Effectively Handling New Relationship Formations in Closeness Centrality Analysis of Social Networks using Anytime Anywhere Methodology

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

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

***Abstract*—The flood of real time social data, generated by various social media applications and sensors, is enabling researchers to gain critical insights into important social modeling and analysis problems such as the evolution of social relationships and analysis of emergent social processes. However, current computational tools have to address the grand challenge of analyzing large and dynamic social networks within strict time constraints before the available social data can be effectively utilized. The computational issues are further exacerbated by the network size, which can range in the millions of nodes, and by the need for analytical tools to work with various computational architectures. Existing methodologies primarily deal with dynamic relationships in social networks by simply re-computing the results, and relying on massive parallel and distributed processing resources to maintain time constraints. In previous work, we introduced an overarching parallel/distributed algorithm design framework called the anytime anywhere framework, which leverages the inherent iterative property of graph algorithms to generate partial results, whose quality increase with the processing time, and which efficiently incorporates network changes. In this paper, we focus on closeness centrality algorithm design for dynamic social networks where new relationships are formed due to edge additions. Using both theoretical analysis and empirical results, we will demonstrate how this algorithm efficiently reuses the partial results and reduces the need for re-computations.**

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

I. INTRODUCTION

Social networking applications in a number of areas including friendship networks, professional networks, and other special interest networks have witnessed explosive growth in the past decade. Analysis of the massive data sets generated from these applications can provide critical insights in various domains such as social science research, business analytics, and healthcare. However, calculating key social network metrics such as closeness and betweenness centrality for real-world social network graphs is challenging due to their massive sizes and network dynamism.

Computational cost and resources required to conduct social network analysis (SNA) grow rapidly with network size [1][2]. This hinders SNA in large-scale networks, especially in time-critical applications. In order to accelerate network analysis, various parallel/distributed frameworks, libraries and tools have been proposed [3][4][5][6]. Although these works address the challenges in handling large-scale networks, less attention has been given to algorithm designs that can efficiently handle network dynamism in large-scale graphs.

Many real-world graph data sets such as social networks, web-graphs, and sensor networks, are inherently dynamic with new vertices and edges being continuously added while existing ones being removed during analysis. To analyze such dynamic datasets, novel parallel/distributed SNA algorithm designs that can efficiently handle network dynamism and avoid massive recomputations are essential. Furthermore, with the availability of heterogeneous computing platforms such as shared-memory supercomputers, cluster computing, and cloud computing it is necessary to design platform independent algorithms.

In our previous work [1][2][7], we presented the anytime anywhere framework, an overarching parallel/distributed algorithm design framework to formulate algorithms for large and dynamic social network analysis. In this approach, large social network graph is partitioned into smaller sub-graphs and are incrementally analyzed and refined over time. The anytime property refers to the ability of the algorithm to provide non-trivial partial results during analysis, the quality of which improve with computation time. The anywhere property refers to the capability to continuously incorporate dynamic changes to the graph and systematically propagate the effects of these changes to the whole network. Moreover, the anytime-anywhere framework is not restricted to specific computational platforms. In previous work, we demonstrated our approach to handle dynamic changes such as edge weight changes [1][7] and edge deletions [8] during centrality analysis. In this work, we focus on handling new relationship formations due to edge additions in closeness centrality analysis.

Given that normal day-to-day online activities including activities such as connecting to a friend or acquaintance in social media, and even Bluetooth based device interconnectivity can be computationally represented as relationship or edge formation in a social network, formation of new relationships and analyzing their changes are challenging and important problems in dynamic social networks. In a number of social network analyses, edge additions could also bridge different parts of the network and create surprising new relationships. This enhanced network connectivity can lead to new social dynamics especially in complex real world problems such as disease spread modeling, modeling spread of social influence and modeling political stability, where time sensitive social information is used. Therefore, it is critical to design efficient SNA algorithms that can incorporate edge additions and systematically propagate its effects for dynamic network analysis.

