.0
14 |  end
   |  /* All nodes are processed in Parallel                               */
15 |  forall v ∈ G(v) do
16 |  |  forall m ∈v do
   |  |  |  /* All neighbors of V                                          */
17 |  |  |  Final_Perm[v] ←Final_Perm[v]+Permanence_curr[m]
18 |  |  end
19 |  end
   |  /* Process in Parallel, and reduce operation is used for SumPerm to avoid
   |     race condition                                                     */
20 |  forall v ∈ G(v) do
21 |  |  sumPerm+=Final_Perm[v]
22 |  end
23 |  iter ←iter+1
24 end
```

Table 6.1: Real-world networks used in this Chapter

Name	Num. of Vertices	Num. of Edges
arabic-2005	2,141,301	26,493,629
com-DBLP	317,080	1,049,866
wiki-topcats	1,791,489	28,511,807
Twitter	40,334,410	1311,732,668

Table 6.2: LFR networks used in this Chapter

Network	Num. of Vertices	Num. of Edges
LFR–1	450K	44.73M
LFR–2	550K	64.73M
LFR–3	650K	59.91M
LFR–4	1.2M	99.13M
LFR–5	1.7M	149.13M
LFR–6	2.3M	199.45M

networks and 6 synthetic LFR-Benchmark [74] networks. The networks and specs used in this chapter are provided in Table 6.1, and 6.2.

The LFR-Benchmark [74] network comes with ground-truth communities which are used later in this chapter to determine F-score [75] and ONMI [76]. The F-score and ONMI scores provide a score (ranging between 0 and 1) when comparing the communities with their ground truth, in general, higher the score of the metrics shows that communities obtained from the implementation and their closeness to the ground-truth communities. Figure 6.5, 6.6,6.7 shows the scalability curve for all 6 LFR and 2 real-world networks. In most cases, the observation is the proposed implementation scales up to 16 threads for all networks, and beyond that, the curve is flattened. A careful examination or profiling on the implementation showed the permanence calculation takes the majority of the time and the implementation doesn't scale beyond 16 threads. In Chapter 9, the author plans to refactor the permanence computation module in the future to improve the scalability. Figure 6.3, shows the scalability curve for all 6 LFR networks for non-overlapping communities, and Figure 6.4, represents scalability curve for non-overlapping communities for the real-world networks mentioned in Table6.1. The scalability curve doesn't scale well beyond 16 threads, in most cases both for synthetic, and real-world networks the same behavior is observed. Table 6.6 provides the F-Score and ONMI score for LFR networks. For measuring the F-score, and ONMI, the experiments were conducted on 16 threads and it was observed that the high F-score and ONMI correspond to a high-quality solution when compared to the ground truth. Figure 6.8, is the graphical representation of F-score, and ONMI for LFR networks. For score comparison, BIGCLAM [77] another state of the art overlapping community detection software was used (implementation is available to public) and

	Threads	Time in Seconds
com-DBLP		
	1	545
	2	485
	4	425
	8	385
	16	325
	32	300
	64	289
	72	280
wiki-topcats		
	1	1983
	2	1900
	4	1874
	8	1800
	16	1741
	32	1700
	64	1680
	72	1660

Table 6.3: Execution time in seconds for the networks mentioned in Table 6.1

the overlapping communities obtained are compared with the ground truth. Figure 6.9, shows the F-score comparison between the quality of communities generated by the proposed implementation and BIGCLAM [77].

	Threads	Time in Seconds
LFR–5		
	1	1975
