Efficient Anytime Anywhere Algorithms for Closeness Centrality in Large and Dynamic Graphs

Eunice E. Santos, John Korah, Vairavan Murugappan, Suresh Subramanian

Department of Computer Science
Illinois Institute of Technology, Chicago, USA
{esantos2, jkorah3}@iit.edu, {vmuruga1, ssubra20}@hawk.iit.edu

Abstract*—Recent advances in social network analysis methodologies for large (millions of nodes and billions of edges) and dynamic (evolving at different rates) networks have focused on leveraging new high performance architectures, parallel/distributed tools and novel data structures. However, there has been less focus on designing scalable and efficient algorithms to handle the challenges of dynamism in large-scale networks. In our previous work, we presented an overarching anytime anywhere framework for designing parallel and distributed social network analysis algorithms that are scalable to large network sizes and can handle dynamism. A key contribution of our work is to leverage the anytime and anywhere properties of graph analysis problems to design algorithms that can efficiently handle network dynamism by reusing partial results, and by reducing re-computations. In this paper, we present an algorithm for closeness centrality analysis that can handle changes in the network in the form of edge deletions. Using both theoretical analysis and experimental evaluations, we examine the performance of our algorithm with different network sizes and dynamism rates.

Keywords-social network analysis; parallel and distributed processing; centrality analysis; dynamic graphs; anytime algorithms

I. INTRODUCTION

Social networks provide the capability to leverage graph algorithms and other computational methodologies to rigorously analyze various social phenomena such as social influence. Depending on what relationships are depicted by the arcs/edges of a social network, the graph can be represented as directed or undirected. A number of social network services such as Twitter, LinkedIn, and a myriad of other similar applications have witnessed explosive growth in the past decade. As a result, these domains generate dynamic network information in the order of millions of nodes and edges. Availability of such large data sets, while significantly extending our understanding of the underlying social phenomenon, has also created new challenges for social network analysis (SNA) research. We now describe the following key issues in performing SNA on large and dynamic social networks.

The most important issue is network size. As the network size grows, computation time and resources required for measuring SNA metrics may increase dramatically[1][2]. Computation time can significantly constrain the utility of SNA techniques in large-scale networks, especially for time critical applications. A number of recent papers have proposed various methodologies such as parallel and multithreaded tools, novel data structures and efficient graph libraries to perform large-scale graph analysis [3][4][5][6]. While parallelization techniques solve some of the scalability issues, there are other issues such as network dynamism that need to be better addressed. Also, these methodologies do not generally prescribe algorithm designs that take into account tradeoffs between memory availability and performance.

In many SNA tasks such as real-time social media analytics, disaster management, etc., the underlying network is continuously evolving, with nodes and edges being added and/or removed, during analysis. One approach used to address this issue is to restart the analysis of the modified graph from scratch [7]. While this method can be feasible for smaller graphs with occasional network changes, restarting the entire analysis from scratch in larger real-time networks with frequent updates will often yield poor performance. Another method is to analyze a static snapshot of the dynamic network. However, this will only yield approximate results which, depending on the rate of change of the network, can quickly become obsolete. Although there are some graph processing methods[6][8] that are sensitive to networks changes, their main objective is to reduce the load imbalance caused by dynamism. We believe less attention has been given to formulating critical algorithm designs that can truly adapt to different rates of changes in the social networks by efficiently incorporating the new information, and avoiding massive re-computations. Finally, as discussed in the background section, many computational platforms such as shared-memory, distributed-memory, massively multithreaded and other specialized architectures are leveraged to perform large-scale graph analysis. In order to take advantage of various computational platforms, any graph analysis algorithm or framework has to be compatible with these environments.

In order to address these challenges, a parallel/distributed anytime anywhere methodology for large and dynamic social network analysis was proposed by Santos *et al.*[1], [9]. In the anytime anywhere approach, the large social network graph is decomposed into smaller sub-graphs. Each sub-graph is analyzed and refined over time, while providing nontrivial intermediate results. The quality of these results are

monotonically non-decreasing with respect to the computation time. Moreover, the dynamic changes in the network occurring over the course of analysis are continuously incorporated. In the anytime anywhere methodology, the term *anytime* refers to the capability of the algorithms to provide nontrivial partial results that will get refined over time. The term *anywhere*, on the other hand, refers to the capability to incorporate changes to the network whenever it happens and to propagate the effects of these changes to the whole network. We demonstrated our anytime anywhere approach for centrality measurements [1] and maximal clique enumeration problem [2] as part of our previous work. In this work, we provide algorithm designs that can adapt to edge deletions efficiently, and validate this approach for the problem of closeness centrality calculations in social networks. Furthermore, we have provided both theoretical and experimental evaluations of the proposed algorithm. Note that our previous algorithm designs[2] for edge deletions were specific for the maximum clique problem.

II. BACKGROUND

A number of graph libraries [4][10], computational frameworks [3][7], tools [5][6], and data structures[3][10] have been proposed for large-scale graph analysis. However, even with the existing computational capabilities researchers and analysts are overwhelmed with the ever-increasing size of the social network datasets. In addition to the enormous scale, these data sets exhibit various degrees of dynamism, which further impedes the network analysis. Some of the prominent single machine, shared-memory based graph analysis tools are LEMON [11], Pajek [1], igraph [12], NetworkX [13], UCInet [14], etc. Pajek and UCInet are graph analysis, software tools that provide implementations of various graph analysis algorithms, and provisions for graph visualization. LEMON, NetworkX and igraph are graph libraries that provides APIs and other useful graph data structures for performing graph analysis. Due to issues such as limited memory size, these tools can only provide analysis for small to medium scale (hundreds of thousands of nodes) graphs. Additionally, these systems are not designed to handle dynamic graphs.