II. BACKGROUND

Recently there has been a lot of interest on formulating new tools [9][10], libraries [11][6], computational models [4][3] and algorithms [12][13][14] for large-scale graph analysis. MapReduce [3] and Pregel [4] are two prominent frameworks that have been proposed for analyzing large data sets. In particular, Pregel uses a vertex centric model and provides a scalable platform for implementing common graph algorithms. It leverages the Bulk Synchronous Parallel (BSP) programming model, in which computation is performed in sequences of supersteps separated by barriers for communication and synchronization. Giraph[1] and Hama[2] are two popular Pregel based implementations. These frameworks provide data structures, APIs, and platforms for graph algorithms. However, one critical aspect that is largely overlooked in these works is designing efficient algorithms for handling dynamic graphs.

Designing efficient graph algorithms is important especially when solving large-scale graphs, since the computational and memory cost increases rapidly with the network size. Many works have proposed algorithms for various social network analysis (SNA) metrics. Computing SNA metrics such as closeness and betweenness centrality is a challenging problem since both requires computation of all pairs shortest paths (APSPs). One of the earliest works for efficiently computing betweenness centrality was proposed by Brandes [15], with the computational complexity of $O(EV)$ for unweighted graphs and $O(EV + V^2 \log V)$ for weighted graphs, where V is the number of vertices and E is the number of edges. Green *et al*. [12] proposed an approach based on the Brandes algorithm [15] to handle insertion of edges during betweenness centrality computation. This algorithm uses an additional BFS tree data structure for each vertex in the graph to store previously computed values. However, this approach is serial and has a space complexity of $O\big(V(V + E)\big)$. QUBE [14] is another serial algorithm that supports edge addition and removal during betweenness centrality computation. In this approach, the graph is decomposed into several disjoint sets of vertices known as minimum union cycles (MUCs). The concept of MUCs is used to identify the set of vertices whose centrality values are not affected by the dynamic updates. Consequently, during dynamic graph updates, the algorithm only recomputes the centrality values of the affected MUCs. Sariyuce *et al*. [16] proposed an incremental serial algorithm for computing closeness centrality in unweighted graphs. This algorithm leverages a filtering approach to detect vertices that are not affected by edge addition/deletion to prevent unnecessary graph updates and reduce recomputations. These works only provide methodologies to handle dynamic graph changes in a serial processing environment, and they are also constrained by the amount of resources (processor, memory, etc.) available on the machine. Although new parallel versions of these methods could be designed to handle larger graphs, their weaknesses lie in that many of their current mechanisms for handling dynamic graph changes are difficult to be mapped efficiently onto a parallel/distributed computing environment.

Kourtellis *et al*. [17] proposed a parallel betweenness centrality framework for dynamic unweighted graphs. This framework uses an additional data structure (to represent a directed acyclic graph of shortest paths) for each vertex and leverages Brandes algorithm to compute the betweenness centrality offline before processing any dynamic updates. The data structure is updated in the preprocessing step and this information is used to support incremental computation during graph updates. This method requires computing betweenness centrality for the entire graph in the preprocessing step. However, for large-scale graphs, this preprocessing step could become a bottleneck especially in time critical applications.

Our anytime anywhere approach provides a parallel/distributed framework that can handle both weighted and unweighted graphs. The main focus of our approach is to design efficient algorithms to handle dynamic large-scale graphs. Our framework is platform independent and can be applied to different computational platforms such as distributed clusters and massively multithreaded architectures. Since, our framework does not impose any restrictions on the data structures used, computational models or communication schedules; it can leverage other tools, libraries and programming models. In our previous work, we developed algorithms to perform centrality analysis [1] [8] and finding maximal cliques [2]. We have focused on designing algorithms for a variety of dynamic graph issues. These include handling edge weight changes and edge deletions during centrality analysis, and edge additions/deletions during maximal clique enumeration. In this paper, we have developed an algorithm to handle edge additions during closeness centrality computation. Edge additions during centrality measurements can significantly affect the intermediate results and in some cases could improve the centrality values of a large number of vertices in the network. Incorporating such changes especially on a parallel/distributed platform requires careful algorithm design to systematically propagate the effects of these changes across the network.