	2	1745
	4	1520
	8	1300
	16	1200
	32	1150
	64	1125
	72	1123
LFR–6		
	1	3067
	2	2665
	4	2536
	8	2163
	16	1854
	32	1800
	64	1756
	72	1700

Table 6.4: Execution time in seconds for the networks mentioned in Table 6.2

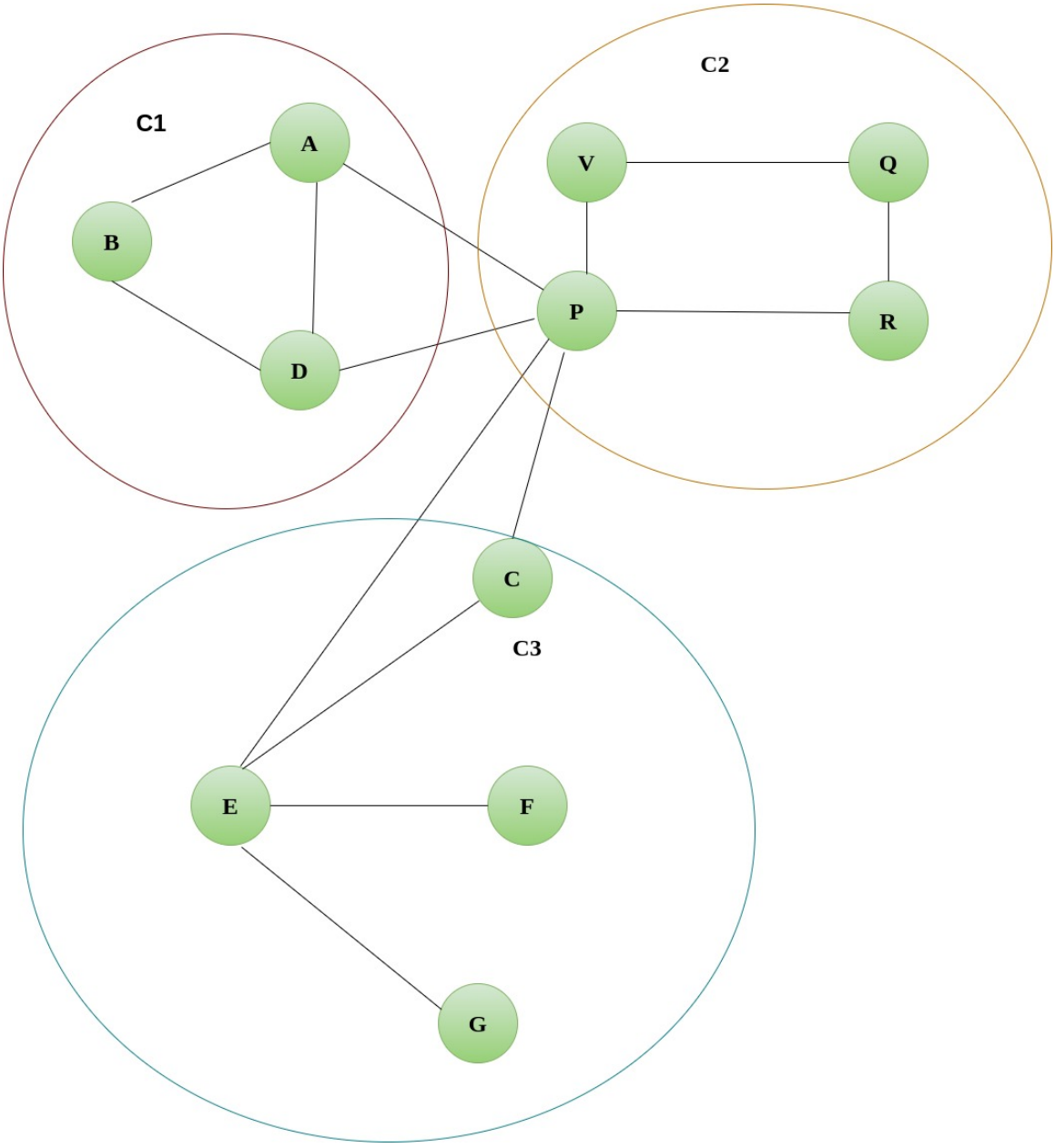


Figure 6.1: Example of Permanence Calculation, for non-overlapping Communities Perm(P)= -0.8. The calculation of Perm (P) is shown in Section 6.1

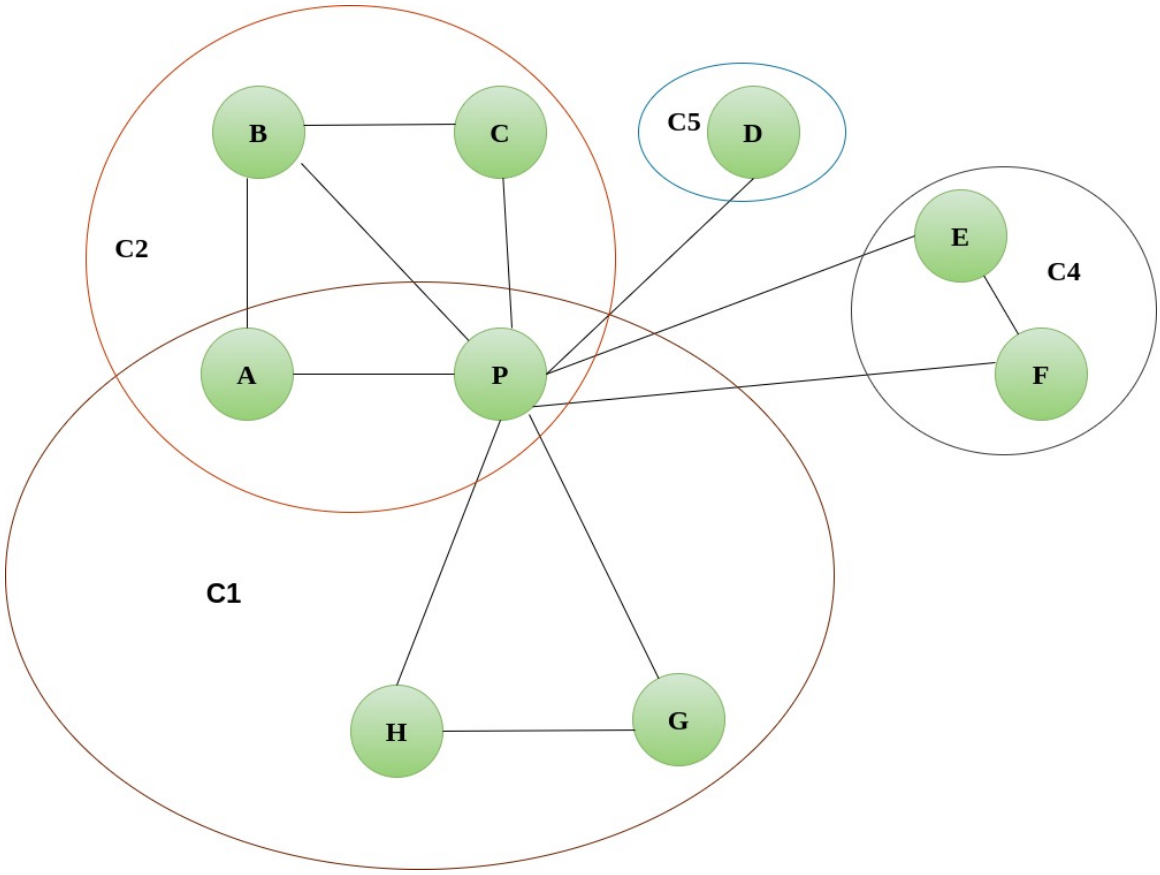


Figure 6.2: Example of Permanence Calculation, for overlapping Communities Perm(P)=-0.19. The calculation of overlapping Permanence (P) is shown in Section 6.2.1

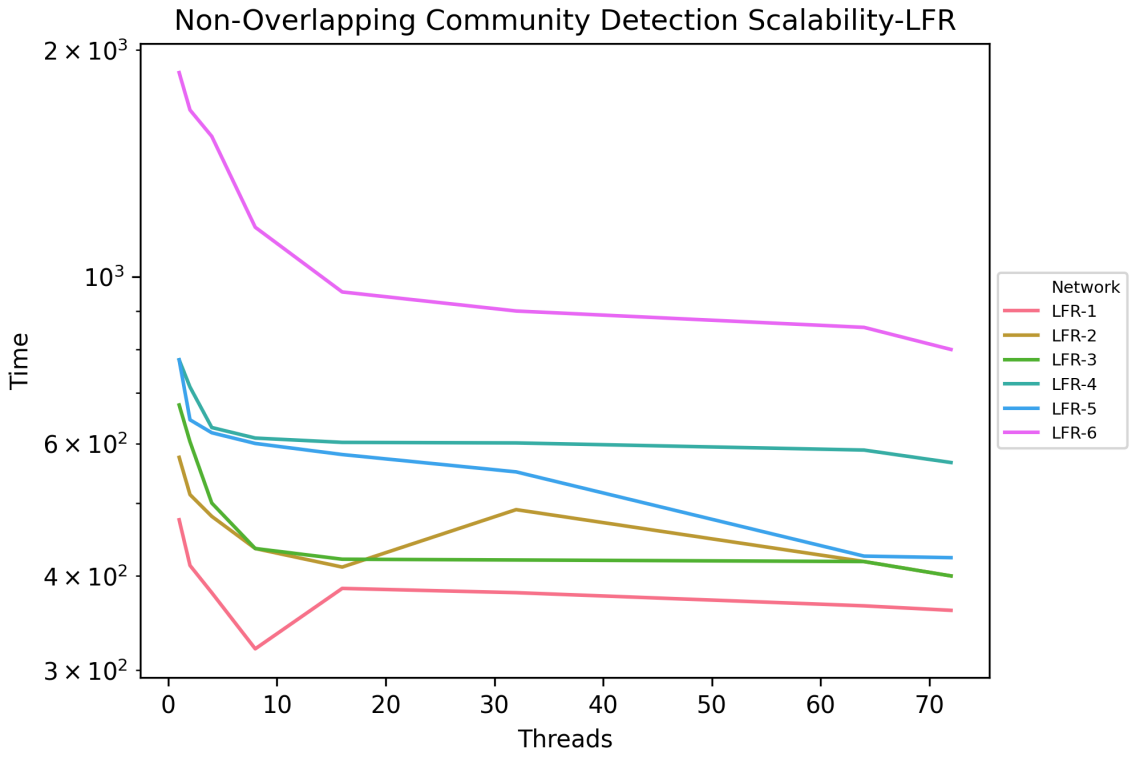


Figure 6.3: Scalability analysis of proposed Community Detection Algorithm using Permanence for Non-Overlapping Communities (LFR Networks)

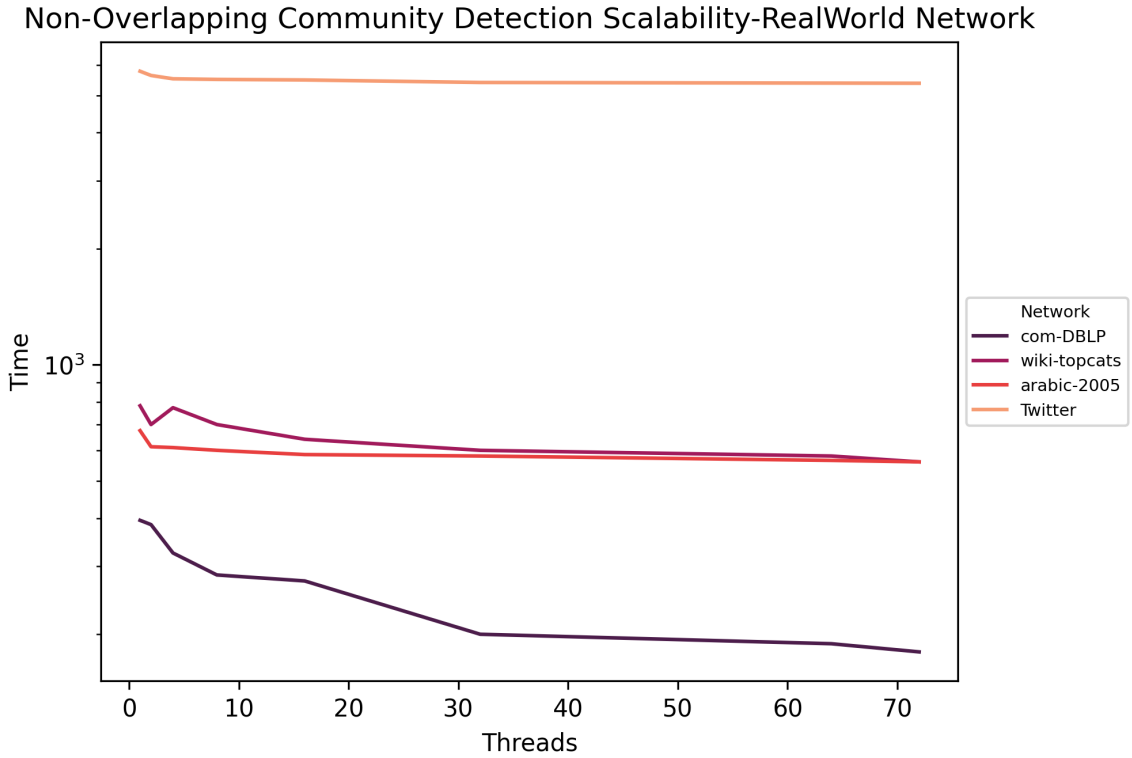


Figure 6.4: Scalability analysis of proposed Community Detection Algorithm using Permanence for Non-Overlapping Communities (Real-world Networks)

	Threads	Time in Seconds
arabic-2005		
	1	2675
	2	2313
	4	2180
	8	1920
	16	1815
	32	1780
	64	1665
	72	1660
Twitter		
	1	30775
	2	30313
	4	28080
	8	26835
	16	24611
	32	24590
	64	23518
	72	23500
LFR-1		
	1	675
	2	613
	4	580
	8	520
	16	485
	32	480
	64	465
	72	460
LFR–2		
	1	775
	2	713
	4	680
	8	635
	16	611
	32	590
	64	518
	72	500
LFR–3		
	1	1075
	2	1003
	4	900
	8	835
	16	721
	32	720
	64	618
	72	600
LFR–4		
	1	1375
	2	1113
	4	1030
	8	910
	16	802
	32	801
	64	788
	72	766

Table 6.5: Execution time in seconds for the networks mentioned in Table 6.1, and 6.2

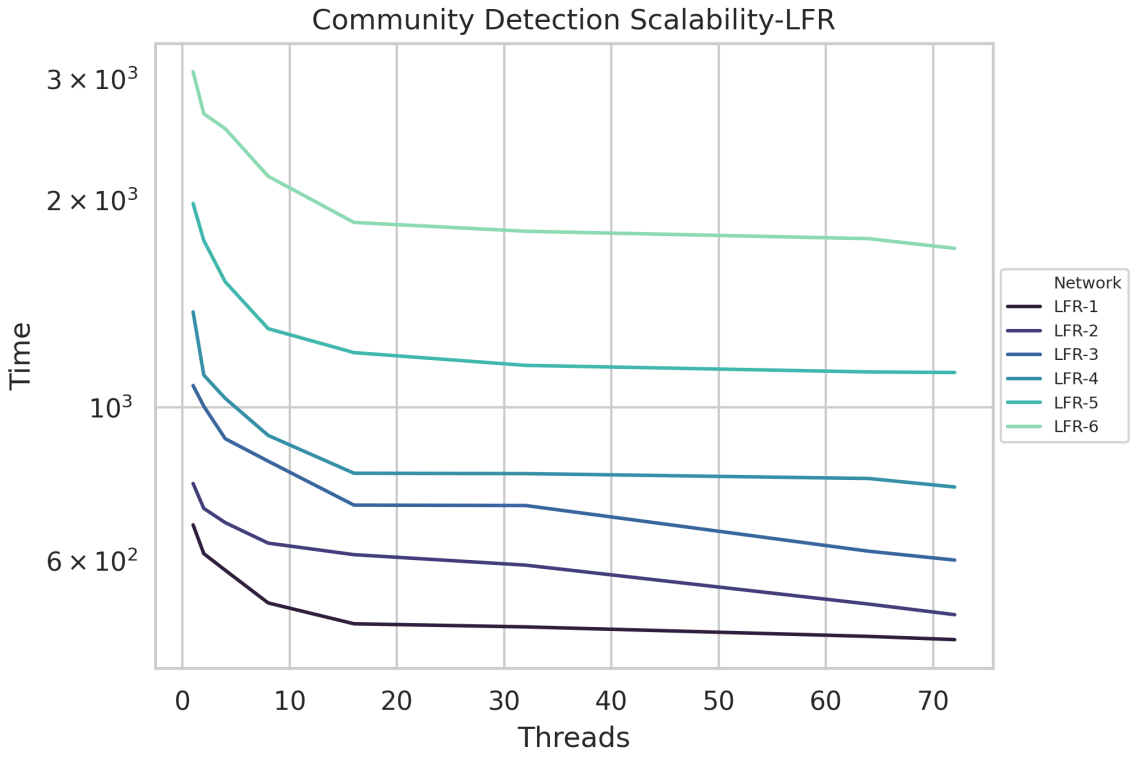


Figure 6.5: Scalability analysis of proposed Community Detection Algorithm using Permanence for Overlapping Communities (LFR networks)

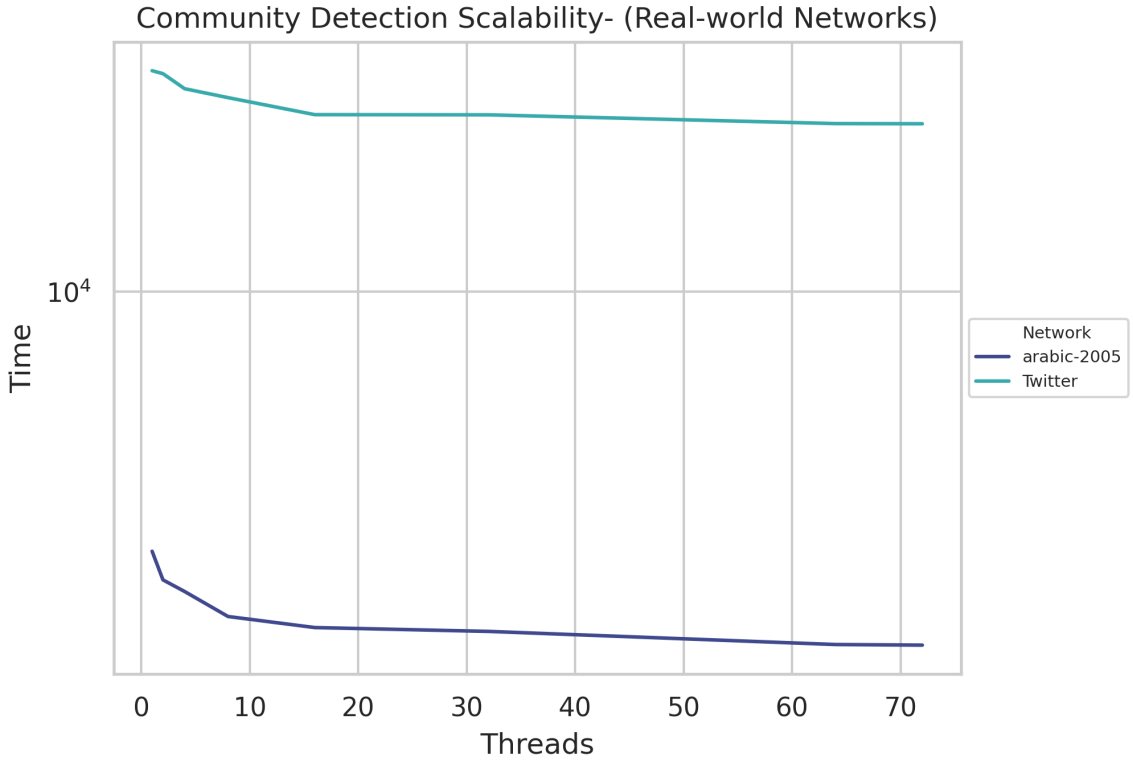


Figure 6.6: Scalability analysis of proposed Community Detection Algorithm using Permanence for Overlapping Communities (Real-world networks)

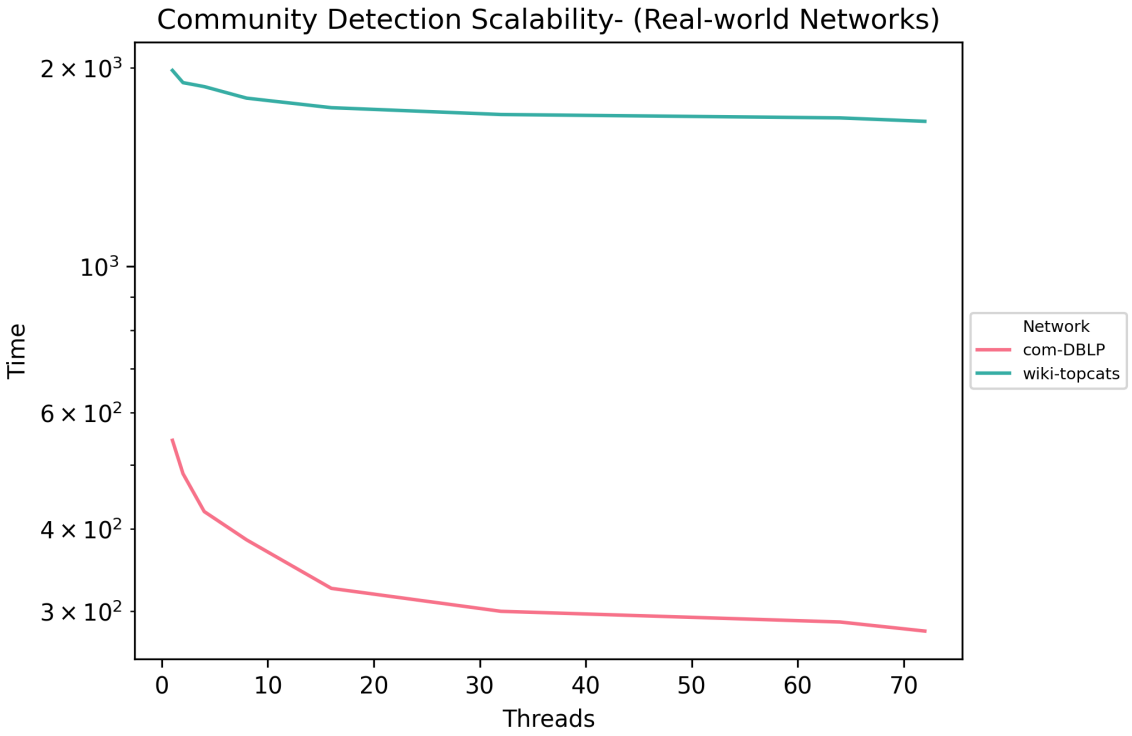


Figure 6.7: Scalability analysis of proposed Community Detection Algorithm using Permanence

Table 6.6: LFR networks F-Score, & ONMI

Network	F-Score	ONMI
LFR–1	0.932	0.926
LFR–2	0.913	0.925
LFR–3	0.910	0.900
LFR–4	0.831	0.861
LFR–5	0.811	0.82
LFR–6	0.790	0.80

Table 6.7: LFR networks F-Score, & ONMI

Network	F-Score(Permanence)	F-Score(BIGCLAM)
LFR–1	0.93	0.91
LFR–2	0.91	0.92
LFR–3	0.91	0.92
LFR–4	0.83	0.87
LFR–5	0.81	0.85
LFR–6	0.79	0.81

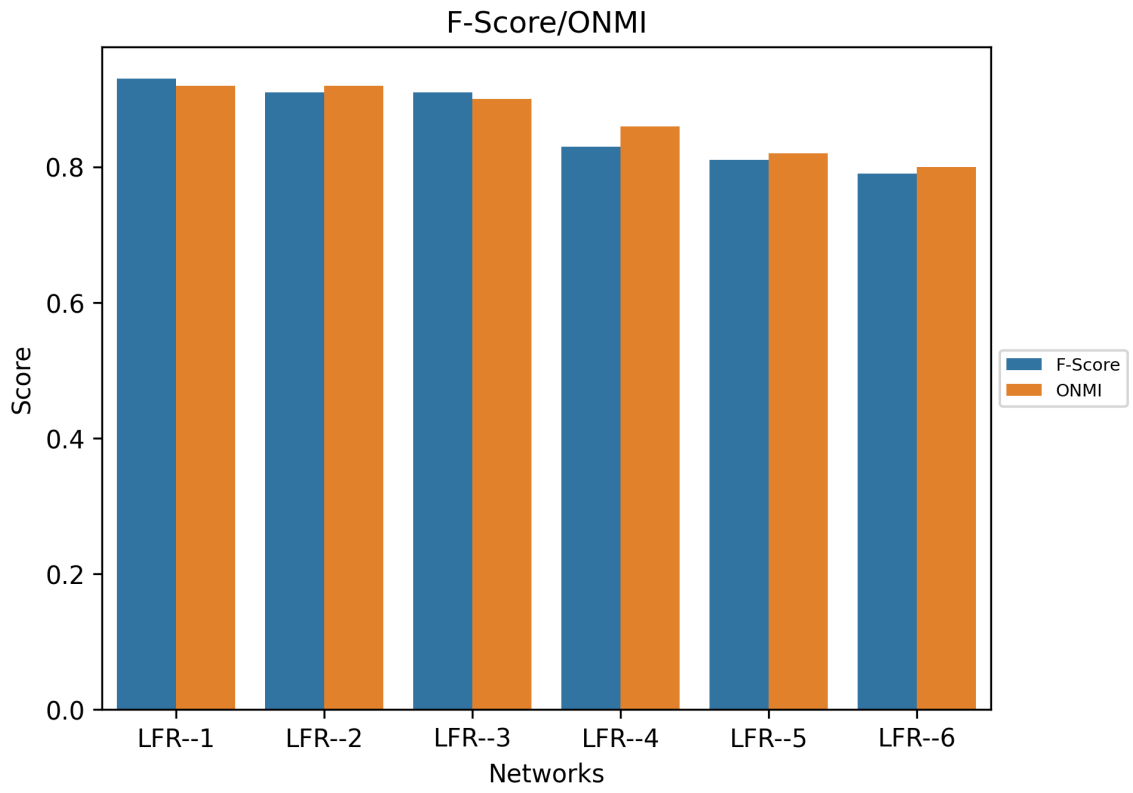


Figure 6.8: F-Score, ONMI comparison of LFR Networks

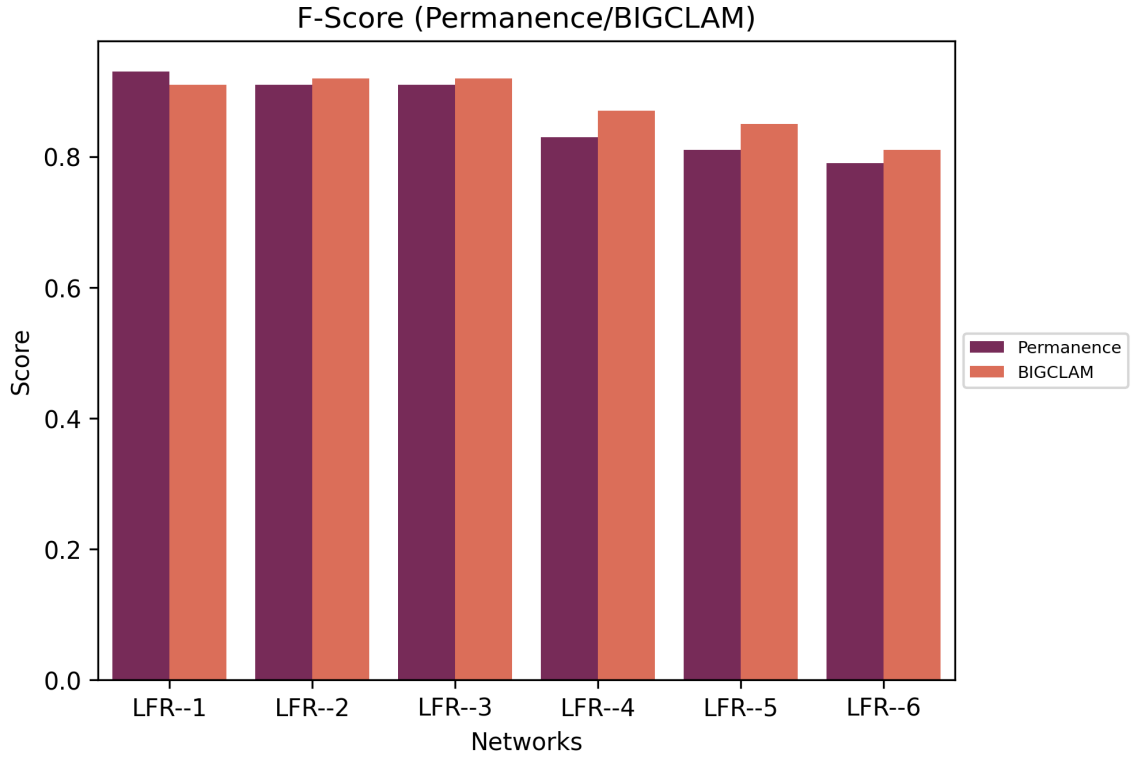


Figure 6.9: F-Score, Permanence/BIGCLAM Comparison

Chapter 7

An Adaptive Approach for Updating Page Rank on Dynamic Networks

The Page Rank (PR) problem involves ranking or assigning a numerical score to each vertex in a graph. It is very commonly used to rank hyperlinked documents such as the World Wide Web (WWW), the goal of this ranking scheme is to identify the importance of documents or vertices in a given set. The computation of a PR is an iterative approach, where each vertex is assigned an initial score, in the next phase the Page Rank values are updated based on the score of their neighbors, this continues until there is no change in the Page Rank of vertices. Page Rank has gained a lot of attention with the success of Google implementation and its accuracy to return search results quicker and accurate. The Page Rank algorithm implementation has many applications such as ranking web pages and designing recommendation systems.

