Application of Graph Sparsification in Developing Parallel Algorithms for Updating Connected Components

Sriram Srinivasan, Sanjukta Bhowmick
Computer Science Department
University of Nebraska-Omaha
Email: sriramsrinivas@unomaha.edu, sbhowmick@unomaha.edu

Sajal Das
Computer Science Department
Missouri University of Science and Technology
Email: sdas@mst.edu

***Abstract*—Analyzing large dynamic networks is an important problem with applications in a wide range of disciplines. A key operation is updating the network properties as its topology changes. In this paper we present graph sparsification as an efficient abstraction for updating the properties of dynamic networks. We demonstrate the applicability of graph sparsification in updating the connected components in random and scale-free networks on shared memory systems. Our results show that the updating is scalable (10X on 16 processors for larger networks). To the best of our knowledge this is the first parallel implementation of graph sparsification. Based on these initial results, we discuss how the current implementation can be further improved and how graph sparsification can be applied to updating other network properties.**

***Index Terms*—dynamic networks, graph sparsification**

I. Introduction

Analysis of large scale networks is a crucial tool for studying systems of interacting entities that occur in diverse disciplines including bioinformatics [15], social sciences [23] and epidemiology [22]. The systems are modeled as networks (or graphs), where the vertices represent the entities and the edges, the pair-wise interactions between the entities. Analyzing properties of the network can provide insights into the properties of their underlying systems. Most real-world networks are dynamic, i.e. their topology and properties change over time. Updating the properties of dynamic networks is an important problem in network analysis.

One of the primary operations in network analysis is graph traversal. Traversal on unstructured graphs give rise to poor locality of access, leading to increase in computation time [13]. The problem gets exacerbated when the network is dynamic, i.e., the structure is changing with time.

Networks that are of a smaller size and denser connections will have more localized access patterns. They will therefore be faster to traverse. Thus the operations can be faster if they are performed on localized subgraphs, and then the results aggregated to apply to the larger network. This is the underlying principle of the very popular reduction approach in designing parallel algorithms.

The problem then translates to expressing graph algorithms in a reduction-like format. This can be achieved by using an elegant algorithmic technique known as *graph sparsification* [11]. In graph sparsification, the network is divided into subgraphs, which form the leaves of a balanced sparsification tree. The operations are performed on the subgraphs and then transferred across the levels of the tree to the root, which contains the entire graph. Graph sparsification techniques have been long studied in the theoretical context [11], [10], [8] and on updating dense graphs. However, there has been little work on actually implementing graph sparsification, and studying the practical utility of this method for updating properties of large sparse dynamic networks.

In this paper, we present graph sparsification as a useful abstraction for creating scalable algorithms for updating dynamic networks. We present the theoretical analysis and preliminary experimental results on updating connected components, and maintaining the corresponding spanning trees. Our results demonstrate that graph sparsification can indeed be used to design scalable algorithms. In particular, processing multiple edge insertions and deletions in parallel forms a bottleneck in many updating algorithms. Due to the localized computations in graph sparsification, large groups of edge insertions and deletions can be handled in parallel and therefore provide a faster update algorithm.

The remainder of this paper is organized as follows; In Section II, we provide the background on graph terminology, an overview of related work in dynamic networks and describe in detail the general graph sparsification technique. In Section III, we present the algorithmic steps for creating and updating connected components using graph sparsification, along with analysis of their memory requirements and computational complexity. In Section IV, we provide our experimental results on random and scale-free networks. We conclude in Section V with an overview of our future research plans.

II. Background

A. Graph Terminologies

A network (also referred to as a graph) is defined by a pair of sets $G = (V, E)$. Each edge $e \in E$ is represented by a pair of vertices i.e. $e = (u, v)$. The vertices u, v are known as the *endpoints* of e. A *path*, of length l, in a network G

is an alternating sequence $v_0, e_1, v_1, e_2, \ldots, e_l, v_l$ of vertices and edges, such that for $j = 1, \ldots, l$; v_{j-1} and v_j are the endpoints of edge e_j, with no edges or internal vertices. A *cycle* is a path whose start and end vertices are the same, i.e. $v_0 = v_l$. A graph is defined as a *tree* if it has no cycles. For our analysis we consider undirected graphs where the number of vertices $|V| = n$, the number of edges $|E| = m$ and that n is a power of 2. The analysis will also hold if n is not a power of 2. In that case, we simply consider the next higher power of 2.