III. ANYTIME ANYWHERE ARCHITECTURE

Generally, when a large-scale graph is analyzed on a parallel/distributed platform, typically, the first step is to partition the input graph and assign the sub-graphs to different processors. After partitioning, each processor performs relevant analysis of its individual sub-graph. Results calculated from these individual analyses are combined and refined in one or more steps to obtain the final solution. In addition, when analyzing dynamic graphs, graph changes have to be incorporated and existing results have to be recomputed. Furthermore, dynamic changes occurring at certain parts of the graph may not affect the entire graph and may be leveraged to perform dynamic updates efficiently.

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

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

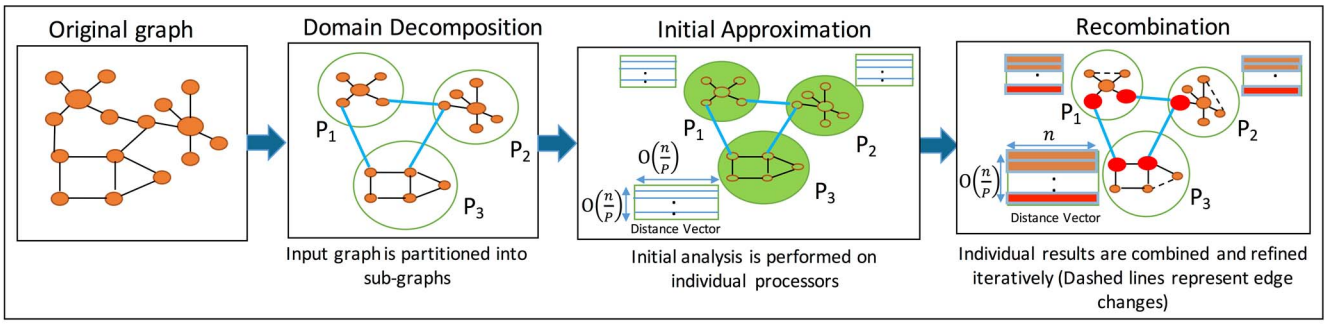

Figure 1. Anytime Anywhere Framework

The anytime anywhere methodology provides a generic framework to design parallel/distributed SNA algorithms that can handle large-scale dynamic networks. In this approach, the input graph is decomposed into sub-graphs of manageable size that are incrementally analyzed/refined over time. The quality of the partial results generated during these intermediate steps continuously improve with time. Furthermore, these partial results are utilized to provide non-trivial, meaningful intermediate results and to aid in the efficient incorporation of dynamic graph changes. The anytime anywhere framework provides an overarching methodology to design algorithms with these characteristics and consists of three phases: domain decomposition (DD), initial approximation (IA), and recombination (RC). We note that some of the details provided below have been discussed in previous work [8].

A. Domain Decomposition

In the DD phase, the workload is distributed across processors by decomposing the input graph into balanced and manageable sub-graphs. This is a crucial phase in the framework since the quality of the partitions significantly affects the load balancing and quality of the results obtained in the subsequent phases. Most social networks have an inherent community structures; in essence, these are structures where vertices that are densely interconnected as a group and fewer connections between such groups)[18]. Partitioning the input graph along community lines may help to improve the quality of the results during the later stages of processing. In addition, since nodes within a community are tightly connected than nodes across different communities, partitioning the graph based on the community structure reduces the amount of information exchanged between processors and thereby reduces communication costs. We used a parallel, multilevel graph partitioning method, based on the Parallel Graph Partitioning and Fill-reducing Matrix Ordering (ParMETIS) algorithm [19] to decompose the input graph to the required size based on the community structures. Figure 1 depicts the domain decomposition phase for a sample input graph, where the input graph is decomposed into three sub-graphs and assigned to three processors (P_1, P_2, P_3).

B. Initial Approximation

The sub-graphs obtained from the DD phase are analyzed individually in the IA phase. The IA phase provides preliminary and approximate results for the SNA metric of interest. For instance, when computing closeness centrality, the APSP values are calculated for the graph partitions obtained from the domain decomposition phase. For a graph with n vertices distributed across p processors, each processor computes the preliminary shortest paths between its $O\left(\frac{n}{P}\right)$ local nodes (Figure 1). These results provide an initial analysis of the network structure and also provide clues about important actors in the network. Algorithms used in the IA phase should also support the anytime anywhere properties so that the partial results generated in the IA phase can be incrementally combined and refined in later stages.