Because of the wide range of applications for finding Page Rank, many researchers have created several sequential and parallel implementation for updating the PR on dynamic networks. There are only a few implementations such as [78] which updates PR based on the changes in the dynamic network. For large networks with a significant amount of changes, updating PR can be computationally challenging. Since the PR value of a vertex depends on its neighbors, if a vertex is affected because of the changing set, all its neighbors PR values should be updated.

In this chapter, the author proposes a novel adaptive approach for updating PR on multi-core

CPU architecture and an algorithm for single GPUs to update PR in large dynamic networks. The proposed adaptive approach based on the changes can choose to update the vertices affected or use the state of the art technology to recompute the PR for all the vertices.

In dynamic graphs, there are constant additions or deletions of edges and vertices. Based on the previous experiments [59] for updating SSSP and MST on dynamic networks, the changes only affect a certain portion of the graph. The proposed implementation tries to identify if the changes for updating PR affects a certain section of the graph or affects the entire graph. If the impact is limited to a certain section of the graph, the adaptive implementation focuses on updating the subgraphs and avoids complete recomputation. To summarize the contributions are:

- A novel adaptive CPU algorithm for updating PR in large dynamic networks, which uses a switch to determine which approach is best updating subgraphs or recomputing from scratch.
- Experimentally demonstrate the performance of the proposed adaptive approach and study the behavior which approach is best suited for different combinations of changed edges.

7.1 Page Rank Background

A graph is mathematically represented as $G(V, E)$, where V is the set of vertices and E is the set of edges. In this chapter, all graphs are unweighted and directed. Page Rank of a given vertex is represented as PR(V), each vertex will have a PR value, which helps to determine their quality. Page et al. [79] introduced PR for internet search engine Google [79]. In general, if a vertex has a high PR value, it means that all its neighbors have a high PR value. The mathematical formula for calculating PR can be written as :

$$PR(v) = d/N + (1 - d) * \sum_{(v,p)\epsilon G} PR(p)/OutDegree(p)$$

[78]

where N is the total number of nodes, the first term d is the probability that any random walk from anywhere can reach the vertex. Ideally, d in the WWW world is that any random web surfer can reach page V can be from a bookmark or a recommendation from any social network. In this chapter, the d value is computed from the initial PR values for the input network. As the network changes if a vertex has an updated PR value its d value is calculated. The Out Degree(p) is the number of outgoing edges. The impact of neighbors and their PR is counted in the second term,

which is reaching the vertex V by traversing thru their neighbors. 1-d is the probability that any random traversal reaches the vertex v by traversing a link.

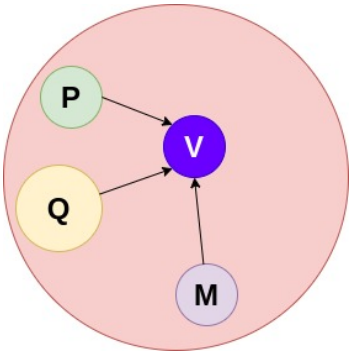


Figure 7.1: Example of Page Rank Calculation

Figure 7.1 shows an example of computing PR of a vertex V based on the value of PR from its neighbors ($(P, Q, and M)$).