B. Related Research in Dynamic Networks

A recent special issue of *Parallel Computing* on "Graph analysis for scientific discovery" [6] presents some of the latest advances in parallel algorithms for graphs, including dynamic networks. Connectivity-based algorithms such as the breadth first search on static networks are available for many HPC platforms including, distributed memory [24], [5], multicores [2], massively multithreaded machines [21] and GPUs [17]. Parallel implementation of a spanning tree using the Shiloach-Vishkin approach is given in [3]. Dynamic update of connected components on multicores is given in [18] and on massively multithreaded machines in [19]. To date, there are either parallel algorithms for static networks or sequential algorithms for dynamic networks. We only know of two projects STINGER [1], [9] and PHISH [20] that include parallel algorithms for dynamic updates of networks.

C. Overview of Graph Sparsification

Graph sparsification, introduced by Eppstein *et. al.* [11], is a divide and conquer technique to reduce the dependence on the edges in a graph, such that the time bounds for maintaining a graph property is commensurate to the computing time for sparse graphs. This technique is part of a larger set of methods, including clustering techniques [12] and randomized algorithm [14] that were developed for obtaining strong theoretical performance bounds on algorithms for updating the topological properties (such as connected components or minimum spanning tree) of dynamic graphs. However, parallel versions of these algorithms were studied only for a conceptual version of the parallel random access machine (PRAM) models [8] and no empirical results were reported. We now provide an overview of the mechanism of graph sparsification.

In the original paper by Epstein [11], first a vertex sparsification tree is created by recursively dividing the vertices into two equal halves. Assuming that the vertices are ordered from 0 to $n-1$ consecutively, then, at each division, two sets of vertices numbered from 0 to $n/2-1$ and $n/2$ to $n-1$ are formed. Therefore, nodes at distance i from the root will have $\frac{n}{2^i}$ vertices, and the height of the tree will be $i = logn$.

The edge sparsification tree is created from the vertex sparsification tree. Let a and b be two nodes in the vertex sparsification tree, containing the vertex sets V_a and V_b respectively. A node at level i in the edge sparsification tree E_{ab}, contains edges whose one endpoint is in V_a and the other in V_b. The parent of node E_{ab} is the node E_{cd}, where c and d are the parents of nodes a and b. The internal nodes will have three to four children. The edge sparsification tree will have the same height as the vertex sparsification tree.

Reduction Like Operations on Sparsification Trees. The edges of the graph sparsification tree only store the sparser subgraphs (known as *certificates*) that are relevant to the property under consideration. Formally, for a property P, a certificate for a graph G is the graph $\bar{G}$, such that G has property P, if and only if $\bar{G}$ has property P.

A *strong certificate* of the graph G is the sparser subgraph $\bar{G}$, if for a graph H with the same vertex set as G, the graph $G \cup H$ has property P, if and only if $\bar{G} \cup H$ has property P. Let A be the function applied to the graph G to create its certificate $\bar{G}$, i.e., $A(G) = \bar{G}$. A is *stable* if $A(G \cup H) = A(A(G) \cup H)$ and for a given edge e, the mapping $A(G-e)$ or $A(G+e)$ differs from $A(G)$ by $O(1)$ edges. A certificate is stable if it is created using a stable mapping [11].

If the certificates in the sparsification tree are strong and stable, then the certificates in the non-leaf nodes can be created using reduction like operations on their children. The root will contain the certificate for the entire graph. Furthermore, in case of stable certificates on dynamic graphs, change in an edge will change the certificate by at most a constant number of edges. The update operation due to the edge occurs at only one node and proceeds along path from the node where the change occurred to the root. So the maximum number of updates per changed edge is proportional to the height of the tree.

The term "graph sparsification" has also been used to to approximate a graph by a sparse graph. One of the well known methods is spectral partitioning [4]. The difference from the technique proposed here is that the entire graph is approximated to a smaller one. In the method proposed here, we are not removing any edges only updating them across the tree.