MultiThreaded Graph Library (MTGL) [10] and GraphCT [3] provide graph analysis algorithms and data structures for massively multithreaded, shared-memory platforms. However, massively multithreaded, shared-memory tools are generally developed for specialized supercomputer architectures, such as Cray Multi-Threaded Architecture (MTAs) and eXtreme MultiThreaded Machine (XMTs). Also, they are not suited for distributed-memory clusters. The primary focus of massively multithreaded graph libraries is to provide high performance graph data structures and algorithms for large-scale graph analysis that are fine tuned for massive shared-memory and many thread processor architectures. Our anytime anywhere framework, on the other hand, is computational platform independent and can be used to develop dynamic large-scale algorithms for both cluster computing platforms and other specialized architectures.

MapReduce[15] is a popular framework that provides a simple, scalable and fault tolerant programming model for developing parallel applications. It is specifically designed for processing large amounts of data stored on commodity clusters. Large-scale graph analysis tools such as Surfer [5] and PEGASUS [4] were developed based on the MapReduce framework. However, most graph analysis algorithms are iterative in nature [16], and MapReduce systems are not well suited for such tasks. Pregel [7], a vertex-centric and Bulk Synchronous Parallel (BSP) model based graph analysis platform, was designed to address these issues. In this model, the computation is divided into a sequence of iterations or supersteps. During each superstep, a user defined function is performed at each graph vertex. Messages are exchanged between the vertices at the end of each superstep. This iterative framework can be used to formulate a number of graph algorithms. In addition, a user defined function can be invoked for each vertex, theoretically in parallel, so the maximum number of parallel tasks in the system could be as high as the number of vertices in the graph, thereby making this framework highly scalable. This framework provides ease of programming due to the synchronicity imposed by the supersteps. Giraph[2], Hama[3], Mizan [6] and Pregelix [17] are some of the popular Pregel based graph analysis systems [18]. These systems provide efficient data structures, APIs, and platforms for large-scale graph analysis. However, MapReduce and Pregel are programming frameworks, and do not specify efficient algorithm designs for handling dynamic graphs.

A large part of existing work on SNA algorithms, especially for computing shortest path related metrics for dynamic graphs [19][20], is focused on providing efficient serial algorithms. While these solutions perform well for the analysis of small to medium sized graphs, they are not scalable and are usually inefficient for large-scale graph analysis. While parallel/distributed graph analysis algorithms can overcome the inherent scalability issue in serial algorithms, designing such algorithms involve performing several challenging tasks such as efficiently partitioning the graph to balance workload, handling communication overhead, etc. An anytime anywhere framework addresses these issues efficiently. We have demonstrated our anytime anywhere approach and framework in previous work by developing and evaluating SNA algorithms for centrality analysis [9] and maximum cliques [2]. Specifically, we demonstrated the capability of our framework to deal with increase and decrease in edge weight during centrality analysis [1]. We have also designed algorithms to deal with

[1] http://mrvar.fdv.uni-lj.si/pajek/

[2] http://giraph.apache.org/

[3] http://hama.apache.org/

addition and deletions of edges during maximal clique enumeration [2]. In this paper, we further extend our capability to analyze dynamic graphs by designing a closeness centrality algorithm that can handle edge deletions. Although we have developed anytime anywhere algorithms to handle edge deletions for maximum clique enumeration, edge deletions generate wider network ripple effects in closeness centrality analysis, thereby requiring different algorithmic design strategies.

In many SNA scenarios it is beneficial to obtain nontrivial intermediate results earlier during the course of analysis, which is one of the capabilities of the anytime anywhere framework [2][1]. In many real-world networks such as online social networks, the underlying graph structure is highly dynamic. In such networks, vertices and edges are continuously evolving and analyzing these graphs in real-time has to be a continuous process. Thus, it is critical for the graph analysis algorithms to be able to accommodate the dynamic graph changes during the course of analysis. Some recent works addressed the issues caused by dynamic large-scale graphs from a load balancing perspective [6][8]. Our key focus in this work is to design algorithms that can efficiently handle dynamic structural changes, and refine the results instead of restarting or re-computing the entire analysis.

III. ANYTIME ANYWHERE METHODOLOGY

The anytime anywhere framework handles the analysis of large and dynamic real-world social networks by providing two key features: nontrivial intermediate results and handling network dynamism. The central idea in our framework is to divide the large-scale graph into smaller sub-graphs and analyze these sub-graphs by incrementally refining and combining their results. In general, many graph analysis algorithms are iterative in nature and exhibit anytime and anywhere properties [1][21][22]. Anytime algorithms have the following properties: 1) *Interruptibility*: The algorithm can be paused at any intermediate step, and a valid result can be obtained, 2) *Preemptability*: The algorithm can be suspended and restarted with minimal overhead, 3) *Result Quality*: A measure for the quality of the intermediate results generated by the algorithm can be defined, and 4) *Predictability*: The quality of the results has a monotonically non-decreasing relation with the amount of computational time and resources available to the algorithm. Moreover, performance profiles of an anytime algorithm can be formulated that allow for the prediction of the quality of future results.

On the other hand, algorithms with anywhere property have the ability to incorporate changes occurring in any part of the network, and efficiently propagate its effects across the whole network. Social network analysis performed using our methodology leverages these innate properties in graph algorithms such as the Floyd-Warshall's algorithm for calculating all pairs shortest paths (APSPs). Our anytime anywhere methodology consists of three phases: domain decomposition (DD), initial approximation (IA) and recombination (RC) and has been discussed in [1][2]. In the DD phase, the large social network graph is partitioned into smaller sub-graphs, which are then distributed among a set of compute nodes such that the workload is evenly distributed. Initial results computed during IA phase is incrementally combined, and refined in the RC Phase. Furthermore, RC phase incorporates any dynamic changes to the network. Detailed explanations for each of these phases are provided below.

A. Domain decomposition

Many social networks have community structures, such that nodes within a community are more tightly connected (i.e. share more edges), than nodes between different communities. Therefore, partitioning the graphs along these community structures helps to ensure that communication and computation costs are distributed equally across the processors. Computational costs for processing each sub-graph depends on the number of vertices in the sub-graph. Whereas, communication cost is determined by cut-edges and cut-size. Cut-edges are edges, whose endpoints belong to different sub-graphs, and cut-size is the number of such edges. Intuitively, sub-graphs with higher cut-size will share more edges with other sub-graphs, thereby increasing the amount of communications performed during each iterative step. On the other hand, sub-graphs with smaller number of cut-edges will provide better anytime results, since most information required by the sub-graph are self-contained.