$$\begin{aligned}[PR(V) = d/N + (1-d) * [PR(P)/OutDegree(P) \\ +PR(Q)/OutDegree(Q) \\ +PR(M)/OutDegree(M) \\ ]\end{aligned} \tag{7.1}$$

All the networks used in this chapter are strongly connected so that it avoids the computational issue of convergence. To handle Dangling [] vertices, those vertices which have no out Degree, they are marked and don't use it for calculation. When processing changes in multiple batches, If the change list in a batch modifies a Dangling vertex and it has an outdegree, it is updated and will be used for computation. Marking of Dangling vertices happens during the pre-processing step for each batch. In [78] it has been identified that the Dangling vertices don't contribute much to the Page Rank of other vertices.

7.2 Updating the Page Rank

The changes in the dynamic network can be in the form of the addition and deletion of edges/vertices. If we consider a single edge for insertion case (a, b), we mark only the vertex 'b' for an update which can now be reached by 'a', since the PR value of a vertex depends on its neighbors, and then compute PR of 'b'. Vertex a won't have any impact, because the nodes that can reach vertex 'a'

aren't changing for the above scenario. The next step is to keep marking nodes that can be reached by the once marked for update and compute their PR value. This step can be computationally challenging as there are chances that single edge change can have an impact on the entire graph and in the worst-case scenario, the PR values for all nodes will require re-computation. In the case of edge deletion again edge (a, b), now 'b' can't be reached by a, so its PR value will decrease, we mark 'b' for update and repeat the same steps as mentioned for insertion. In both scenarios, only those vertices are marked for an update which sees a gain or loss in the connection that can reach them. Algorithm 16 explains updating PR for a single change.

Algorithm 16: Updating PR for a Single Change

```
1  Input: Unweighted Graph G(V, E), PR, Changed edge E = (a, b), VertexUpdateMarker Q.
2  Output: Updated PR [1] Updating_per_ChangeE, G, Q
3  Find the affected vertex and mark them in G, V
4  if E(a, b) == Insertion||E(a, b) == Deletion then
5  end
6  G(b).updateFlag ← True
7  Process_VertexG
8  for v ∈ G(v) do
9      if G(i).updateFlag == True then
10     end
11     Compute PR (v)
12     G.N is the total number of nodes in Graph G
13     G(v).d is the probability that a vertex can be reached randomly for vertex v
14     PR(v) ←G(v).d/G.N
15     +(1 − G(v).d) ∗ Σ_{(v,p)ϵG} PR(p)/OutDegree(p)
16     for i where i is the node which can be reached by v do
17         if G(i).updateFlag == False then
18             G(i).updateFlag = True
19         end
20     end
21 end
22 Return PR
```

In this section, we present our shared memory algorithm for updating Page Rank. To process changed edges in parallel, we identified there are two ways to distribute: one is to distribute the graph, which is each thread gets a portion of the subgraph. This approach isn't scalable because of load imbalance and there are chances few threads might be idle and other threads might be doing the heavy work. The second approach is to process each set of changed edges in parallel, this approach still has the load imbalance issue as discussed in the previous approach, but it guarantees that all threads will have work and wont be idle. We choose the second approach, apart from load imbalance there are few other challenges:

Race Conditions & Redundant Computation There are chances of overlap when two or more threads are processing the same subgraphs which include the same set of vertices that can lead to redundant computation. To avoid a race condition, we might need to use locks that are expensive and can become a bottleneck to scalability.

7.2.1 Parallel Algorithm Parallel Edge Updates

To address the challenges listed above, the following algorithm is proposed.

Inputs and outputs The input is the directed graph, set of changed edges, and initial page Rank. Since the algorithm is focused on updating existing PR, not re-computing. The output is the updated PR value after processing changed edges.

Step 1. Process Changed Edges and mark the nodes that are affected

In this step, we go over all the changed edges in parallel, and mark the nodes which have a gain or loss in neighbors that can reach them. Line 4 in the Algorithm 16 checks and marks the node as affected.

Step 2. Compute PR of the Affected Vertices

In this step PR values are computed for the affected ones, and the update flag is set to True for the nodes which can be now be reached by the nodes identified in step 1. (see Algorithm 18)

This step process all the affected nodes in parallel, for all the affected nodes the new PR value is computed with the changes, then it is compared with the current PR value, if the difference is significantly greater than the threshold, the old PR value of the node gets overwritten by the new PR value and all the that can be reached by updated PR node is marked for update. There is a while loop that continues to do this until every affected node gets the correct value or if it meets the maximum number of iterations. The implementation allows the user to define their threshold limit and number of iterations for code convergence. The default threshold limit is set to 0.000000001, and the max number of iterations is set to 100.

7.2.2 Addressing Challenges

The proposed implementation addresses the challenges of avoiding using locks for preventing race conditions. Due to iterative updates, the usage of locks is eliminated. The Page Rank of the vertices in the subgraphs that are affected by multiple changed edges gets updated across multiple iterations.

If the Page Rank of the vertex is set to a higher or lower value in one iteration, in a subsequent or later iteration, it gets updated to the correct value. There exist redundant computations when the value of affected vertices is computed multiple times during each iteration.

Algorithm 17: Updating Affected Vertices in Parallel

```
1  Input: Unweighted Graph G(V, E), Affected, PR.
2  Output: Updated PR [1]
3  Process_Affected_Vertex_ParallelG(V, E), PR.
4  Change ← True
5  count ← 0
6  while Change||count < maxIteration do
7      Change ← False
8      for v ∈ G(v) do
           /* Process vertices in parallel                                  */
9          if v.updateFlag == True then
10             PR_new(v) ←G(v).d/G.N
11             +(1 − G(v).d) ∗ Σ_{(v,p)ϵG} PR(p)/OutDegree(p)
12             Threshold is set to 0.000000001, if difference is less than threhold we can remove
               from the affected
13             if PR_new(v) − PR(v) > 0.000000001 then
14                 PR(v) ← PR_new(v)
15                 for n where n is the neighbor which can be reached from v in G do
16                     n.updateFlag ← True
17                     Change ← True
18                 end
19             end
20         end
21     end
22     count + +
23 end
```

7.3 Scalability Analysis

In order to study the performance of the proposed shared memory implementation, the time for updating PR on 1-million, 5-million, and 10-million changed edges are performed. The changed edges are in different fractions of edge deletions and insertions. The real-world networks used are provided in Table 7.2. Figure 7.2 shows the parallel scaling of the proposed algorithm on the real-world networks. The proposed implementation shows poor scalability as while processing the changed edges it is observed that around 95 % or above vertices are marked for update. The proposed implementation rather than updating the subgraph ends up recomputing the entire network. Table 7.2 shows the percentage of vertices whose PR is computed by the dynamic implementation or marked for update for the real-world networks.

Table 7.1: Real-world networks used in Scalability experiments

Name	Num. of Vertices	Num. of Edges
ego-Gplus	18,520,486	13,673,453
cit-Patents(*Temporal Network)	3,774,768	16,518,948
wiki-topcats	1,791,489	28,511,807
soc-LiveJournal1	4,847,571	68,993,773

Table 7.2: Percentage of Vertices affected/Recomputed

Network Name	Changes	% of Vertices
ego-Gplus	10M	97
ego-Gplus	5M	98.3
ego-Gplus	1M	95.5
cit-Patents	10M	97.3
cit-Patents	5M	96.1
cit-Patents	1M	95.5
wiki-topcats	10M	97
wiki-topcats	5M	96.5
wiki-topcats	1M	96.1
soc-LiveJournal1	10M	99.1
soc-LiveJournal1	5M	97.6
soc-LiveJournal1	1M	97.1

7.4 Adaptive Page Rank Approach

As mentioned in section 7.3 the proposed dynamic implementation doesn't show good scalability for real-world networks. As observed that while updating PR, the proposed implementation recomputes PR for the majority of the nodes in the network, a novel adaptive approach is proposed where % of vertices affected is determined while pre-processing the changed edges. Algorithm 18 (Line 5-7) evaluates the % of changes in a given batch and if the % of nodes affected are above 50 % (Line 10-12) updates the network and calls the state of the art implementation Galois [17], which recomputes the PR for the entire network and the updated value is used for next batch.

7.4.1 Scalability Analysis of Adaptive PageRank Approach

To study the performance of the adaptive implementation, the time for updating PR on 100-million, and 150-million changed edges are performed. The changed edges are in different fractions of edge deletions and insertions. The real-world networks used are provided in Table 7.3. Figure 7.5 shows comparison between dynamic and adaptive implementation. Two real-world networks with (100M and 150M) change edges, and (100, and 75) % edge insertion were divided into equal batches and then both implementation (dynamic proposed implementation, and adaptive approach) were used

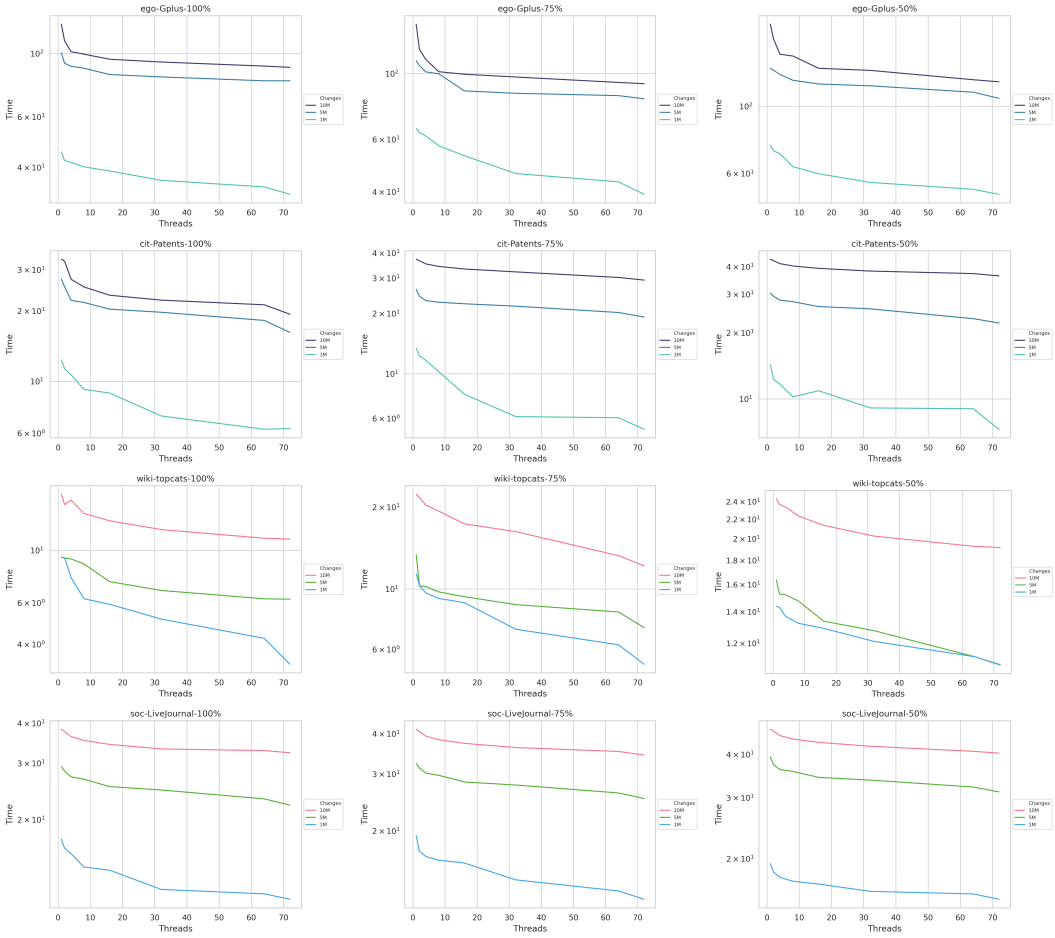


Figure 7.2: Scalability of shared-memory parallel PR computation with 1M, 5M, and 10M changes consisting of 100%, 75%, and 50% insertions.

to update the changed edges. The comparison Figure 7.5 for the Baidu network shows that the adaptive approach always performs better than the dynamic proposed implementation. The average number of times when dynamic implementation was called from the adaptive approach was 3 for Baidu and 2 for Twitter, and for both networks and their changed edges, dynamic implementation was called at least once. For lower thread count adaptive approach takes equal or more time than the dynamic approach. Figure 7.6 shows the scalability of the adaptive approach for two real-world networks (Baidu and Twitter). For Baidu, the adaptive approaches scale well, for the Twitter network around 20 or more threads the scalability curve gets flattened. The Algorithm 18 only calls the recomputing only if the amount of affected vertices is greater than 50 %, decreasing the % shows a great improvement on scalability, when the amount was decreased from 50 % to 30 %, the adaptive approach showed good scalability for Twitter. However, decreasing the affected count mostly calls the recomputing approach.