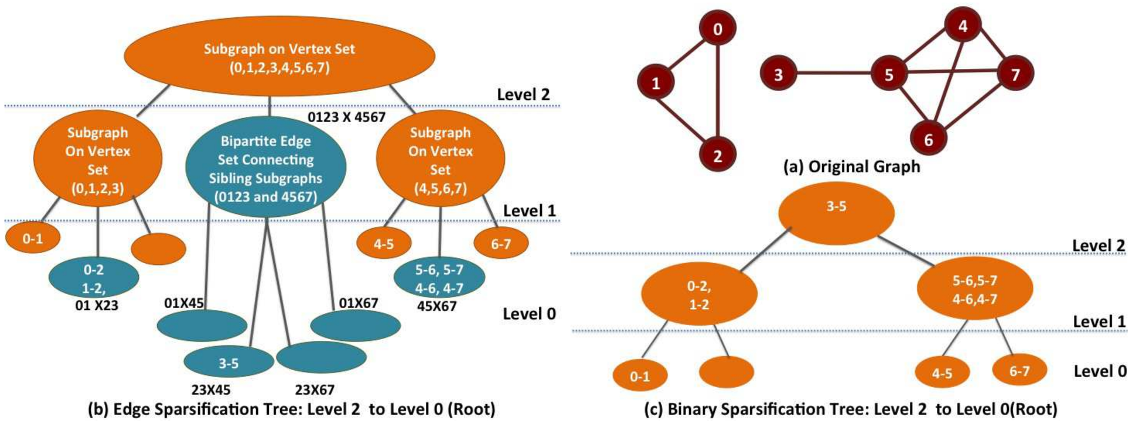
In the remainder of the paper, in order to distinguish between the elements in graph and elements in the tree, we will refer to the nodes in sparsification tree as "nodes" and the vertices in the original graph as "vertices".

III. Graph Sparsification for Updating Connected Components

We now present our main contribution; updating the connected components of a network using graph sparsification. Our goal is to identify the vertices in the same connected component and also maintain a spanning forest connecting vertices in the same components. The main steps to the process are as follows; *(i)* creating the sparsification tree, *(ii)* creating the initial certificates; i.e. the spanning tree and identifying the connected components and *(iii)* maintaining the connected components and spanning trees as the network changes.

A. Creating the Sparsification Tree.

We modify the original sparsification tree into a binary tree. This modification simplifies the implementation while retaining the operational advantages. To convert the original



Subgraph on Vertex Set (0,1,2,3,4,5,6,7)
Level 2
0123 X 4567
Subgraph On Vertex Set (0,1,2,3)
Bipartite Edge Set Connecting Sibling Subgraphs (0123 and 4567)
Subgraph On Vertex Set (4,5,6,7)
Level 1
0-1
0-2 1-2, 01 X23
4-5
6-7
5-6, 5-7 4-6, 4-7 45X67
01X45
01X67
Level 0
3-5
23X45
23X67
(b) Edge Sparsification Tree: Level 2 to Level 0 (Root)
(a) Original Graph
0
1
2
3
4
5
6
7
3-5
Level 2
0-2, 1-2
5-6,5-7 4-6,4-7
Level 1
0-1
4-5
6-7
Level 0
(c) Binary Sparsification Tree: Level 2 to Level 0(Root)

Fig. 1: **Creation of Sparsification Trees.** (a) The original graph. (b) Original form of edge sparsification tree. (c) Modification to binary sparsification tree

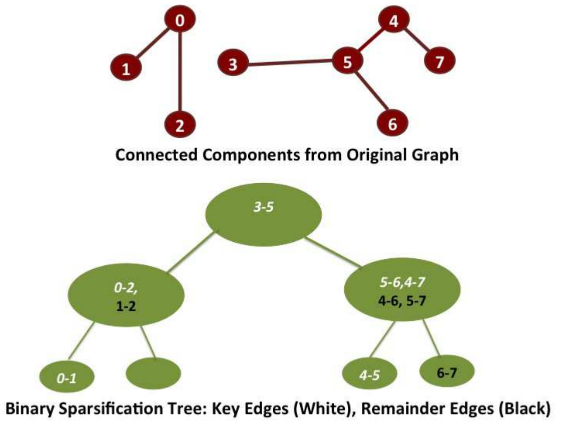
tree to the binary form, we term the nodes in the sparsification tree that contain a bipartite set of edges connecting two disconnected subgraphs as bridge nodes and the other nodes as non-bridge nodes. Then in each bridge node we store the union of the edges from their non-bridge descendants. After the union, the sparsification tree becomes binary and all nodes are non-bridge nodes. The number of nodes in the tree are $n-1$, where the number of vertices is n.

Although the original sprasification method only stores the edges relevant to the property, in our implementation, we store all the edges of the network. We classify the edges into two types. We term the first set as *key edges*. These edges form the subgraphs that are relevant to the property under consideration, such as the edges that form the spanning tree in each of the connected components. We term the other edges in the network as *remainder edges*. We assume that the initial spanning trees have already been computed by some other efficient algorithm, and so we already have the key and remainder edges marked accordingly. Therefore, this stage consists of creating the binary sparsification tree and assigning the edges, along with their type, in the appropriate node.