To address these issues, we used a partitioning methodology based on the Parallel Graph Partitioning and Fill-reducing Matrix Ordering (ParMETIS) algorithms [23]. ParMETIS partitions the graph such that the resulting sub-graphs have minimal cut-edges. Naturally, as the number of cut-edges becomes smaller the amount of communications performed in the RC phase reduces significantly.

B. Initial approximation

The IA phase provides the preliminary approximation results for the whole network. The quality of results obtained in this phase is dependent on the computation time applied and the amount of information contained in the sub-graphs. Algorithms employed in the IA phase are designed such that the partial results calculated from each sub-graph can be combined and refined further in the RC phase.

C. Recombination

In the RC phase, the partial results are incrementally combined and refined to obtain a final result. It is iterative, with each iteration consisting of two key steps: 1) Refine the results obtained from the previous RC step, and 2) Adopt the dynamic changes in the graph. During each iteration, partial results in each sub-graph is refined based on the updates obtained from the neighboring sub-graphs. Furthermore, as discussed in earlier sections, the vertices and edges of dynamic social networks change over time. The RC phase continuously incorporates these changes into the network and

propagates the effects to the whole network. Communication costs are a major component of the overall run time of parallel/distributed algorithms. Therefore, a careful design of communication schedules based on the analysis requirements and on the computational platform is necessary. The anytime anywhere framework supports rigorous performance analysis using performance models such as LogP[24], which in turn, provides insights for choosing the appropriate communication schedule. Note that the RC phase does not impose a fixed communication model or schedule. Moreover, the framework is also platform independent.

IV. ANYTIME ANYWHERE ALGORITHM DESIGN FOR CLOSENESS CENTRALITY

Closeness centrality[25] is a key social network analysis (SNA) metric for identifying important actors/nodes. Such nodes figure prominently in the social process being analyzed as they play an important role in spreading social interactions and information. Closeness centrality analysis requires the computation of all pairs shortest paths (APSPs) in the network. Due to the computational challenges of calculating APSPs of large and dynamic networks, closeness centrality is a suitable problem to demonstrate the strengths of the anytime anywhere framework. Various heuristic based algorithm designs have been proposed for finding approximate centrality values [26], [27]. However, these designs require a "restart" when the graph changes and do not efficiently reuse already calculated partial solutions. Our anytime anywhere framework [1] was formulated to deal with large and dynamic social networks and we will provide algorithm designs based on this framework that efficiently deal with graph changes.

A. Closeness centrality

Definition 1: Given a graph $G(V, E)$ with a set of vertices V and a set of edges E such that $|V| = n$ and $|E| = m$, closeness centrality C_c of a vertex $u \in V$ is defined as,

$$C_c(u) = \frac{1}{\sum_{k=0}^{n} d(u, v_k)}$$

where $d(u, v)$ represents the shortest path distance between the vertices u and v.

In [1], we presented algorithms for handling dynamic increases and decreases in edge weight. We now extend the work to efficient algorithm design for dynamic edge deletions. Unlike changes to edge weight, edge deletions cause substantial changes to the structure of the network. A deleted edge can completely disconnect a sub-graph from the rest of the network. Our focus is on algorithm designs for efficiently reusing the partial solutions and reduce re-computations during edge deletions. The other challenge for parallel/distributed algorithm design is that, dynamic changes can cause load imbalances. However, the question of dynamic load balancing will be dealt in future work.

Below, we provide details on the algorithm design for closeness centrality, followed by discussions on the designs for handling dynamic changes. As we described earlier, algorithms designed using our anytime anywhere approach have three phases: 1) Domain decomposition (DD), 2) Initial approximation (IA), and 3) Recombination (RC).

B. Domain decomposition

We used a partitioning algorithm based on the Parallel Graph Partitioning and Fill-reducing Matrix Ordering (ParMETIS) algorithm [23]. The partitioning algorithm provides roughly $\frac{n}{p}$ vertices in each partition [28], where n and p are the number of vertices and number of processors respectively. Following the methodology in ParMETIS algorithm, the graph partitioning in DD consists of the following steps: coarsening, folding, initial partitioning, un-coarsening and un-folding. One of the limitations of ParMETIS is that it requires perfect square number of processors. Since the anytime anywhere framework does not impose such restrictions on processor count, we overcome this limitation in the graph partitioning by idling a subset of the processors during the DD phase.

Coarsening is the first phase of ParMETIS algorithm. During coarsening, multiple vertices are merged together into super nodes. Heavy edge matching, random matching, light edge matching, etc. are some of the strategies used for merging vertices. For balanced partitioning in our implementation, we ensure that during initial partitioning, no partition gets more than $O\left(\frac{n}{p}\right)$ vertices, where n is the number of vertices in the graph. In the ideal case, the graph is coarsened to p partitions, where p is the number of processors. However, achieving this level of coarsening in small world/scale free networks is challenging and time consuming. Therefore, we set a threshold of 20% as the required reduction in number of vertices. Since initial partitioning utilizes a randomized serial bisection algorithm, it was performed using all processors, and the best partition based on the edge cut-size is selected.

C. Initial approximation

In the initial approximation phase, the all pairs shortest paths are calculated for the sub-graphs assigned to each processor. Any all pairs shortest paths algorithm such as Dijkstra's or Floyd-Warshall's algorithm, that exhibits the anytime and anywhere properties can be used here. Since shortest paths for different sources can be computed in parallel, we used a multithreaded version of Dijkstra's single source shortest path algorithm in our implementation. We used a binary search tree for the graph traversal aspect of the algorithm.