C. Recombination

In the recombination phase, each processor refines its partial values based on the solutions obtained from the neighboring processors. For example, during closeness centrality computation, each processor computes preliminary shortest path values from local nodes to every other node in the overall graph (Figure 1). This computation/communication steps are repeated until the final results are obtained. The communication schedule used for exchanging the partial results across the processors plays a critical role in improving the run-time and preventing issues such as network congestion. Since most real-world social networks are continuously evolving, it is crucial for the SNA algorithm to have the ability to incorporate dynamic changes. In the RC phase, graph updates are continuously incorporated into the sub-graphs and the effects of these changes are incrementally propagated to the whole network.

IV. ANYTIME ANYWHERE ALGORITHM DESIGN FOR CLOSENESS CENTRALITY

Centrality measures help to understand and analyze actor roles in social networks. In particular, closeness centrality value of a node measures its connectivity to the rest of the actors in the network based on geodesic distances. Calculating closeness centrality requires computing APSPs, and one of the popular APSP algorithms is Floyd-Warshall, which has $O(n^3)$ time complexity. The computational challenge of calculating APSP in large and dynamic networks makes closeness centrality a suitable problem to demonstrate the effectiveness of the anytime anywhere framework.

Consider a graph $G = (V, E)$, where V is the set of vertices (actors) in the graph and E is the set of edges representing the relationships between the vertices ($|V| = n$, $|E| = m$). Let $d(u, v)$ represent the shortest path distance between actors u and v. Then closeness centrality C_c of an actor $u \in V$ can be defined as the reciprocal of the total distance from actor u to all the other actors in G:

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

In our previous work, we designed and demonstrated algorithms to handle edge weight changes [1][7] and edge deletions [8] during centrality computations. Here we have presented an algorithm to handle edge additions. The crucial difference between edge deletion and edge addition is that edge deletions can degrade the network connectivity and increase the shortest path distance, while edge additions can enhance network connectivity and decrease the shortest path distance. This is because, during edge deletions, some shortest paths could become invalid; therefore, there is an additional cost to recalculate these invalidated shortest paths. However, edge additions can only provide additional paths but will not invalidate existing paths. Some aspects of the closeness centrality algorithm presented in subsections A – C were also discussed in our previous work [8]. However, previous work in closeness centrality analysis did not consider new edges (edge additions).

A. Domain Decomposition

In this phase, the large social network graph is partitioned into balanced sub-graphs and each sub-graph is assigned to a different processor. Most social networks possess inherent community structures and decomposing the social network graph using partitioning methods such as hash based or random partitioning could fragment these structures [18]. Graph partitioning methods that seek to preserve community structures, can potentially lead to results with good quality in the subsequent phases and help to reduce communication costs during the RC phase. Cut-edges are an important factor in graph partitioning and can be used to determine the quality of the graph partitions. Cut-edges are edges that connect sub-graphs across different processors (edges highlighted in blue, Figure 1). Cut-size is the total number of cut-edges in the graph. Reducing the cut-size during partitioning helps in preserving the community structure and is one of the critical objectives of the DD phase.

Parallel Graph Partitioning and Fill-reducing Matrix Ordering (ParMETIS) [19] based algorithm is used to decompose the input graph based on the community structure and the number of processors. In this method, the input graph is coarsened systematically by combining vertices with stronger connections to obtain a relatively small graph. In an ideal case, we would like to coarsen the input graph to p nodes, p being the total number of processors. However, coarsening the input graph to such a level is time consuming, so we halt the coarsening process once we reach 20% of the original graph size. Moreover, we conducted pilot tests which indicated that coarsening beyond the 20% mark gave diminished returns in terms of algorithm performance. In the next step, the coarsened graph is partitioned into P partitions, such that each partition has $O(n/P)$ vertices, n being the total number of vertices in the input graph.

B. Initial Approximation

In the IA phase, each processor analyzes the sub-graph obtained from the DD phase locally. These results provide an initial approximation of the whole network. A relevant SNA algorithm that exhibit anytime anywhere properties can be applied to compute the initial approximation results. Similar to our previous work [8], we used a multithreaded implementation of Dijkstra's single source shortest path algorithm during the initial approximation in order to leverage multiple cores available in our processors.