The second part in creating the tree is to allocate the edges at the proper nodes. Each node in the tree is identified by two indices: the *level index* l,(starting from 0 at the leaves to $logn - 1$ the root), and the *position index* p (the leftmost node per level is marked as position 0, and the numbers keep increasing towards the right). Assuming that the vertices are numbered consecutively from 0 to $n - 1$, an edge with end points having id u and v will be placed in node with level $l - 1$ and position $s - 1$ if $\lfloor \frac{u}{2^l} \rfloor = \lfloor \frac{v}{2^l} \rfloor = s$. Figure 1 shows an example of a graph and the original and the binary sparsification tree with key and remainder edges.

B. Creating the Graph Certificate.

The next step is to create the certificates from the key edges. We maintain two adjacency lists, one for the network formed by the remainder edges (the *remainder graph*) and the other for the network formed by the key edges (the *certificate graph*). We create one remainder graph and one



0
1
2
3
4
5
6
7
Connected Components from Original Graph
3-5
0-2,
1-2
5-6,4-7
4-6, 5-7
0-1
4-5
6-7
Binary Sparsification Tree: Key Edges (White), Remainder Edges (Black)

Fig. 2: **Key and Remainder Edges in Sparsification Tree** Sparsification trees from the network in Figure 1. The bottom figure shows the key and remainder edges at each node.

certificate graph at each level of the tree consisting of only the edges found at that level. The union of the remainder (certificate) graph from all the levels gives the complete remainder (certificate) graph. The vertices that are in the same spanning tree in the certificate graph are in the same component. Union of the complete remainder and certificate graph gives the entire network. The subgraphs at each tree node do not overlap. Therefore creating the adjacency list can be done in parallel over all the tree-nodes.

If we want to know whether two vertices are in the same component we can traverse the certificate graph over the different levels. However, as discussed earlier, traversal is an expensive operation. We therefore also list the component of the vertex, and keep an updated list of the components across the levels.

At each level the component of a vertex is updated if it is connected by a key edge to another vertex with a lower numbered component. This process can be done in parallel over the nodes in the same level, because the subgraphs in the same level do not overlap. However, the update from one level to the next has to be done sequentially, since the update at a parent node is dependent upon the components formed at its children. Because of the structure of the tree, the scope for parallelization decreases as we go further up, i.e. less number of nodes. Therefore, having more edges in the lower levels and fewer edges in the top of the tree can reduce the time to update the components. Figure 2 shows an example of how the key and remainder edges are stored.

C. Maintaining the Components.

The final part is updating the certificate as edges are added or deleted to the network. This process consists of the following two steps;

Identifying Edges When a new edge (u, v) is inserted, we check whether u and v have the same component. If so, then the edge is a remainder edge and is added to the remainder graph at the appropriate tree node. If the edge is a key edge, then it is stored in an array for insertion edges, at the appropriate tree node, to be added later. We do not insert new key edges immediately because simultaneous addition of edges can cause a cycle in the spanning tree.

When a new edge (u, v) is deleted, we check the certificate graph at the appropriate tree node, to see if the edge was a key edge. If so, it is deleted from the certificate graph, otherwise it is deleted from the remainder graph. If the edge was a key edge, we store it in an array for deleted edges. We will use this information later to update the components. This process of identifying the edges can be done in parallel at all the tree nodes.

Updating the Components Once the edges have been classified, we update the components and the certificate graph at each level. This process has to be done sequentially level by level since the certificates at the lower levels affect the certificates at the higher level. At each level we check whether the components for the nodes marked for insertion have already been connected at a lower level. If the components are still disconnected, the edge is added to the certificate graph, otherwise it is added to the remainder graph. Note that since we are adding the edges level by level, we remove the possibility of creating a cycle in the spanning tree.

Note that this insertion process is different from the "hook" and "graft" operations in the Shiloach-Vishkin algorithm. This is because we do not "hook" by connecting the roots of two trees, but can connect any two of the vertices. The vertices at the lower levels of the sparsification tree have a higher priority of getting connected. Because we do not use hook, therefore, grafting the tree also does not reduce the computation.