D. Recombination

Recombination algorithm for closeness centrality uses the partial results calculated during initial approximation phase and further refines them on every iteration. In the recombination phase, updates from multiple sub-graphs are combined in an iterative process to generate better result. We used Distance Vector Routing (DVR) algorithm [29], a well-known distributed algorithm, to process incremental updates.

In DVR, each processor notifies its neighboring processors periodically (in this case during every RC step) about changes in its sub-graph, by sending its updated distance vectors. Distance vector (DV) of node v stores the currently known shortest path distances from node v to all other nodes in graph G.

During each iteration, partial results from each sub-graph are propagated to its neighboring processors through boundary vertices. Recombination phase makes use of the fact that a boundary vertex acts as a gateway to its corresponding sub-graph. Therefore, there is a need to only communicate the updates for the boundary vertices. This significantly reduces the amount of data communicated during each iteration.

```
1.  INPUT:P = {P_1, ... P_i, ... P_p} //set of processors assigned
                to the problem
2.  INPUT: n    //number of vertices in the overall graph G
3.  INPUT: G_i(V_i, E_i)  //sub-graph assigned to processor
                P_i
4.  INPUT: DV_i^0  //Distance Vectors (|V_i| X n) for sub-
            graph G_i generated in IA phase
5.  FOR EACH processor P_i do in parallel
6.      k = 0    //initialize the recombination step index
7.      DO //propagate updates to neighboring processors
8.          k = k + 1 // increment the recombination step
            index
9.          FOR j = 1 to p
10.             IF i ≠ j
11.                 RECV DVs of external boundary
                    nodes from processor P_j in
                    messages of size α.
12.                 Update local boundary vertices
                    using the DVs of external boundary
                    vertices.
13.             ELSE
14.                 SEND DVs of respective external
                    boundary vertices to (p − 1)
                    processors in messages of size α.
15.         END FOR
16.         Perform dynamic changes
17.         Update the local DVs to generate DV_i^k
18.     UNTIL k = p − 1 OR no more updates in any
        processor
19. END FOR
```

Figure 1. Pseudo-code 1: Recombination algorithm for closeness centrality

In Figure 1, lines 9 – 15 represent the personalized all-to-all communication that happens during the recombination phase. We use a communication schedule that avoids network flooding by ensuring that only one message traverses the network at any given time. In this schedule, only one processor sends at any one time, and each sender sends its updates to neighboring processors sequentially before the next sender starts communicating. Other forms of all-to-all communication scheduling (such as, linear wrap around, pair-wise exchange, etc.[30]) exist in which multiple processors communicate simultaneously. However, in practice, these schedules can lead to network congestion, and result in unpredictable performance. This is especially true for the Ethernet network[30] fabric, which was used in our experimental setup, described in later sections. Even though our all-to-all scheduling takes $O(p^2)$ steps (where p is the number of processors) in the worst case, it mitigates excessive network flooding. In each communication step, the boundary vertices of every sub-graph receive the updated values, which are then used to update the shortest paths by utilizing the Floyd-Warshall's algorithm.

Dynamic changes to the network can be incorporated at any step during the recombination phase. However, in order to increase the accuracy of the data that is being propagated to other processors; dynamic changes (Figure 1 *line 16*) to the graph are applied before local computations are initiated. This ensures that the data being shared between processors is up to date with the changes in the local sub-graph. In a static graph, maximum number of iterations/steps in the recombination (RC) phase is bounded by the number of sub-graphs, which in this case is equivalent to the number of processors, p. On the other hand, updates can occur during any iteration in a dynamically changing graph. Therefore, refinement of the results in the RC phase continues until there are no more updates to be shared between processors.

E. Anytime anywhere approach for edge deletion

```
1.  INPUT: P = {P_1, ... P_i, ... P_p}  //set of processors assigned to
    the problem
2.  INPUT: n    // number of vertices in the overall graph G
3.  INPUT: G_i(V_i, E_i)  //sub-graph assigned to processor P_i
4.  INPUT: ADJ_i  //Adjacency list of the vertices V_i of the sub-
    graph G_i; the list also contains the boundary nodes in all other
    sub-graphs which are connected to a vertex in G_i
5.  INPUT: DV_i  //Distance Vectors (|V_i| X n) for sub-graph G_i
6.  INPUT: E = {(a_1, b_1), ... (a_j, b_j), ... (a_l, b_l)}  // set of edges to
    be deleted
7.  INPUT: W' = {w'_1, ..., w'_j, ..., w'_l}  //weights of the edges to be
    deleted
8.  INPUT: P' = {P'_1, ..., P'_j, ..., P'_l}  //processor containing the
    vertex a_j of the edge to be deleted
9.  INPUT: P'' = {P''_1, ..., P''_j, ..., P''_l}  //processor containing the
    vertex b_j of the edge to be deleted
10. FOR EACH processor P_i do in parallel
11.   FOR EACH edge (a_j, b_j) to be deleted
12.     IF b_j is a node in sub-graph G_i
13.       SEND row DV_i[b_j] to all other processors //using tree
          broadcast
14.     ELSE
15.       RECV row DV_i[b_j] from processor P''_j
16.     END IF
17.     IF w'_j == DV_i[a_j][b_j]
18.       FOR EACH u ∈ V_i
19.         FOR EACH v ∈ V
20.           IF DV_i[u][v] ! = ∞ AND
                DV_i[u][v] == DV_i[u][a] + w'_j + DV_i[b][v]
```

```
21.                 DV_i[u][v] = ∞
22.                 Enqueue edge (u, v) into Queue Q
23.               END IF
24.             END FOR
25.           END FOR
26.         END IF
27.         IF a_j ∈ V_i//update the adjacency list after deleting edge
28.           DELETE edge (a_j, b_j) from ADJ_i[a_j] and ADJ_i[b_j]
29.           IF b_j ∉ V_i and ADJ_i[b_j] is empty // for cut-edge
    deletion
30.             Notify P_j'' to stop sending DV of b_j to P_j'
31.           END IF
32.         END IF
33.       END FOR
34.       WHILE Q is not empty
35.         Dequeue (u, v) from Q
36.         FOR EACH neighbor u' of u in sub-graph G_i
37.           IF DV_i[u][v] > DV_i[u][u'] + DV_i[u'][v]
38.             DV_i[u][v] = DV_i[u][u'] + DV_i[u'][v]
39.             mark DV_i[u][v]as updated  //for local computations
    and propagation through RC
40.           END IF
41.         END FOR
42.       END WHILE
43. END FOR
```