C. Recombination

```
1.  INPUT:{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 2. Pseudo-code 1: Recombination algorithm for closeness centrality [8]

The results obtained from the IA phase are based on the information contained in the individual sub-graphs. However, in order to obtain the complete results, the sub-graphs should be further analyzed by combining and refining the partial results obtained from the neighboring processors. In addition, dynamic updates to the graphs are incorporated in the RC phase. Thus, the RC phase performs three tasks iteratively: receive updates from other processors, perform dynamic changes and refine results based on the updates received. We

have utilized Distance Vector Routing (DVR) algorithm [20] which is used to perform incremental graph updates on parallel/distributed systems. Each vertex v in the graph maintains a Distance Vector (DV) to store the current shortest path distances to other vertices. Using DVR algorithm each processor notifies its updates to other processors by sending the updated boundary DVs. Boundary DVs are the distance vectors of the boundary vertices (vertices shown in red, Figure 1). These boundary vertices act as bridges connecting the sub-graphs belonging to different processors. Hence, it is sufficient to communicate only the DVs of these boundary vertices to the neighboring processors and this significantly reduces the amount of communication that needs to be performed in each RC step.

The recombination algorithm for closeness centrality is presented in Figure 2. During the communication step (lines 9 - 15), each processor communicates the updated boundary DVs to its neighboring processors. In order to avoid network flooding [21] and obtain predictable performance, we employed a personalized all-to-all communication schedule in which only one message traverses the network at any given time. Although for P processors this schedule takes $O(P^2)$ steps, it prevents network flooding. Each processor propagates the updated boundary vertices information to its local sub-graph by applying Floyd-Warshall's algorithm. During each recombination step, the RC algorithm checks if there are any dynamic updates to the graph. To increase the accuracy of the results being propagated, dynamic changes are applied to the sub-graphs before the local computations are performed. The maximum number of recombination steps required to obtain the final APSP values on a static graph is bounded by the number of processors p. However, on a dynamic graph, the number of RC steps required to obtain the final solution is based on the arrival of dynamic updates. Therefore, when analyzing dynamic graphs, the RC algorithm halts when there are no more updates to be shared between processors.

D. Anytime Anywhere Approach for Edge Addition

A key aspect of edge addition is that it can improve shortest path distances or create new shortest paths; unlike edge deletion, where the existing shortest path distances could deteriorate. Since the anytime design is based on continuously improving the existing results, dynamic changes (including edge/node additions or deletions) can be efficiently incorporated in our anytime anywhere framework. The core idea we utilize here is the fact that not all shortest paths in the graph will be affected by edge additions. Therefore, the shortest paths that are not affected by edge additions need not be recomputed. The unaffected sub-paths need not be recomputed and, in addition, can be used to expedite the recomputation of affected shortest paths.