We now have to identify and separate the components due to edge deletion. To do this, we consider the certificate at the next lower level from where the deleted edge is allocated, since this was the last level where the components were separated. We traverse the certificate in that level with each end point of the deleted edge as the source. The vertices reached through the traversal will be in the component of the respective end point. This information is updated across the levels of the sparsification tree. This operation is done in parallel at each level and then updated across the levels sequentially.

Finally there is a possibility that the deleted components can be joined again by a remainder edge. This process is the most expensive since we might have to check all the remainder edges. We use the following scheme for "repairing" the deleted components as follows; In the level that the edge is deleted, we traverse the network represented by all the remainder and union edges upto that level, to see if there is a remainder edge that connects the two components. This process reduces the search space because we are considering only the remainder edges that are connected to one of the components, however as we go up the higher levels the time for traversal increases. Moreover, the edge connecting the two components can be at a higher level, therefore to correctly repair the break in components we should check all the levels. Currently, to lower the computation costs, we are only checking for edges at one level only. Figure 3 shows an example of insertion and deletion in the network

D. Memory Requirements and Time Complexity

Every edge, key or remainder, is stored at one and only one of the tree nodes, requiring a total storage of $O(m)$. The components of the vertices are stored at each level requiring a storage of $O(nlogn)$. Most of the operations are on the certificates, therefore it is important to keep the number of key edges low.

Creating the Sparsification Tree. Each edge(u, v) requires $O(log(u))$ operations to find the appropriate tree nodes, where the numerical value of u is greater than that of v. However, the major portion of the time is dependent on the I/O time for reading the network data.

*Creating the Graph Certificate.*The time for adding the edges to their respective graphs is constant and can be done in parallel over all the tree nodes. Assume we have p processors. If we assume that the edges are evenly distributed across the tree nodes, then the time to add them to the remainder and certificate graphs is $O(m/n)/p$.

The time for updating the components is done level by level. At each level the vertices in the key edges are assigned a new component , in parallel, if they are connected to a vertex that has a lower component. This process is repeated over several iterations, until at that level, the components can no longer be changed. If we assume that there are m/n edge at each node, then with p processors, the time taken will be $\sum_{i=0}^{i=log(n)-1}(\frac{t*O(m/n)}{q})$, where $q = p$ if $p \geq \frac{n}{2^{i+1}}$ and $q = \frac{n}{2^{i+1}}$ otherwise, and t is the number of iterations.

Updating the Certificate. Let the number of edges changed in the network be k. In practice the number of edges in the spanning tree is much lower than the total number of edges and this ratio is also reflected in the type of changed edges. We assume of the k changed edges, only $r \ll k$ edges are identified as key edges to be inserted or deleted. We also assume that the edges are evenly distributed among the nodes of the sparsification tree, so that there are k/n new edges per node and out of them r/n edges are key edges. Checking

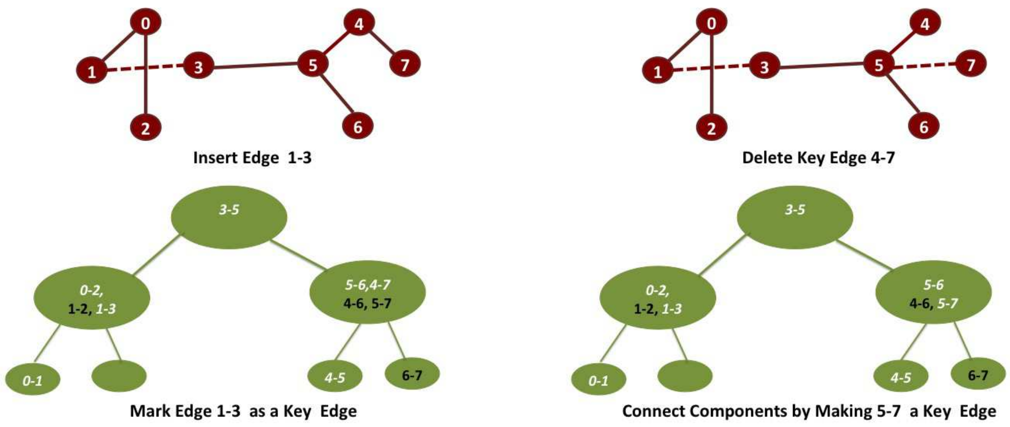


Fig. 3: **Example of Inserting and Deleting Edges**. Left: Edge 1-3 is inserted and identified as a key edge connecting two components. Right: Edge 4-7 is deleted, but a remainder edge 5-7 replaces it as the connecting key edge.