Figure 2. Pseudo-code 2: Anytime anywhere approach for edge deletion

In a social network, continuing an ongoing analysis after an edge deletion is a challenging task. However, as described in earlier sections, algorithms such as all pairs shortest paths have anytime characteristics, where removal of an edge may not alter every shortest path. Therefore, the shortest paths that are not affected by the deleted edge need not be re-computed. Moreover, the shortest paths that are altered can be efficiently recomputed using the sub-paths information, which were previously generated and stored. This is the core idea behind our algorithm for efficiently dealing with edge deletions in dynamic social networks.

The delete edge algorithm, described in this paper, makes use of the fact that the shortest path values affected by the deletion of the edge $e(a, b)$ are those, which contain the vertices a and b in their paths. When an edge is deleted, each processor P_i checks the distance vectors (DVs) of its local and boundary vertices to see if any of the paths went through that edge. This can be done by checking if, $DV_i[u][v] == DV_i[u][a] + w(a, b) + DV_i[b][v]$ where $u \in V_i$ and $v \in V$.

In order to perform this check, the processor that owns vertex b broadcasts the distance vector of b ($DV_i[b]$) to all other processors (Figure 2 *line 11 – 13*). Every path that satisfies this equality condition is reset to infinity. Once the paths, that contain the deleted edge, are reset, they have to be recalculated. To do this efficiently, we use the fact that if there exists another path between vertices u, v, then it has to go through a neighboring vertex (u') of u. Hence we only need to check the paths going through the neighboring vertices of u. Note that this could be an overestimation and there may be a better path between vertices u and v. A shorter path will then be discovered during successive recombination steps. After the deletion of an edge, if there is no path between u and v that goes through u' it is either because there exists no path between u and v or the new path has not been discovered yet. The latter case will also be handled during successive iterations of the recombination phase. When a new shortest path is discovered, it is communicated to the neighboring processors, and their boundary vertices and local DVs are updated.

V. Algorithm Analysis (complexity/run time analysis)

In this section, we analyze the delete edge algorithm described in the previous section, along with the time complexity of the initial approximation (IA) and recombination (RC) phases. We do not include the analysis of the domain decomposition (DD) phase as graph partition is only done once, and dynamism in the graph does not trigger repartitions in our implementation. Given a graph $G(V, E)$ with a set of vertices V and a set of edges E such that $|V| = n$ and $|E| = m$. Let $P = \{P_1, P_2, \dots, P_p\}$ be the set of compute nodes or processors available for processing the graph. A processor P_i is assigned a vertex set $V_i \subseteq V$. During the DD phase, the vertex set V is partitioned into subsets of vertices $\{V_i\}_{i=1}^{p}$ such that $\bigcup_{i=1}^{p} V_i = V$ and $\bigcap_{i=1}^{p} V_i = \phi$. $G_i(V_i, E_i)$ denotes the sub-graph of G induced from the vertex set V_i and assigned to processor P_i during the DD phase. Given an edge $e(u, v) \in E$, where $u \in V_i$ and $v \in V_j$, if $V_i \neq V_j$ then $e(u, v)$ is referred to as a *cut-edge*, and u and v are referred to as a boundary node of V_i and V_j, respectively. Also $C_i \subseteq V_i$ denotes the set of boundary nodes in V_i. Moreover, given a vertex $v \in C_i$, $cut\_deg(v)$ is defined to be the number of cut-edges, where v is one of the vertices in the edges. Furthermore, we define $\gamma_i = \max_{v \in C_i} cut\_deg(v)$.

We leveraged the LogP distributed memory model[24] to analyze the asymptotic run time (provided below) of the various phases of our anytime anywhere algorithm. The LogP model utilizes the following four parameters: 1) Latency (L): the delay incurred by the message to reach the target processor, 2) Overlap (o): the amount of time spent exclusively by the source processor on sending a message or the time spent exclusively by the target processor on receiving a message, 3) Gap (g): the minimum time interval between two consecutive send operations at the source processor or two consecutive receive operations at the target processor, and 4) Processors (p): the number of processors utilized.

A. Initial Approximation

During the initial approximation (IA) phase, each processor creates partial results from the information available within its local sub-graph. These partial values can be combined later with the information from other sub-graphs

during recombination. Therefore, it is essential to use or design algorithms for IA such that they exhibit the anytime and anywhere properties. We used Dijkstra's algorithm to calculate all pairs shortest paths within the local sub-graph during initial approximation. The runtime for Dijkstra's algorithm[31] on processor P_i using a binary search tree is $O(|V_i||E_i|\log|V_i|)$. Assuming that in the worst case, the processors are assigned fully connected sub-graphs, the runtime of IA is $O\left(\frac{n^3}{p^3}\log\frac{n}{p}\right)$. However, real world social networks tend to exhibit the scale free property with relatively few high degree nodes. Therefore the sub-graphs generated during domain decomposition are generally not fully connected graphs. If support for multithreads is available on the processors, the shortest paths for different sources in the Dijkstra's algorithm can be calculated in parallel. The run time of IA with multithread support is $O\left(\frac{n^3}{\tau p^3}\log\frac{n}{p}\right)$ where τ is the number of threads used in each processor.