Algorithm 18: Adaptive Page Rank

```
1  Input: Unweighted Graph G(V, E), Affected, PR.
2  Output: Updated PR in sorted order [1]
3  Filter Affected Changed EdgesG(V, E), PR, Affected, TotalBatches,
   Changed_EdgeList_Insertion, Changed_EdgeList_Deletion .
4  while TotalBatches > 0 do
5      NumberofVerticesAffected ← CountNumberofVerticesAffected(Affected)
6      PR_new ←PR
7      if ((NumberofVerticesAffected/G− > V)×100) < 50) then
           /* Where G− > V, is the total number of nodes, including the changed
              edges from current batch.  If the affected is less than 50, use the
              Dynamic implementation, or else recompute using Galois Implementation
              */
8          Process_Affected_Vertex_Parallel(G(V, E), PR_new)
9          else
10             G′(V, E) ←G(V,E)+Changed_EdgeList_Insertion−Changed_EdgeList_Deletion
11             RemoveDuplicateEdgesIfExist(G′(V, E))
12             Galois_PageRank_Recomputation(G′(V, E), PR_new)
13         end
           /* Update PR to get new values from PR_new                          */
14         for m ∈V do
15             PR[m] ←PR_new[m]
16         end
17     end
18     TotalBatches=TotalBatches-1
19 end
```

Table 7.3: Real-world networks used for studying Adaptive Approach

Name	Num. of Vertices	Num. of Edges
Baidu	2,141,301	26,493,629
Twitter	40,334,410	1311,732,668

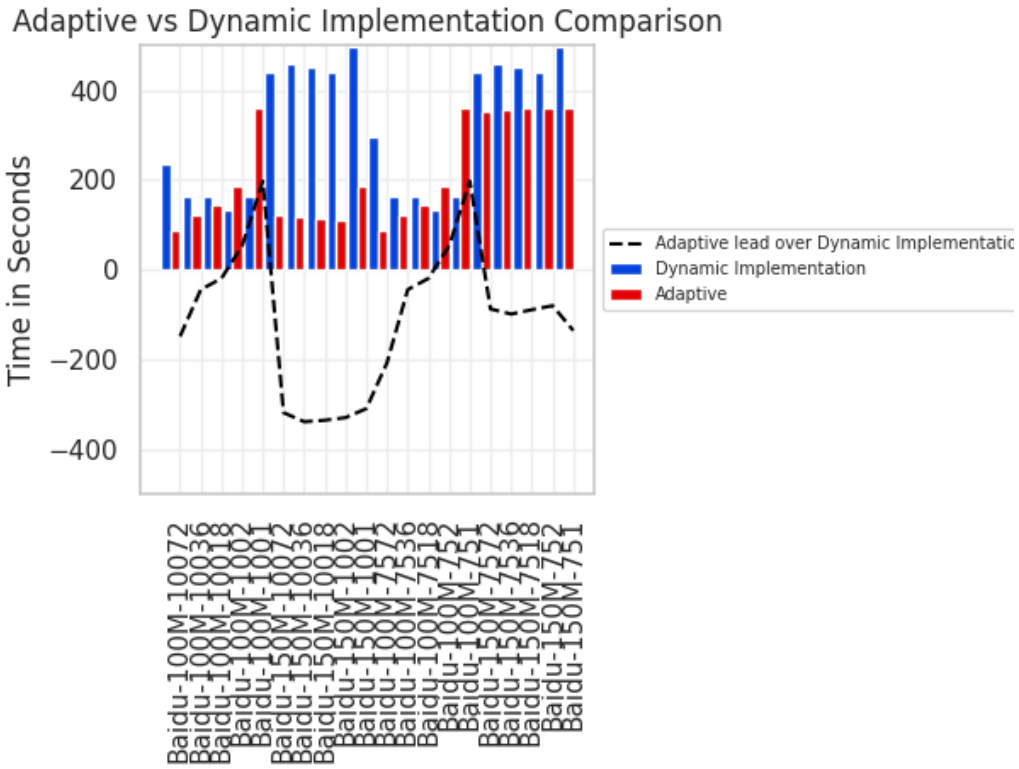


Figure 7.3: Baidu Network

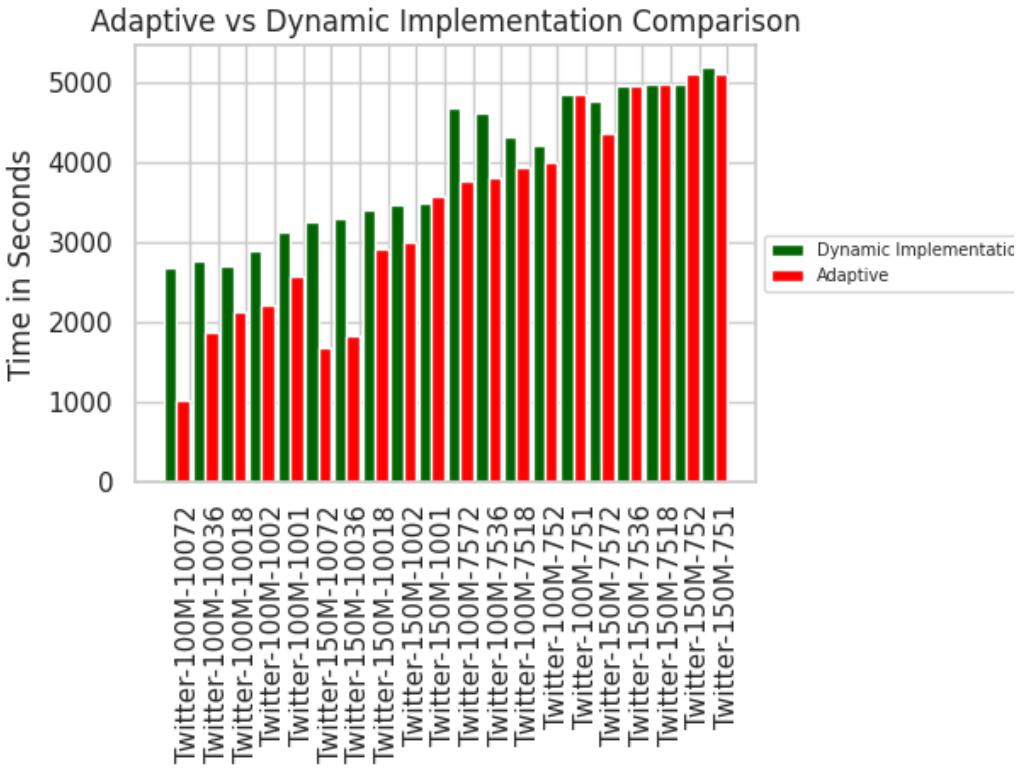


Figure 7.4: Twitter Network

Figure 7.5: Comparison of Adaptive approach vs Dynamic Implementation for networks discussed in Table 7.3 with 100M, and 150M changed edges consisting of p = 100%, 75%, and 50% insertions and (100-p)% deletions).

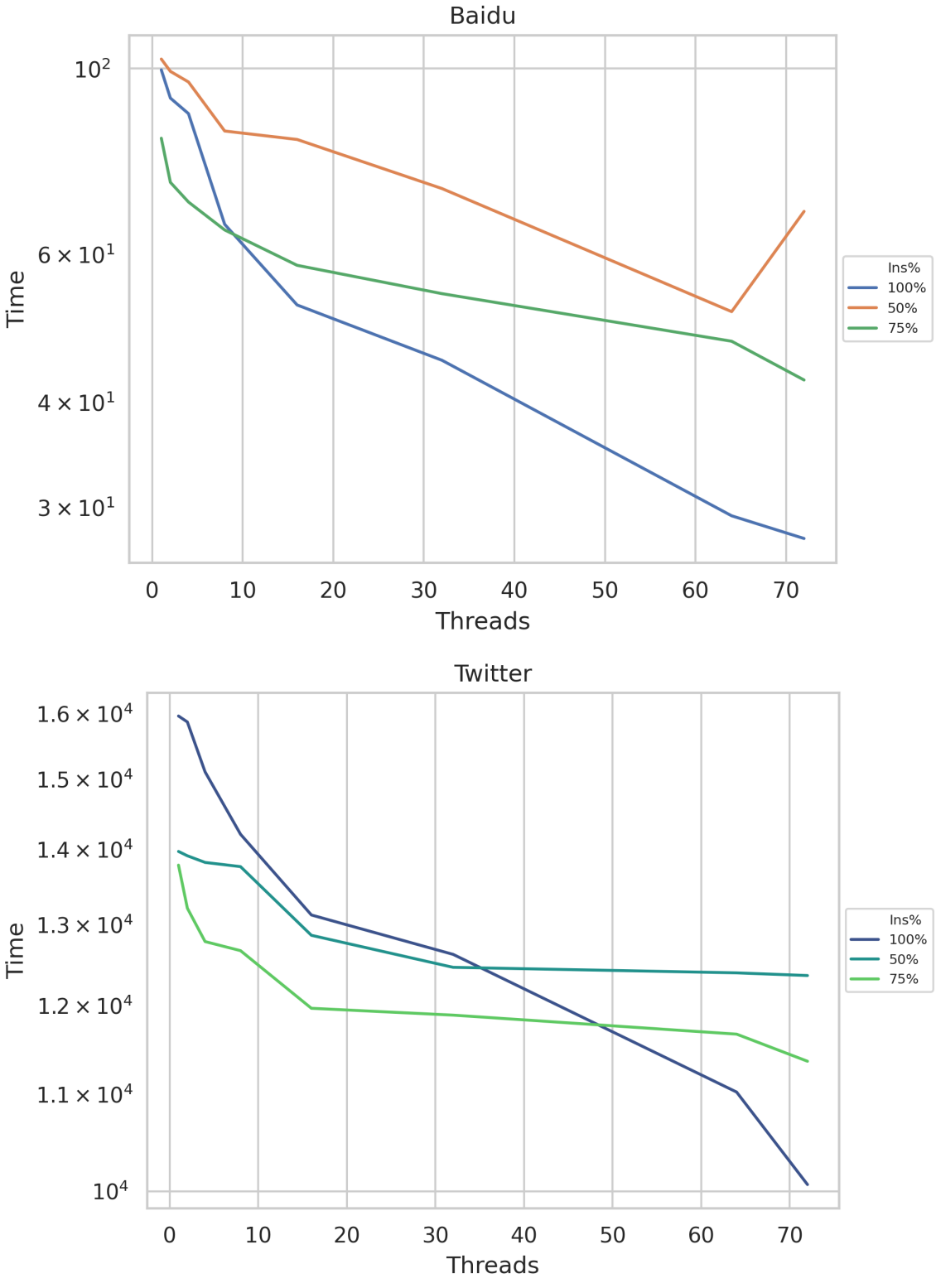


Figure 7.6: Scalability Analysis of Adaptive approach

Chapter 8

Scalable Algorithm to Update Strongly Connected Components

Detection of strongly connected components (SCCs) refers to identifying maximal strongly connected subgraphs in a directed graph. In a directed graph if a subgraph is identified to be an SCC, then there exists a path between every vertex to all other vertices in the subgraph. Detecting SCCs in the real-world graph has many applications.

The popular sequential algorithm to detect SCCs is Tarjan's algorithm, which uses a recursive depth-first search (DFS). There have been many attempts to parallelize the DFS approach but most of them didn't get good scalability. The alternate approach is Forward-Backward (FW-BW [80]) and Orzan's coloring [81]. Slota et al. [82] have observed that shared memory parallelizing of FW-BW and Orzan's coloring doesn't perform well when compared to their sequential counterpart. In Figure 8.1 a directed graph with 30 nodes is randomly generated.

Figure 8.1 has 3 SCC components, the largest one identified is marked and grouped in red color. The nodes 16 and 27 are not part of the largest SCC identified for the toy network. All the nodes which are grouped in red color are part of the largest SCC, have a connection to reach all the nodes except nodes 16 and 27. Ji et al. implementation [83] identified that in real-world, and synthetic networks the single largest SCC holds the majority of the vertices. The other 2 SCC's in Figure 8.1 are the individual nodes 16 and 27 by itself.