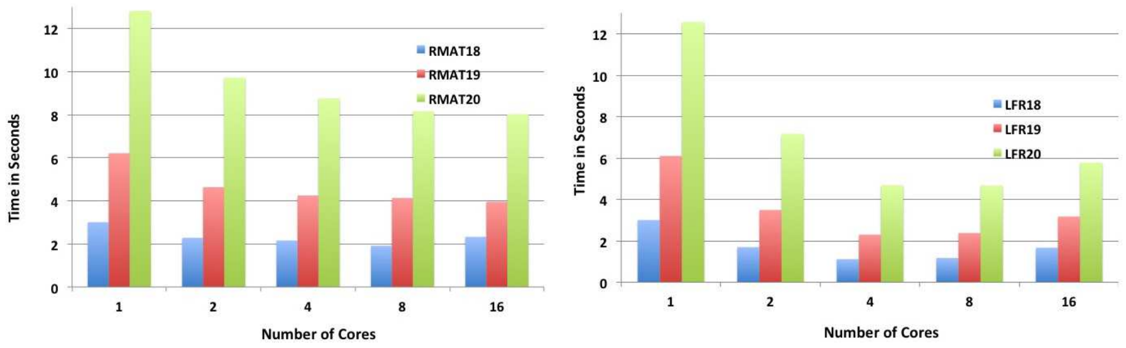


Fig. 4: **Time to Create the Certificates** Left: RMAT graphs; Right: LFR graphs

whether the edges should be included in the certificate or the remainder graph can be done in parallel over the tree nodes. This requires time of the order of $O(k/n)/p$ for insertions (since checking for same components is a constant operation). For deletions we potentially have to check all the edges in the certificate at that level. The maximum size of the certificate at a given level is equal to the number of vertices per node in that level, therefore at level i the size of the certificate is $\frac{n}{2^{i+1}}$. Thus the time to identify whether a deleted edge is key or not is $O(\frac{k}{n}\frac{n}{2^{i+1}})/p$.

Updating the components due to edge insertion is a similar operation to creating the certificates in the second stage, except fewer iterations are needed and requires time $\sum_{i=0}^{i=log(n)-1}(\frac{t*O(r/n)}{q})$. The traversal to separate the components can be done in parallel for each end points at each level. The number of end points per level is proportional to the number of key edges at that level, which is r/n. The time to traverse all the certificates, and update the components after deleting key edges, will be $\sum_{i=0}^{i=log(n)-1} O(\frac{r}{n}\frac{n}{2^{i+1}})/q$.

For the repairing step, the traversal has to be done over the entire network at that level, not just the certificate. If the edge density of the network is d, then the size of the network at level i will be approximately $d\frac{n}{2^{i+1}}$. The total time to traverse the networks at all the levels will be $\sum_{i=0}^{i=log(n)-1} O(d\frac{r}{n}\frac{n}{2^{i+1}})/q$.

The number of sequential steps is only $logn$, and the graphs at the lower levels are smaller in size and can be traversed quickly. These properties render update using graph sparsification a very fast operation. However, the opportunity for parallelization decreases at the higher levels of the tree to being sequential at the root. Thus if most of the edges are at the higher levels, then the advantages due to graph sparsification are diminished. We have also assumed in the analysis that the edges and the corresponding work across the nodes to be evenly distributed. This is rarely the case in practice. However, the most effective utilization of graph sparsification will be when the vertices are numbered such that most edges are in the lower levels of the tree and the work load evenly distributed across the nodes at the lower levels.

IV. Experimental Results

We now present our experimental results for updating connected components on a shared memory system. Our test suite consists of three Erdos-Reyni networks, created using the RMAT [7] code, and three scale-free networks using the LFR code [16]. The RMAT networks were created by setting parameters as $a = b = c = d = .25$. The LFR networks were created by setting the mixing parameter $\mu = .1$, which ensured that they had well defined communities. The number of edges and vertices of the size graphs are given in Table I. Both the LFR and RMAT network of the same scale (18,19,20)