B. Recombination

The worst case analysis of the recombination phase is provided here. During the recombination (RC) phase (refer pseudo-code in Figure 1), the size of data that is communicated by a processor depends on the number of boundary vertices of the sub-graph assigned to it. In the worst case, processor P_i has to send $n|C_i|$ data items to all the other processors. Note that the data is communicated in messages of size α in order to mitigate network flooding and to also constrain memory requirements. Time taken for updating the distance vector (DV) of boundary vertices (see Figure 1, *line 12*) in processor P_i is $O(n\gamma_i|C_i|)$. In real world social networks with the scale free property, we make the following assumption on the maximum number of cut-edges that a boundary vertex assigned to processor P_i [32], [33][34] can have: $\gamma_i \leq \frac{n}{\log n}$. The time taken for updating the DV of the sub-graphs using the updated information from boundary vertices (see Figure 1, *line 17*) is $O\left(\frac{n^3}{p}\right)$.

The run time to complete the RC phase is:

$$O(\frac{n^3}{p}+\frac{n^3}{\log n}+\frac{n^2}{\alpha}p^2L+n^2p^2g)$$

C. Edge Deletion using anywhere approach

In this section we analyze the performance of the delete edge anytime anywhere approach (refer pseudo-code in Figure 2). Let l be the number of edges to be deleted.

- Figure 2 *Line 11 – 16*: This communication utilizes a tree broadcast approach. Hence the time complexity for this operation is $O\big(l\log p\,(L+ng)\big)$.
- Figure 2 *Line 17 – 26*: The time complexity for equality comparisons by a processor P_i , to check if the edge is present in any of the paths, is $O(l|V_i|n)$.
- Figure 2 *Line 34 – 42*: In the worst case all the paths in G_i are affected, hence the time taken to recalculate new paths using its neighbors is $O(n|V_i|^2)$.

Total runtime of anytime anywhere delete edge algorithm in the worst case is $O\left(\frac{n^3}{P^2}+l\frac{n^2}{P}+l\log p\,(L+ng)\right)$.

It may be noted that the analysis provided above takes into account the additional computations required to incorporate changes in the DVs of the sub-graphs so that the processors can continue with the RC phase. An additional $(p-1)$ steps may be needed once an edge is deleted regardless of how many RC steps have already been performed.

VI. Implementation and Experimental Results

We implemented and tested our approach on a distributed-memory cluster with 32 compute nodes connected over 1 Gb/s Ethernet network. Each compute node had dual Intel Xeon E5 (1.8 GHz) processors (total of 16 cores per node) and 32 GB of memory. The algorithms were implemented in C++. MPICH's Message Passing Interface (MPI) library was used for inter-node communications, and the multithreaded aspect of the IA phase was implemented using OpenMP.

A. Experimental setup

As real world social network graphs tend to exhibit the scale free property, we generated a test bed of medium sized social networks using the Pajek tool, with the node degree of the networks obeying the power law property. Keeping in mind the prevalence of various online social networking applications such as Twitter and YouTube, where the underlying social networks contain directed edges, our experimental study focused on the closeness centrality analysis of directed graphs. However our anytime anywhere approach is applicable to both directed and undirected graphs, and our current implementation can also handle undirected graphs. As this is preliminary work, the experimental results presented in this paper, focus on understanding the performance of our approach for handling deletions of regular edges in medium sized graphs. In future work, we plan to provide a more comprehensive experimental study that includes performance results with larger networks, and with deletions of both regular edges and cut-edges.

Although, there are existing work on analysis of dynamic networks, their focus has been on designing efficient data structures to handle graph changes or studying the effects of load balancing. To the best of our knowledge the anytime anywhere framework described in this paper, is the only work that focuses on the parallel algorithm design aspects of the dynamic social network problems. Due to this reason, we compared our anytime anywhere approach with the baseline restart approach, where the network analysis has to be restarted from scratch when an edge is deleted. The baseline algorithm also has the domain decomposition (DD), initial approximation (IA), and recombination (RC) phases. The crucial difference is that the baseline algorithm has to restart

the analysis, i.e., redo both the IA and RC stages when the network changes.

B. Results

In this section we report the performance results of our edge deletion algorithm. According to the performance analysis in Section V, the stage in the analysis at which the change (i.e. one or more edge deletions) happens, size of change and the rate at which the change happens has significant impact on the performance of our anytime anywhere algorithm. Based on these factors, we expect the baseline approach to perform better when edge deletions happen at an initial stage of the analysis. However, when the change occurs at a later stage, the anytime anywhere approach should benefit from the precomputed partial results. In addition, the size of the change affects the number of shortest paths that have to be recomputed. In order to understand the performance variations caused by these factors, we designed two experiments. In the first experiment, we keep the rate of change low by selecting a specific recombination step and introducing edge deletions during that step. In the second experiment, we introduce edge deletions at multiple points during the graph processing. Therefore the dynamism rate is much higher in the second experiment.

Figure 3 shows the results for the first experiment which was performed on a graph with 50,000 vertices using 16 processors. In separate experimental runs, edge deletions were introduced during the initial (RC0) and later (RC09) steps of the RC phase. Note that the edges deleted were equally divided among the processors. As mentioned in Section V.C, our anytime anywhere methodology for edge deletion has a worst case running time of $O\left(\frac{n^3}{p^2} + l\frac{n^2}{p} + l(L+ng)\log p\right)$. The dominant terms in the run time are $(l\frac{n^2}{p})$ and $(\frac{n^3}{p^2})$, which correspond to the time complexity for determining the affected shortest paths and recalculating those paths, respectively. When edge deletions occur during the initial steps of recombination (e.g. RC0), the number of informative partial results available is low. Hence the time taken to calculate the shortest paths by the baseline and the anytime anywhere method is similar. However, the anywhere approach has an additional cost $(l\frac{n^2}{p})$ of determining the paths that are affected by the deleted edges, which is dependent on the number of deletions, l occurring at a particular RC step. As a result, for smaller number of edge deletions both methods perform similar whereas, as the number of deletions increases, the anytime anywhere method takes longer due to the additional computational and/or communication overhead.