To detect SCC's in a network, the FW-BW approach is discussed in Algorithm 19. In this approach, for a given set of nodes in $G(V, E)$, a random pivot element is selected, the next BFS (Breadth-First Search) is performed on the pivot element to see which all vertices can be reached

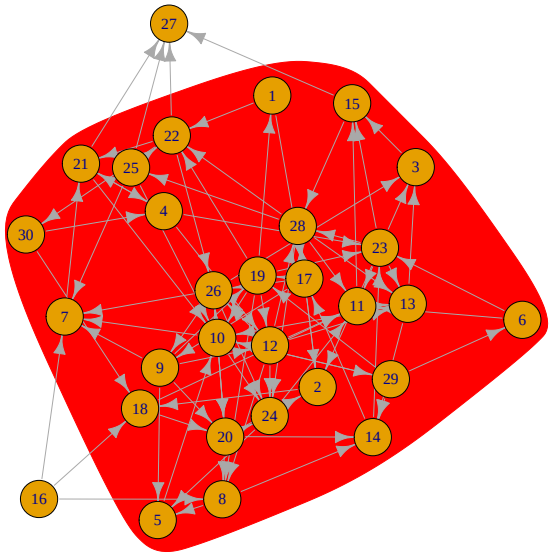


Figure 8.1: Random Network, and the largest SCC marked in red color

from the pivot element. The next step is again performing BFS from the pivot element to see which all vertices can reach the pivot element. The first one can be called a forwarding sweep, and the later one can be referred to as a backward sweep. Now, there are 2 sets of vertices are obtained one which can be reached from the pivot, and the other is which can reach the pivot. Using the two sets, if an intersection is taken, SCC can be obtained. Now subtracting the original graph with edges in SCC, the remaining is again passed to the FE-BW function and this keeps going until there are no set of vertices remaining. In this approach, DFS (Depth First Search) can be used alternatively to BFS. The Algorithm 19 is for the static network. The extension of FW-BW [84] also added a trimming function that is done to remove the nodes whose in-degree or out-degree had a degree of zero.

Algorithm 19: Detecting SCC's using FW-BW

```
1  Input: Nodes.
2  Output: Updated SCC's
3  if Nodes == ∅ then
4  |  return ∅
5  end
   /* Select a Pivot Element from the given list of available Edges          */
6  Pivot ←u
7  Forward ← BFS(G(V,E(v),Pivot)
8  Backward ← BFS(G(V,E(v),Pivot)
9  RemainingSet ← (Nodesn(Forward∪Backward))
10 SCC ← (Forward∩Backward)
11 FW-BW(Forward \SCC)
12 FW-BW(Backward \SCC)
13 FW-BW(RemainingSet)
```

8.1 Tarjan's Algorithm

Tarjan's algorithm is an alternative to FW-BW to detect SCC's. The implementation uses Depth First Search (DFS) to traverse the graph, and the nodes which are part of the SCC form a subtree in the DFS spanning tree of the graph. The order of complexity for Trajan's implementation is $O[V + E]$. Figure 8.2, 8.3, and 8.4 is the graphical representation of Tarjan's algorithm to find SCC's, initially select random node from Figure 8.2, and then perform a DFS, and assign a low link score to each node during the traversal. The low link score stores the lowest reachable from nodeID which is not part of another SCC. Initially, all nodes are assigned low link values to each node ID. Figure 8.3, assuming starting from node 0 and performing DFS, low link score is assigned for all nodes and all the nodes have low link score assigned as nodeID. Figure 8.4,shows the updated low link score for all nodes. Nodes 1,2 can be reached by 0 as their low link score is updated to 0, similarly, nodes 6,7, and 8 can be reached from 5 and their low link is updated to 5. Now there is 4 SCC's, Figure 8.5 shows all 4 SCC's. The implementation of Tarjan's algorithm uses the stack, where instead of low link a modified version is used where for each node the proposed implementation stores dfs_num(), and dfs_low(). The dfs_num(v) stores the value of the counter when the node v is encountered for the first time during DFS traversal. The dfs_low(v) stores the lowest dfs_num reachable from v which is not part of another SCC. The proposed implementation uses a stack whereas the nodes are explored they are pushed to the stack. The nodes which are not explored, are explored and their dfs_low is accordingly updated. The node encountered with dfs_low and dfs_num is explored in its SCC and all the nodes above it in the stack are deleted (popped) out and then an SCC number is assigned.

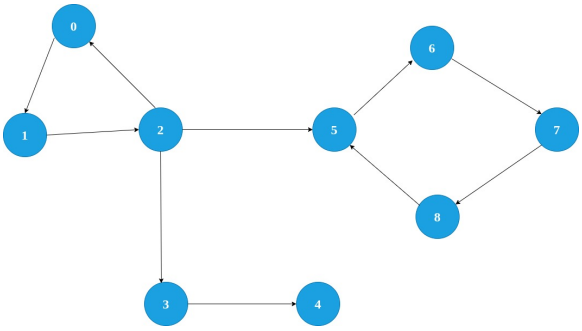


Figure 8.2: Toy-Network

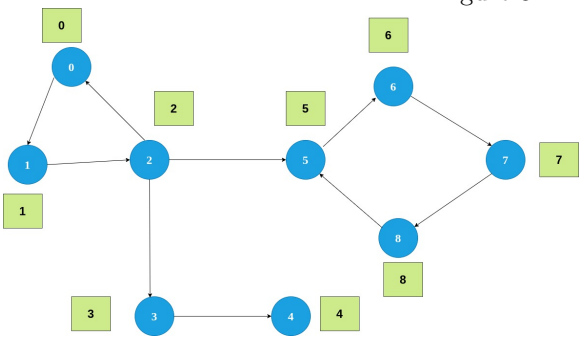


Figure 8.3: Performing DFS, and assigning lowlink

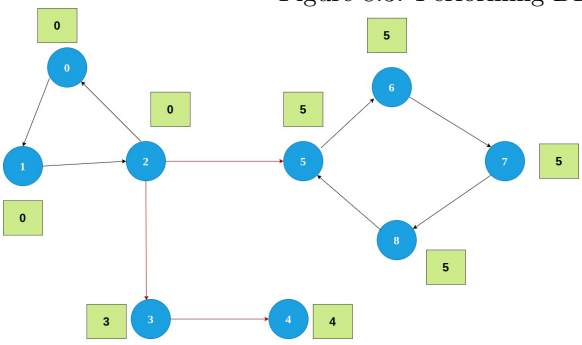


Figure 8.4: Updating Lowlink

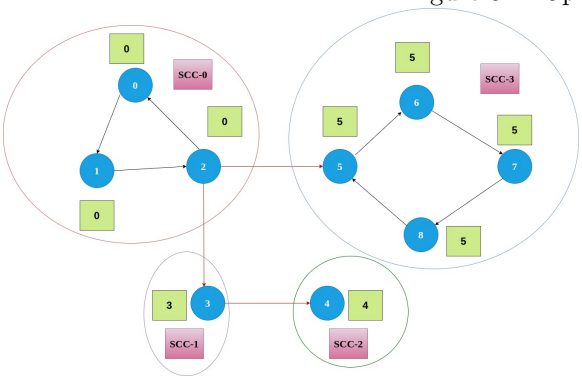


Figure 8.5: SCC's of the network shown in Figure 8.2

8.2 The dynamic graph problem for updating SCC

Given a graph G =(V, E), SCC's, a batch of edge insertions and deletions update the SCC's.

8.3 SCC Algorithm

Algorithm 20 shows the approach to update the SCC of a graph when there is a single edge identified for insertion/deletion. Initially, the changed edge is checked if it is marked for insertion or deletion. Next, SCC IDs are obtained for the changed edge, and in the case of deletion if the IDs are different, and for insertion, if IDs are the same then those cases are ignored. As in both cases, it doesn't affect the existing SCC. If the IDs are the same and are marked for deletion then the next step is to see the connectivity of all the nodes in the SCC ID which was identified in the previous step. If all the nodes can still reach others in the SCC after the deletion of the edge, the SCC remains intact and there is no change required. Alternatively, if there exists a node that can't be reached, the next step is to decompose the SCC and in the repair phase, the nodes which were decomposed needs to be checked for ensuring if there exists any new SCC.

In the case of insertion, where the changed edges connect two different SCC's, there is a possibility that those two SCC's can be merged if the nodes in both SCC's can reach each other. To see if the SCC's can be merged, a subgraph is generated which is the nodes and edges in both the SCC's. On the subgraph, FW-BW backward implementation is called to detect the SCC. Figure 8.3 gives a brief overview of the Algorithm 20. The directed graph is shown in 8.3 has 10 nodes, in Figure 8.3 the SCC of the initial network is represented. The largest SCC is marked in red color and the other two standalone nodes are individual SCC. In Figure 8.3, the network is modified with the changed edges mentioned there are two insertions and deletions. The two insertions/deletions are within the same SCC and different SCC. The Algorithm 20 discusses both cases and how they are handled. When insertions are in the same SCC, we ignore them as they don't contribute to detect new SCC's. If the insertions are in different SCC, such as the edge 6,7- Insertion, the Algorithm 20 checks if the initial SCC which the node 6 belongs can be merged with the SCC of node 7 (which is the individual SCC for the graph shown in Figure 8.3. Similarly for deletion again two cases are considered one within the same SCC and one in between two different SCC's. As discussed in Algorithm 20 deletions between two different SCC's can be ignored, if it is within the same SCC, then the detected SCC needs to be re-evaluated to ensure the SCC remains intact. Figure 8.3, shows the revised SCC after incorporating those changes, and the given changes keep the initial detected

SCC intact.

Algorithm 21, takes the input of the original graph, changed edges, and initial SCC for the original graph. In Algorithm 20, only one changed edge is processed, here multiple changes are initially read, and then the SCC's in which the changed edges belong are marked. To narrow down the effect of changed edges on SCC's, the approach discussed in Algorithm 20 is used only the following changed edges are considered :

- **Deletion**:- If the nodes in the changed edges belong to the same SCC.
- **Insertion**:- If the nodes in the changed edges belong to different SCC's.

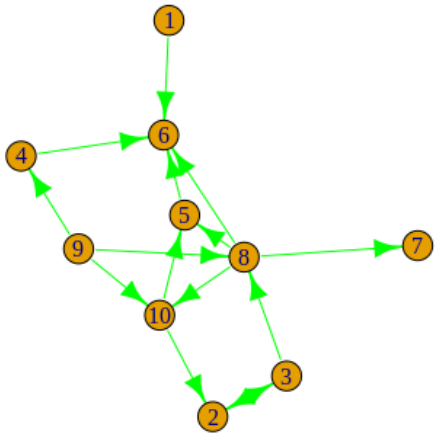
The changed edges are processed in parallel and marked the affected SCC's. The next step is to process deletions, and run Tarjan implementation in parallel from multiple sources and this will give new SCC's if the initial SCC's are decomposed to generate new ones. This implementation works well for networks that have small SCC's, for networks where the majority of the vertices belong to one SCC, this implementation is not scalable.

After processing deletions, the next step is to create a meta graph, where all the SCC's itself becomes a node, and the edges connecting SCC's are only preserved. All the other edges within a given SCC are ignored, as discussed above the insertion of an edge to an SCC, doesn't have any impact. The meta graph is the abstraction of the original graph with changed edges and only representation of SCC and their SCC IDs as nodes, and the edges connecting SCC's. Figure 8.7, and 8.8 provides an example of a given graph and SCC's, and it is meta graph representation. Finally, the meta graph is traversed to verify if there exist any cycles between two nodes (which are the SCC's), if there are any then both the nodes are marked. The nodes identified in the previous step which forms cycle are merged and their SCC IDs are updated.

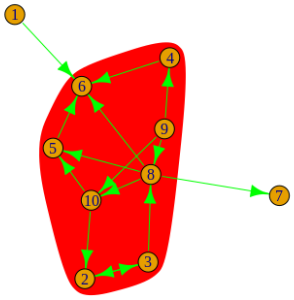
8.3.1 Experimental Result

In this section, experimental results are present for the proposed shared memory SCC implementation. The SCC algorithm was implemented using C++/STL. All experiments were run on Intel (R) Xeon (R) Gold 6248 CPU at 2.50 GHz.

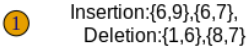
To study the scalability of the proposed implementation 4 large real-world directed graphs were chosen. Table 8.1 gives the specs of real-world networks taken from [16] [63].