```
1.  INPUT: {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  //Adj. list of vertices V_i of sub-graph 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 added
7.  INPUT: W' = {w'_1, ... , w'_j, ... , w'_l}  //weights of the edges to be added
8.  INPUT: {P'_1, ... , P'_j, ... , P'_l}  //processor containing the vertex a_j of the edge to be added
9.  INPUT: {P''_1, ... , P''_j, ... , P''_l}  //processor containing the vertex b_j of the edge to be added
10. FOR EACH processor P_i do in parallel
11.   FOR EACH edge (a_j, b_j) to be added
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] > DV_i[u][a] + w'_j + DV_i[b][v]
21.             DV_i[u][v] = DV_i[u][a] + w'_j + DV_i[b][v]
22.           END IF
23.         END FOR
24.       END FOR
25.     END IF
26.     IF a_j ∈ V_i //update adj. list after adding edges
27.       ADD edge (a_j, b_j) to ADJ_i[a_j] and ADJ_i[b_j]
28.       IF b_j ∉ V_i and ADJ_i[b_j] is empty // for cut-edge addition
29.         Notify P''_j to start sending DV of b_j to P'_j
30.       END IF
31.     END IF
32.   END FOR
33. END FOR
```

Figure 3. Pseudo-code 2: Anytime anywhere approach for edge addition

When a new edge (a, b) is added, each processor P_i checks if the new edge has affected any previously computed shortest path values (line 18 - 24). This can be done by performing the following assessment, $DV_i[u][v] > DV_i[u][a] + w'_j + DV_i[b][v]$, where $u \in V_i$, $v \in V$ and w'_j is the weight of the new edge. This assessment requires DV of the target vertex b; hence, it is broadcasted by the processor that owns the vertex b. Affected paths are updated by incorporating the new edge information (line 21, Figure 3). After applying these changes, the DVs of boundary vertices that have been updated by the edge additions are propagated to the neighboring processors.

V. ALGORITHM ANALYSIS

In this section, we have presented the run time complexity of our algorithm. Using the same naming conventions as in our previous work [8], let $G = (V, E)$ be the graph where V is the set of vertices and E is the set of edges ($|V| = n, |E| = m$).

Given a set of processors $\{P_1, P_2, \dots, P_P\}$, during the decomposition phase, vertex set V of the input graph is partitioned into P distinct subsets $\{V_i\}_{i=1}^{P}$ such that $|V_i| = O\left(\frac{n}{P}\right)$. Each processor P_i is assigned a sub-graph $G_i = (V_i, E_i)$ which is induced on the vertex set V_i.

We used the LogP [22] distributed memory model to analyze the runtime for the various phases of our anytime anywhere algorithm. However, runtime analysis of the domain decomposition phase is not provided since it is performed only once and the dynamic changes do not require graph partitioning in our approach. Please note that the analysis provided in the following sub-sections A-B has been discussed in previous work [8].

A. Initial Approximation

In the IA phase, each processor P_i calculates all pairs shortest paths values for the sub-graph G_i. Using Dijkstra's single source shortest path algorithm it takes $O(E_i \log V_i)$ to calculate the shortest path for one vertex. In our implementation we used a multi-threaded version of Dijkstra's algorithm to calculate shortest paths for all the vertices in V_i; therefore, it takes $O\left(\frac{n^3 \log \frac{n}{P}}{\tau P^3}\right)$, where τ is the number of threads used. In the IA phase, there is no information sharing among processors hence the results obtained from the IA phase provide preliminary and approximate results.

B. Recombination (RC)

During the RC phase, the partial results obtained in the IA phase are iteratively combined and refined to produce the final solution. The RC phase has three main steps – communicating the distance vector of the boundary vertices (Figure 2, lines 9 - 15), updating the boundary vertex values using the information received and propagating the updated values to the rest of the local sub-graph. Furthermore, dynamic changes to the graph are handled during the RC phase. A vertex, which has an edge whose endpoints belong to different processors, is referred as a boundary vertex and the edge is referred as a cut-edge. The total size of DVs of the boundary vertices in processor P_i is $O(nC_i)$, where C_i is the number of boundary vertices in the processor P_i. Updating the boundary vertices with the information received from neighboring processors takes $O(n\gamma_i C_i)$, where γ_i is the maximum number of cut-edges for any boundary vertex in P_i. Based on previous studies [23][24][25] we approximate $\gamma_i \leq \frac{n}{\log n}$ for graphs with scale-free property. All pairs shortest paths calculation is performed to propagate the updated boundary vertices values to the rest of the sub-graph and it takes $O\left(\frac{n^3}{P}\right)$. These three steps are performed iteratively until there are no more changes or for the size of the longest processor chain. Hence, for a system with p processors, in the worst case, it takes $P - 1$ steps. The total runtime for the recombination phase is $O\left(\frac{n^3}{P} + \frac{n^3}{\log n} + \frac{n^2}{\alpha} P^2 L + n^2 P^2 g\right)$, where L and g represent the latency and gap from the LogP model, and α is the size of the messages sent between processors during recombination steps. The message size (α) should be selected to maintain a lightly loaded network and is bounded by the memory capacity of the processor.