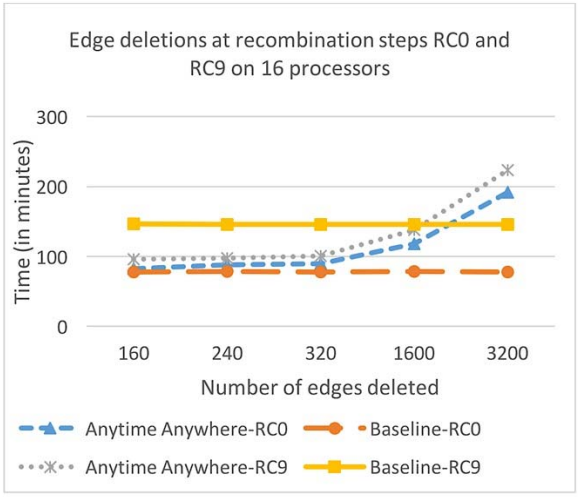

Figure 3. Performance comparison of anytime anywhere vs. baseline edge deletions at recombination (RC) step RC0 and RC9 using 16 processors on graph with 50,000 vertices

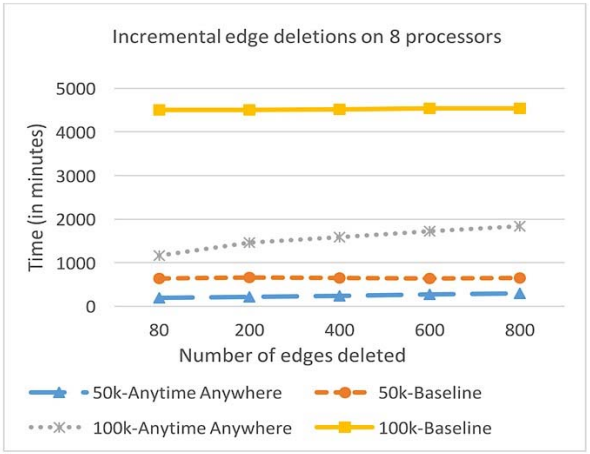

Figure 4. Performance comparison of anytime anywhere vs. baseline incremental edge deletions using 8 processors on graphs with 50,000 and 100,000 vertices

For changes happening during later stages of the RC phase, the time for re-computation is less when compared to the baseline due to the utilization of partial results. Therefore, our anytime anywhere method performs better than the baseline for smaller number of edge deletions. However, as the number of changes increases, the time taken to identify and recalculate the affected shortest paths also increases, thereby affecting the performance of our anytime anywhere approach. Note that this experiment requires the baseline to restart only once as all the edge deletions occur in the same RC step. The performance of our approach should be significantly better when edge deletions occur in multiple steps of the RC, as the baseline now has to be restart multiple times.

Our next set of experiments were designed to understand the effect of multiple changes during the course of analysis. In these scenarios, we expect the anytime anywhere approach

to perform better since baseline method does not utilize partial results, and has to restart from scratch for every change. To study the performance of our anytime anywhere approach under these conditions, we introduce edge deletions across multiple iterations. Given λ, the total number of edge deletions, that are introduced in the network during k RC steps or iterations, i.e. $l = \frac{\lambda}{k}$. For the baseline, this would require restarting the analysis k times, each time with the modified network. This would require a significant amount of time and resources. To simplify the experimental analysis, we ran the baseline after deleting the specified number of edges (λ) from the network and measured the runtime for each iteration separately. This ensures a computational advantage for the baseline algorithm as it essentially processes a smaller network (in terms of number of edges) than the anytime anywhere approach. This also helps in reducing the experimentation time, since essentially we need to run the baseline only once.

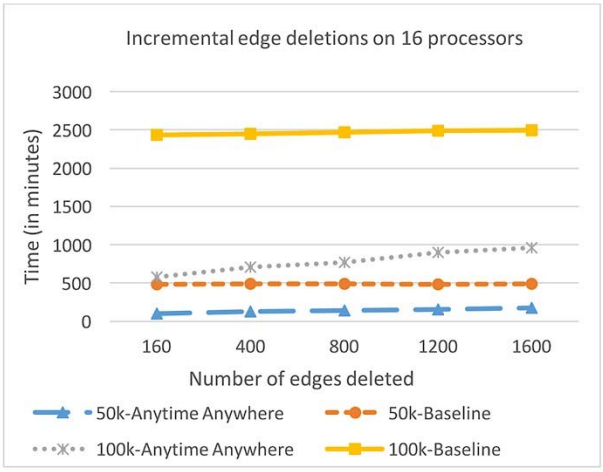

Figure 5. Performance comparison of anytime anywhere vs. baseline incremental edge deletions using 16 processors on graphs with 50,000 and 100,000 vertices

Figure 4 and Figure 5 shows the incremental edge deletions on 8 and 16 processors respectively. In the experimental run with 8 processors, the λ edge deletion operations are split equally across 8 iterations. On the other hand, in the run with 16 processors, edge deletions are split across 10 iterations. Here, the baseline method has to restart from scratch every time a change occurs, whereas anytime anywhere method utilizes the partial results without restarting the computations. When compared to the previous evaluations discussed in Figure 3, the number of edge deletions are distributed over the RC phase, which reduces the number of shortest paths that needs to be re-computed at any given iteration. A large number of partial results calculated thus far would still be useful; therefore, our anytime anywhere approach performs much better than the baseline in this case. Figure 4 shows the results for graphs with 50,000 and 100,000 vertices computed on 8 processors and Figure 5 shows the results for the same on 16 processors.

The experimental results agree with the theoretical analysis, and it is clear that the anytime anywhere approach performs much better at higher rates of network dynamism. Also, changes occurring, in the final steps of the RC refinement process have a more negative impact on the baseline performance. The results also show that when baseline does not have to restart multiple times, its performance is significantly better when l is large. It is clear from the results that our approach of using anytime anywhere design is highly effective in real world scenarios where the rate of change of networks is high.