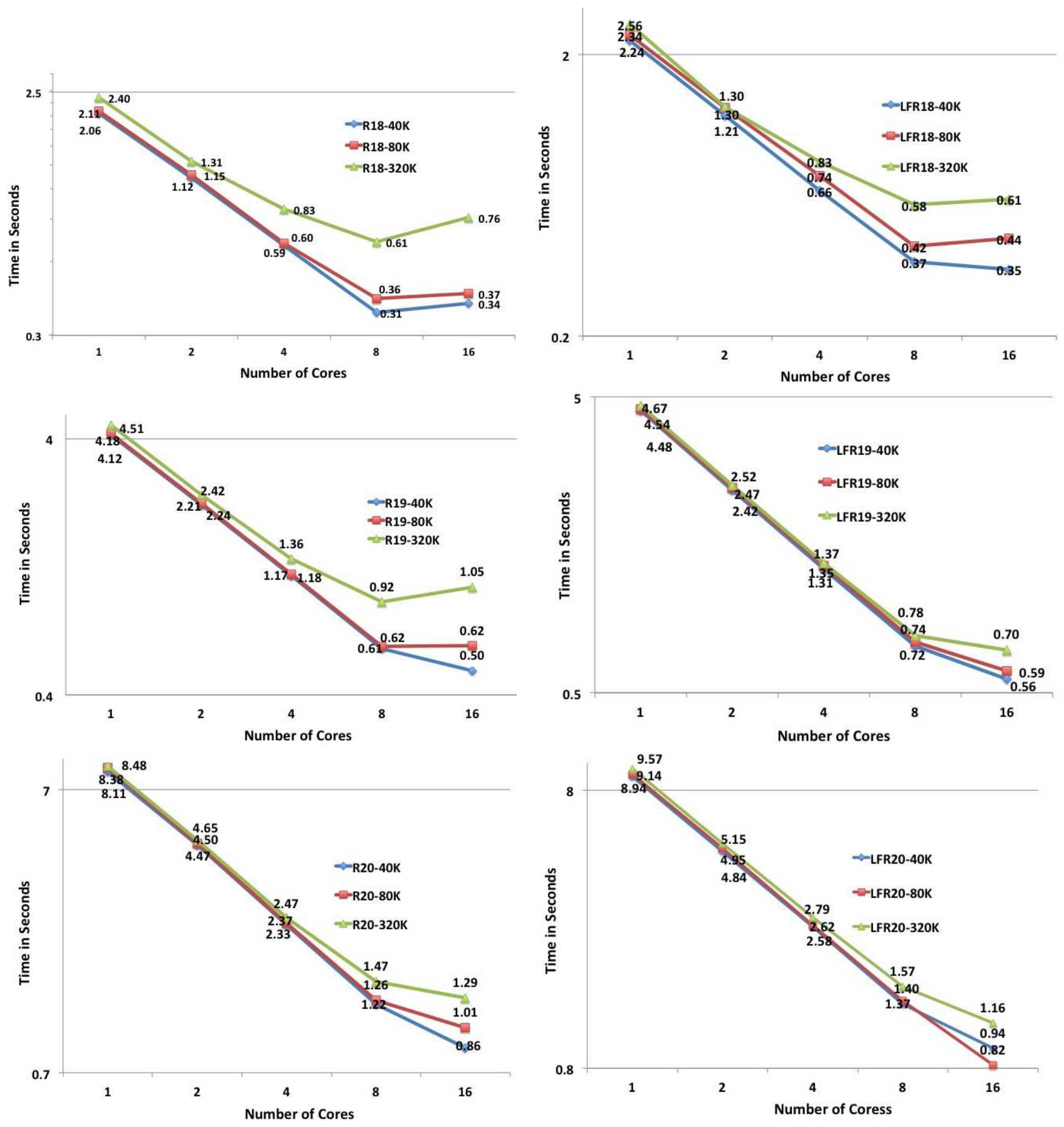


Fig. 5: **Time for Updating the Networks** Left: RMAT Network; Right: LFR Network; From Top to Bottom: Scale 18,19, 20. Each chart has three plots showing the time to update 40K, 80K and 320K edges.

have the same number of vertices, but the edges in the LFR networks is slightly larger. For each network we generated a set of updated edges of size 40K, 80K and 160K. All the edges in the updated set were unique. We ran the experiments on the Tusker machine available at Holland Computing Center. The machine consists of 6,784 cores interconnected with Mellanox QDR Infiniband along with 523TB of Lustre storage. Each compute node is an R815 server with at least 256 GB RAM and 4 Opteron 6272 (2.1 GHz) processors.

Since the time for creating the sparsification tree is primarily dependent of the I/O we analyze time to create the initial certificates (Figure 4) and the time to update them (Figure 5).

We see from Figure 4 that the time for creating the the initial certificate does not scale well with increasing number of processors. This is because there is no parallelization opportunity at the top levels of the sparsification tree. Moreover neither the random nor the scale-free networks were easily partitionable into equal divisions with low number of cut-edges across them. Therefore the time at the top levels dominate the computation cost. The reduction of the computation cost is due to the parallelization at the lower levels.