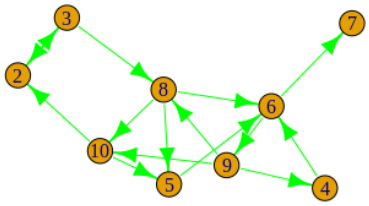


(a) Toy-Network

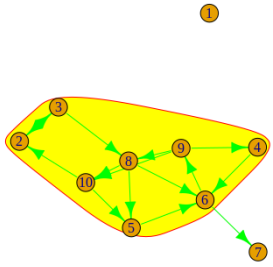


(b) SCC





(c) Modified Network with changes



(d) Updated SCC

Figure 8.6: SCC update algorithm

Algorithm 20: Updating SCC's for a Single Change

Input: Graph $G(V, E)$, SCC, and Changed edge $E = (a, b)$.
Output: Updated SCC
Updating_per_Change$E(a, b)$, G, SCC `/* Check SCC's which are affected */`
$Insertion \leftarrow []$
$Deletion \leftarrow []$
`/* Check Edge for insertion and deletion and store it in appropriate vector */`
if *e.type=="Insertion"* **then**
  $Insertion \leftarrow$ e
else
  $Deletion \leftarrow$ e
end
`/* Check if insertion and deletion is valid */`
if *Deletion.size > 0* **then**
  `/* Check for Valid Deletion */`
  `/* This case no need to do anything as both end points are in different SCC */`
  if $SCC[e.first != SCC[e.second]$ **then**
    $validDeletion \leftarrow False$
  else
    $validDeletion \leftarrow True$
  end
else
  `/* Check Insertion if it is same SCC or different, if both nodes in edge have same SCC, it can be ignored */`
  if $SCC[e.first = SCC[e.second]$ **then**
    $validInsertion \leftarrow False$
  else
    `/* If they are in different SCC, it should be validated that if two SCC's can be merged */`
    $validInsertion \leftarrow True$
  end
end
if $validInsertion == True$ **then**
  `/* Check if two SCC's identified by node can be merged */`
  `/* Call FW-BW algorithm to run SCC on the sub graph */`
  $G'(V, E) \leftarrow EdgesinbothSCC'sFW - BW$(G'(V,E))
else
  `/* Check if SCC is still valid, if not break the SCC */`
  $G'(V, E) \leftarrow EdgesinSCCidentifiedfordeletion$
  $valid \leftarrow$ ValidSCC(G'(V,E))
  if *valid=False* **then**
    Decompose$G'(V, E)$
  end
end

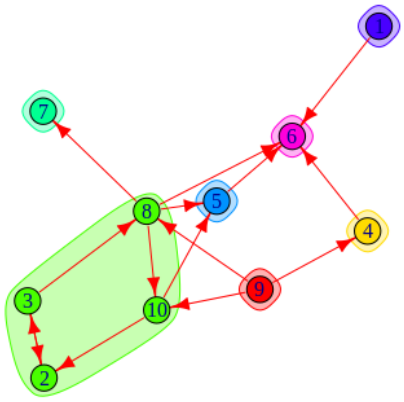


Figure 8.7: Toy-Network, and its SCC

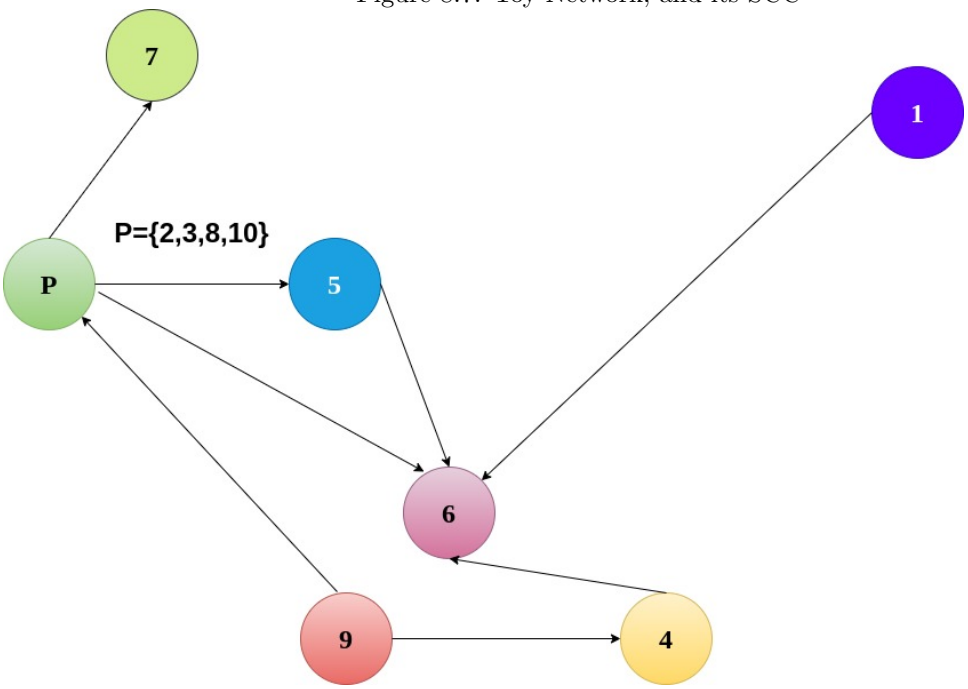


Figure 8.8: MetaGraph

Algorithm 21: Updating SCC's

```
1  Input: Graph G(V, E), SCC, and Changed edges E.
2  Output: Updated SCC
3  Deleted_Nodes Check_Valid_Deletion(G(V, E),E,SCC) /* The logic for this method is
       already discussed in Algorithm 20 Line Number :-11-16                    */
4  Multiple_Tarjan_del (G(V, E),SCC,Delete_Nodes, Stack) /* Obtain new SCCs by running
       Tarjan from multiple sources, this will be happening in parallel         */
5  G'(V, E) ← Create_Metagraph(G(V,E),SCC)
   /* The metagraph creation converts all SCC's as a node and uses the metagraph
       for verifying the new changed edges for SCC insertion                    */
6  Trim (G''(V, E) ←
     G'(V,E))/* Trim function is used to remove nodes whose in-degree, out-degree and self loops a
7  inserted_Nodes ← Check_Valid_Insertion(G"(V,E),E,SCC)
   /* The logic for this method is already discussed in Algorithm 20 Line Number
       :-18-23                                                                  */
8  Multiple_Tarjan_Insertion (G''(V, E),SCC,Delete_Nodes, Stack) /* The Tarjan algorithm
       for insertion is run in parallel from different sources                  */
9  return updatedSCC's
```

Table 8.1: Real-world networks used in Scalability experiments

Name	Num. of Vertices	Num. of Edges	Largest SCC Size
web-BerkStan	685,230	7,600,595	334857 [16]
web-Google	875,713	5,105,039,	434818 [16]
web-NotreDame	325,729	1,497,134	53968 [16]
web-Stanford	281,903	2,312,497	150532 [16]

Scalability Analysis

To study the scalability of the proposed implementation, the time for updating SCC on 1-million changed edges is performed. The changed edges consist of different fractions of edges deletions and insertions. The edge insertion percentage means if the changed edges contain x % insertion then there exist (100-x)% edge deletions. For this chapter, edge percentage used are 100 %, 90%, 85%, 75%,and 50 %. Figure 8.9, and 8.10, shows the scalability of the proposed implementation for the networks discussed in Table 8.1. For 100%, and 90 % the proposed implementation scales well, however, for the 85 %, 75%, and 50 % edge insertion, the proposed implementation doesn't scale. The reason for poor scaling is because due to the large SCC in all the networks which hold the majority of the vertices for the above networks. Table 8.1, gives the largest SCC size for all the networks used in this chapter. The table below provides the time to update SCC in seconds for all networks with different combinations of threads and edge insertion percentages.

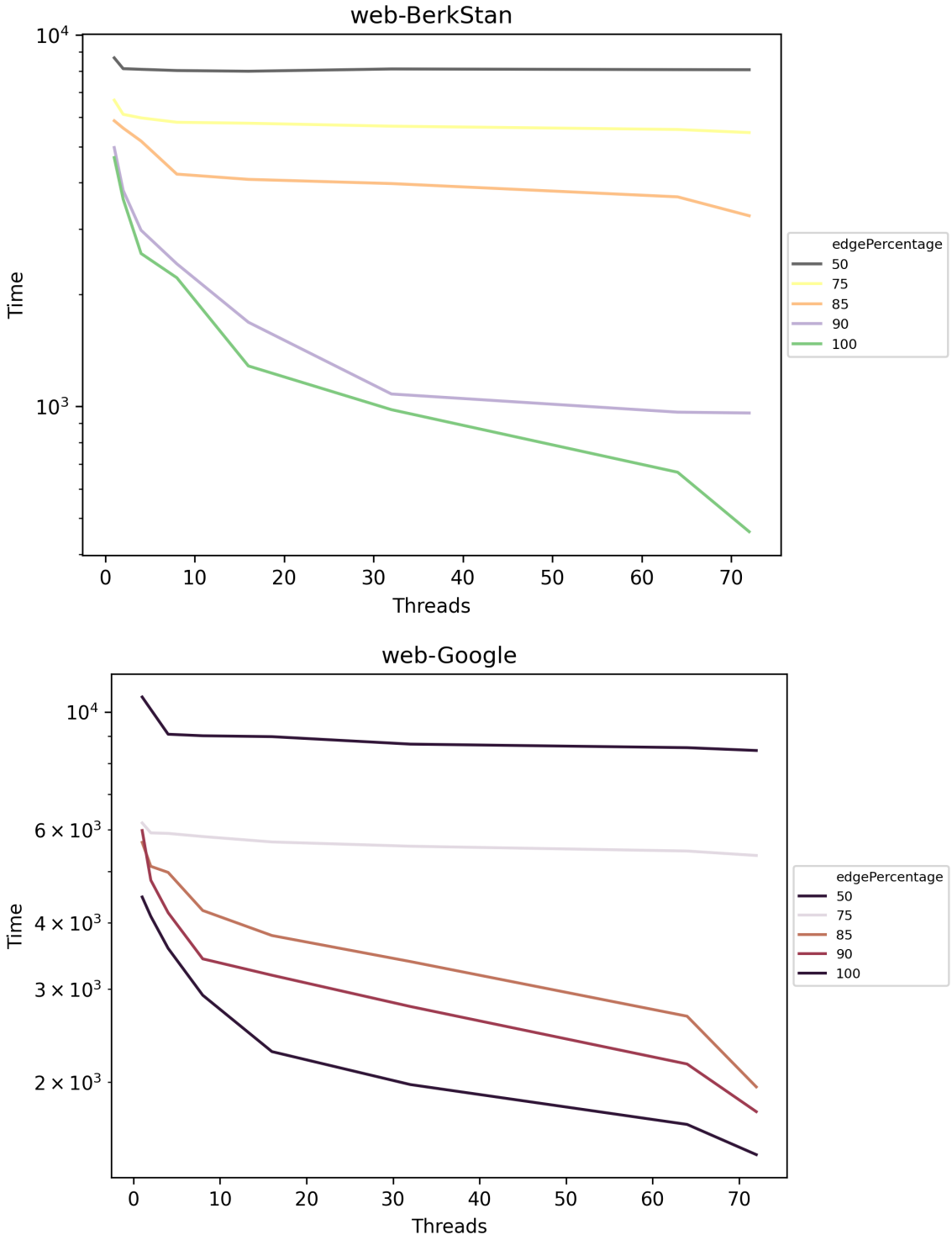


Figure 8.9: Scalability analysis of proposed Shared Memory SCC update Algorithm

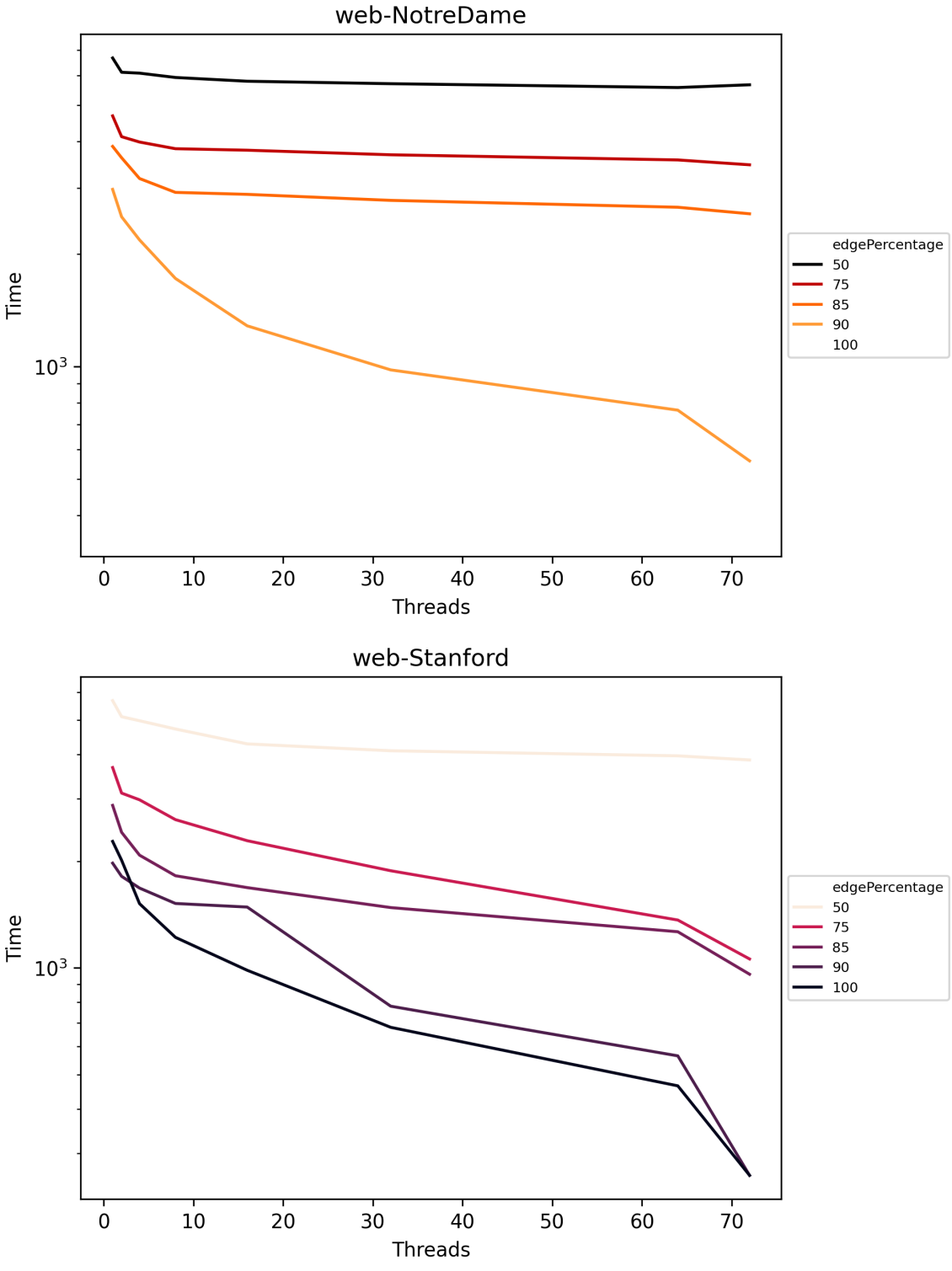


Figure 8.10: Scalability analysis of proposed Shared Memory SCC update Algorithm

	Threads	Time in Seconds	Edge Insertion %
web-BerkStan			
	1	4675	100
	2	3613	100
	4	2580	100
	8	2220	100
	16	1285	100
	32	980	100
	64	665	100
	72	460	100
	1	4975	90
	2	3813	90
	4	2980	90
	8	2420	90
	16	1685	90
	32	1080	90
	64	965	90
	72	960	90
	1	5875	85
	2	5613	85
	4	5180	85
	8	4220	85
	16	4085	85
	32	3980	85
	64	3665	85
	72	3260	85
	1	6675	75
	2	6113	75
	4	5980	75
	8	5820	75
	16	5785	75
	32	5680	75
	64	5565	75
	72	5460	75
	1	8675	50
	2	8113	50
	4	8080	50
	8	8020	50
	16	7985	50
	32	8098	50
	64	8065	50
	72	8060	50
web-Google			
	1	4475	100
	2	4113	100
	4	3580	100
	8	2920	100
	16	2285	100
	32	1980	100
	64	1665	100
	72	1460	100
	1	5975	90
	2	4813	90
	4	4180	90
	8	3420	90
	16	3185	90
	32	2780	90
	64	2165	90
	72	1760	90

	Threads	Time in Seconds	Edge Insertion %
web-Google			
	1	5675	85
	2	5113	85
	4	4980	85
	8	4220	85
	16	3785	85
	32	3380	85
	64	2665	85
	72	1960	85
	1	6175	75
	2	5913	75
	4	5900	75
	8	5820	75
	16	5685	75
	32	5580	75
	64	5465	75
	72	5360	75
	1	10675	50
	2	10113	50
	4	9080	50
	8	9020	50
	16	8985	50
	32	8698	50
	64	8565	50
	72	8460	50
web-NotreDame			
	1	2675	100
	2	2013	100
	4	1718	100
	8	1320	100
	16	1085	100
	32	780	100
	64	565	100
	72	360	100
	1	2975	90
	2	2513	90
	4	2180	90
	8	1720	90
	16	1285	90
	32	980	90
	64	765	90
	72	560	90
	1	3875	85
	2	3613	85
	4	1180	85
	8	2920	85
	16	2885	85
	32	2780	85
	64	2665	85
	72	2560	85
	1	4675	75
	2	4113	75
	4	3980	75
	8	3820	75
	16	3785	75
	32	3680	75
	64	3565	75
	72	3460	75

	Threads	Time in Seconds	Edge Insertion %
web-web-NotreDame			
	1	6675	50
	2	6113	50
	4	6080	50
	8	5920	50
	16	5785	50
	32	5698	50
	64	5565	50
	72	5660	50
web-Stanford			
	1	2275	100
	2	2013	100
	4	1518	100
	8	1220	100
	16	985	100
	32	680	100
	64	465	100
	72	260	100
	1	1977	90
	2	1812	90
	4	1683	90
	8	1521	90
	16	1481	90
	32	782	90
	64	564	90
	72	267	90
	1	2876	85
	2	2414	85
	4	2083	85
	8	1822	85
	16	1687	85
	32	1481	85
	64	1264	85
	72	962	85
	1	3674	75
	2	3111	75
	4	2984	75
	8	2622	75
	16	2287	75
	32	1881	75
	64	1368	75
	72	1062	75
	1	5676	50
	2	5114	50
	4	4982	50
	8	4721	50
	16	4284	50
	32	4099	50
	64	3967	50
	72	3861	50

Table 8.4: Execution time to update SCC in seconds

Chapter 9

Future Work

9.1 Future Work

The future research direction in developing scalable graph algorithms for dynamic networks will be :