VII. Conclusion

In this work, we designed an anytime anywhere methodology for calculating closeness centrality in large and dynamic social networks that allows for edge deletions. Using both theoretical analysis and experimental evaluations, we demonstrated the ability of our methodology to reuse partial results and to reduce the overheads caused by re-computations. We evaluated our approach to handle edge deletions on a test bed of scale free graphs under different dynamism rates. Performance of the edge delete algorithm was compared with a baseline method. While smaller or less dynamic networks did not readily benefit from our methodology, our method performed significantly better than the baseline under high dynamism rates and as network sizes increased. Our approach is platform independent, and is only constrained by the memory available on the underlying architectures. Overall, these results have further demonstrated the ability of our anytime anywhere framework to handle network dynamisms in social network analysis.

In the future, we plan to extend our anytime anywhere methodology to handle various other network dynamisms such as node additions and deletions. Approaches used in handling social network dynamisms can also be utilized to design algorithms that adapt to hardware failures. Furthermore, we are interested in applying our methodology to analyze and handle load imbalances, caused by node and edge deletions/additions in platforms such as cloud computing.

Acknowledgement

This work has been supported by NPSG-N00244-15-1-0046.

References

[1] E. E. Santos, L. Pan, D. Arendt, and M. Pittkin, “An Effective Anytime Anywhere Parallel Approach for Centrality Measurements in Social Network Analysis,” in *2006 IEEE International Conference on Systems, Man and Cybernetics*, 2006, vol. 6, pp. 4693–4698.

[2] L. Pan and E. E. Santos, “An anytime-anywhere approach for maximal clique enumeration in social network analysis,” in *the IEEE International Conference on Systems, Man and Cybernetics - SMC 2008*, 2008, pp. 3529–3535.

[3] D. Ediger, K. Jiang, E. J. Riedy, and D. A. Bader, "GraphCT: Multithreaded Algorithms for Massive Graph Analysis," *Parallel Distrib. Syst. IEEE Trans.*, vol. 24, no. 11, pp. 2220–2229, 2013.

[4] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: Mining peta-scale graphs," *Knowl. Inf. Syst.*, vol. 27, no. 2, pp. 303–325, 2011.

[5] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," *Proc. 2010 Int. Conf. Manag. Data - SIGMOD '10*, no. June, pp. 1123–1126, 2010.

[6] Z. Khayyat, K. Awara, A. Alonazi, and D. Williams, "Mizan : A System for Dynamic Load Balancing in Large-scale Graph Processing," *EuroSys*, pp. 169–182, 2013.

[7] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel : A System for Large-Scale Graph Processing," in *SIGMOD'10*, 2010, pp. 135–145.

[8] L. M. Vaquero and C. Martella, "Adaptive Partitioning of Large-Scale Dynamic Graphs," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2014, pp. 35:1–35:2.

[9] E. E. Santos, L. Pan, D. Arendt, H. Xia, and M. Pittkin, "An Anytime Anywhere Approach for Computing All Pairs Shortest Paths for Social Network Analysis," in *Integrated Design and Process Technology*, 2006.

[10] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and Algorithms for Graph Queries on Multithreaded Architectures," *2007 IEEE Int. Parallel Distrib. Process. Symp.*, 2007.

[11] B. Dezso, A. Jüttner, and P. Kovács, "LEMON - An open source C++ graph template library," *Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 5, pp. 23–45, 2011.

[12] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Sy, p. 1695, 2006.

[13] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, 2008, vol. 836, pp. 11–15.

[14] S. P. Borgatti, M. G. Everett, and L. C. Freeman, "Encyclopedia of Social Network Analysis and Mining," R. Alhajj and J. Rokne, Eds. New York, NY: Springer New York, 2014, pp. 2261–2267.

[15] S. Sakr, A. Liu, and A. G. Fayoumi, "The Family of MapReduce and Large-Scale Data Processing Systems," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 1–44, 2013.

[16] O. Batarfi, R. El Shawi, A. G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr, "Large scale graph processing systems: survey and an experimental evaluation," *Cluster Comput.*, vol. 18, no. 3, pp. 1189–1213, Jul. 2015.

[17] Y. Bu, V. Borkar, J. Jia, M. Carey, and T. Condie, "Pregelix: Big (ger) Graph Analytics on A Dataflow Engine," *Vldb*, pp. 161–172, 2015.

[18] R. R. McCune, T. Weninger, and G. Madey, "Thinking Like a Vertex," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 1–39, 2015.

[19] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *J. ACM*, vol. 51, no. 6, pp. 968–992, 2004.

[20] M.-J. Lee, S. Choi, and C.-W. Chung, "Efficient algorithms for updating betweenness centrality in fully dynamic graphs," *Inf. Sci. (Ny).*, vol. 326, pp. 278–296, Jan. 2016.

[21] S. Zilberstein, "Using Anytime Algorithms in Intelligent Systems," *AI Mag.*, vol. 17, no. 3, p. 73, 1996.

[22] J. Korah, E. E. Santos, and E. Santos, "Multi-agent framework for real-time processing of large and dynamic search spaces," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*, 2012, p. 755.

[23] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *J. Parallel Distrib. Comput.*, pp. 1–21, 1998.

[24] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, "LogP: a practical model of parallel computation," *Commun. ACM*, vol. 39, no. 11, pp. 78–85, 1996.

[25] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[26] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating Betweenness Centrality," *Technology*, vol. 4863, pp. 124–137, 2007.

[27] D. Eppstein and J. Wang, "Fast Approximation of Centrality," *J. Graph Algorithms Appl.*, vol. 8, no. 1, p. 2, 2000.

[28] G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, 1998.

[29] J. F. Kurose and K. W. Ross, *Computer Networking A Top-Down Approach Featuring the Internet*, vol. 1. 2005.

[30] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Tert. Educ. Manag.*, vol. 10, no. 2, pp. 127–143, 2004.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.

[32] M. E. J. Newman, "Ego-centered networks and the ripple effect," *Soc. Networks*, vol. 25, no. 1, pp. 83–95, 2003.

[33] M. E. J. Newman, "The structure and function of complex networks," *SIAM Rev.*, vol. 45, no. 2, pp. 167–256, 2003.

[34] R. Albert and A. L. Barabasi, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, no. 1, pp. 47–97, 2002.