TABLE I: **Test Suite of Networks.**

Name	Vertices	Edges
RMAT18	262,144	4,194,128
RMAT19	524,288	8,388,438
RMAT20	1,048,576	16,777,196
LFR18	262,144	5,121,908
LFR19	524,288	10,262,022
LFR20	1,048,576	20,507,134

However, as seen in Figure 5, updating the certificate is very scalable, and the scalability improves with the size of the network. We obtain 10X speedup on 16 processors for the scale 20 networks and 40K edges We also note that increasing the set of updates only causes a slight increase in the update time. We believe that this scalability is due to two reasons. First most of the updated edges are remainder edges, and therefore can be processed in parallel over all the tree nodes. Of the key edges, the insertions take at most $logn$ steps, and the deletions also need to traverse over relatively small networks. The only expensive operation is the repairing. However, in this implementation we have restricted it to only one level. Increasing the number of levels to be checked for repairing can hamper the scalability.

V. Discussion and Future Research

In this paper we presented graph sparsification as an efficient abstraction for updating dynamic networks. Our experiments on random and scale-free networks show that updating connected components using graph sparsification is indeed scalable. However, the step for creating the initial certificate is not as scalable. This is because of the uneven distribution of the edges in the tree nodes. We plan to investigate different graph partitioning techniques to develop a method for distributing the edges evenly, and thereby improving the scalability of this step.

Although our initial experiments were on shared memory machines, graph sparsification is general enough to be used on any parallel platform, and also on heterogenous systems. The main issue in designing an effective algorithm is to determine how the edges are distributed across the tree nodes. Graph sparsification can be used to update other network properties (for example theoretical work on updating minimum spanning trees already exist), so long as efficient data structures are maintained to quickly process the updates. We believe that due to its flexibility graph sparsification can be an useful building block for dynamic network algorithms. In particular, the operations on creating the sparsification tree and creating the initial certificates will be similar across most algorithms. Identifying efficient data structures for maintaining the results at each level can further point to other common building blocks. As part of our future research we plan to develop graph sparsification based algorithms for updating other network properties and also explore techniques to improve the performance and scalability of those algorithms.

Acknowledgment

This research was funded by NSF-CCF; award numbers 1533918 and 1533881.

References

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

[2] 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, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[3] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006, Sept. 2005.

[4] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng. Spectral sparsification of graphs: Theory and algorithms. *Commun. ACM*, 56(8):87–94, Aug. 2013.

[5] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[6] A. Buluç, L. Oliker, and J. Gilbert. Special issue on graph analysis for scientific discovery. *Parallel Computing*, 47:1 – 2, 2015. Graph analysis for scientific discovery.

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

[8] S. Das and P. Ferragina. An erew pram algorithm for updating minimum spanning trees. volume 9, pages 111–122, 1999.

[9] D. Ediger, R. McColl, E. J. Riedy, and D. A. Bader. STINGER: high performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*, pages 1–5, 2012.

[10] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms, 1999.

[11] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification&mdash;a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, Sept. 1997.

[12] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 252–257, New York, NY, USA, 1983. ACM.

[13] B. Hendrickson and J. W. Berry. Graph analysis with high-performance computing. *Computing in Science and Engineering*, 10(2):14–19, 2008.

[14] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 519–527, New York, NY, USA, 1995. ACM.

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

[16] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Phys. Rev. E*, 78(046110), 2008.

[17] 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, pages 52–55, New York, NY, USA, 2010. ACM.

[18] K. Madduri and D. A. Bader. Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[19] R. McColl, O. Green, and D. A. Bader. A new parallel algorithm for connected components in dynamic graphs. *High Performance Computing (HiPC)*, pages 246–255, 2013.

[20] J. Plimpton and T. Shead. Streaming data analytics via message passing with application to graph algorithms. *Available at http://www.sandia.gov/ sjplimp/phish/papers.html*.

[21] E. J. Riedy and D. A. Bader. Massive streaming data analytics: a graph-based approach. *ACM Crossroads*, 19(3):37–43, 2013.

[22] E. Stattner and N. Vidot. Social network analysis in epidemiology: Current trends and perspectives. In *Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on*, pages 1–11, May 2011.

[23] S. Wasserman and K. Faust. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[24] 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*, page 25. IEEE Computer Society, 2005.