1. *Studying the Performance on Distributed Memory and GPU:* This dissertation has proposed shared memory implementation for updating Connected Components, Minimum Spanning Tree, Single Source Shortest Path, Page Rank, and Strongly Connected Components. There is also a shared memory implementation of Permanence to detect overlapping communities. This dissertation was more focused on designing algorithms for shared memory implementation. The shared memory implementation of SSSP was extended to GPU and it was observed that GPU implementation gives better scalability. The author has plans to extend all the proposed implementation to GPU and study their performance in a single/multi GPU environment. SCC implementation was initially started with a distributed approach, however, there were a couple of roadblocks such as the gathering of SCC's from the different processors and finding the overall SCC, later the shared memory implementation was proposed. Implementation of all proposed approaches in this dissertation on distributed approach and their performance analysis can help design/develop dynamic algorithms for large networks.

2. *Scalable Implementation of Community Detection using Permanence & Constant Communities:*

   This dissertation provides a shared memory implementation of community detection using permanence, however, the results on networks show poor scalability. While profiling the im-

plementation, it was observed that the permanence calculation is very expensive and leads to poor scalability. The future work in this research involves optimizing the permanence calculation and improving the scalability. The Permanence implementation depends on initial seeding to detect accurate communities. Having a poor seeding in the initial stage always gives bad communities, the future work also involves finding a good seeding technique that can be generic and based on the input network and the algorithm should be able to determine which can give better community structure. The proposed implementation in this dissertation is restricted to static networks. The next step after developing scalable implementation is to extend the proposed implementation to detect communities on dynamic networks.

Constant Communities: Community detection approaches are inherently stochastic which means results of community detection can vary based on the algorithm and parameters used. It has been observed while doing scalability results for shared memory implementation of Permanence while doing different thread combinations each generates a different set of community. The constant communities are the group of vertices that are always grouped together and are stable. The common and one of the most used solutions is to run the input dataset thru various different community detection algorithms and also run it multiple times. The next step is to see which nodes always fall in the same community for multiple runs and mark them as constant communities. This solution is expensive and it is not possible to scale it for large networks. The author plans to work on this problem and develop a shared memory implementation to detect constant communities in large networks.

3. *Improving Scalability of the SCC Algorithm:*

   Shared memory implementation of SCC shows poor scalability on large networks. Graphical Processing Units (GPUs) provide more computing power and memory bandwidth than CPUs [85]. GPUs can be a good candidate for updating the SCC of dynamic networks. Recently there has been a plethora of graph algorithms that have been implemented on GPUs such as [85]. Sha et al. [86] have an implementation of Dynamic Graph Analytics on GPUs, however, their software allows only three network properties such as BFS, connected component, and PageRank. The author plans to work on the extension of the shared memory implementation of SCC to GPUs.

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] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, (USA), p. 262268, IEEE Computer Society, 2012.

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

[59] 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.

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

[61] D. Kirk *et al.*, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, vol. 7, pp. 103–104, 2007.

[62] P. J. Martín, R. Torres, and A. Gavilanes, "Cuda solutions for the sssp problem," in *International Conference on Computational Science*, pp. 904–913, Springer, 2009.

[63] "Konect network dataset – KONECT." `http://konect.uni-koblenz.de/`. Accessed on 2018-03-27.

[64] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, 2016.

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

[66] M. E. Newman, "Modularity and community structure in networks," *Proceedings of the national academy of sciences*, vol. 103, no. 23, pp. 8577–8582, 2006.

[67] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, 06 2009.

[68] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, p. P10008, oct 2008.

[69] S. Bhowmick and S. Srinivasan, "A template for parallelizing the louvain method for modularity maximization," *Dynamics on and of Complex Networks*, vol. 2, pp. 111–124, 04 2013.

[70] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo, "Scalable static and dynamic community detection using grappolo,"

[71] T. Chakraborty, S. Srinivasan, N. Ganguly, A. Mukherjee, and S. Bhowmick, "On the permanence of vertices in network communities," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1396–1405, ACM, 2014.

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

[73] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, p. P10008, Oct. 2008.

[74] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Phys. Rev. E*, vol. 78, no. 046110, 2008.

[75] J. Yang and J. Leskovec, "Overlapping communities explain coreperiphery organization of networks," *Proceedings of the IEEE*, vol. 102, no. 12, pp. 1892–1902, 2014.

[76] A. F. McDaid, D. Greene, and N. Hurley, "Normalized mutual information to evaluate overlapping community finding algorithms," Oct. 2011.

[77] J. Yang and J. Leskovec, "Overlapping community detection at scale: a nonnegative matrix factorization approach," in *Proceedings of the sixth ACM international conference on Web search and data mining*, pp. 587–596, 2013.

[78] P. K. Desikan, N. Pathak, J. Srivastava, and V. Kumar, "Divide and conquer approach for efficient pagerank computation," in *Proceedings of the 6th international conference on Web engineering - ICWE '06*, (Palo Alto, California, USA), p. 233, ACM Press, 2006.

[79] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

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

[81] S. Orzan, *On distributed verification and verified distribution*. PhD thesis.

[82] G. M. Slota, S. Rajamanickam, and K. Madduri, "Bfs and coloring-based parallel algorithms for strongly connected components and related problems.," in *IPDPS*, vol. 14, pp. 550–559, 2014.

[83] Y. Ji, H. Liu, and H. H. Huang, "iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, (Dallas, TX, USA), pp. 731–742, IEEE, Nov. 2018.

[84] G. M. Slota, S. Rajamanickam, and K. Madduri, "BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, (Phoenix, AZ, USA), pp. 550–559, IEEE, May 2014.

[85] 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.

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