C. Anytime Anywhere Approach for Edge Addition

As mentioned in previous section, dynamic changes to the graph such as edge additions are performed during the recombination phase. Here we analyze the cost of performing edge additions in our algorithm.

- Figure 3, lines 12 -16: Let (u, v) be the edge that has to be added such that $v \in V_i$, then the distance vector of the target vertex v is broadcasted by the processor P_i to all other processors. Here we used a tree broadcast approach and it takes $O(l \log P(L + ng))$, where l is the number of edges to be added.
- Figure 3, lines 18 – 24: The comparison to find and update the affected shortest paths in a processor P_i takes $O\left(l \frac{n^2}{P}\right)$

Therefore, the overhead for incorporating l edge additions at a specific recombination step is $O\left(l \log P\,(L + ng) + l \frac{n^2}{P}\right)$. Additional recombination steps may be required to propagate path changes brought on by the edge additions, to other processors. However, the asymptotic runtimes of additional recombination steps is unchanged.

VI. Implementation and Experimental Results

In order to validate our anytime anywhere approach for handling edge additions during centrality measurements, we implemented our algorithm on a distributed Linux cluster (32 compute nodes) connected with 1 Gb/s Ethernet network. Each compute node has 1.8 GHz dual Intel Xeon E5 processors with 8 cores per processor and 32 GB of memory. The algorithms were implemented in C++ programming language and used MPICH's Message Passing Interface (MPI) library and OpenMP.

A. Experimental Setup

We evaluated our implementation on an undirected scale-free graph generated using the Pajek[3] tool. Since our focus in this paper is on network dynamism and since this is an initial validation of our edge addition methodology, we decided to conduct experiments on medium sized graphs. We will look at experimental results with large sized dynamic graphs in future work. Since there is no other work that closely aligns with our method, we have implemented a baseline algorithm to compare the performance of the anytime anywhere approach. The baseline algorithm does not have anytime anywhere characteristics, therefore, it has to restart the analysis (IA and RC phases) for every edge addition. Some aspects of the

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

experimental design were also discussed in our previous work [8].

B. Results

From the analysis of the additional computations for handling edge changes (see section V.C), we see that the time taken by the anytime anywhere methodology to incorporate edge additions increases with l (number of edge additions). While it is clear from the analysis that there are a number of cases in which the anytime anywhere methodology can efficiently deal with edge additions, it is also clear that there is a tipping point, in terms of number of edge additions, when it can be less efficient to adopt anytime anywhere with respect to the baseline approach. Additionally, the decision to utilize the anytime anywhere methodology, also depends on the recombination step at which the changes occur. For example, if the recombination is in its final steps, then handling the changes using our method may be more efficient than the baseline approach of restarting the processing from scratch. However if the recombination is in its earlier steps, then it may be better to adopt the baseline approach. In order to analyze the impact of this factor and to study the tipping point, described earlier, we conducted experimental results where we compare the runtime results of our anytime anywhere algorithm against the baseline approach for graphs with 50,000 and 100,000 vertices. For brevity, the performance results with graphs of size 50,000 are presented in this section. Note that our experiments did not consider cut edge additions. Since cut edge additions can potentially lead to dramatic changes in shortest paths, a separate experimental study will be performed in future work to understand relevant performance issues.

In the first experiment, changes were introduced (see Figure 4) during initial step (RC0 or the first step), middle step (RC5 or sixth step) and later step (RC9 or tenth step) of the recombination phase. During this experiment, the number of edge additions performed was varied between 5% and 30% of the edges in the initial graph. Our theoretical analysis indicate that there is a tipping point, in terms of increasing number of edge additions, before which there is advantages to using our approach. Figure 4 shows that for the recombination steps RC5 and RC9, experimentally, this tipping point is around 15%. However during the first stage of recombination (RC0), it is clearly less efficient to use the anytime anywhere methodology. In order to understand why this is so, we analyzed the change in the overhead incurred by applying the anytime anywhere approach at different recombination stages (see Figure 5). From the pseudocode in Figure 2, we see that the likelihood that a new edge addition can lead to more recomputations may also depend on the number of paths that have been calculated. In the initial steps, the paths discovered are more likely to be sub-optimal paths. Therefore it is more likely for edge additions to lead to path changes and to triggering of recomputations. This is borne out in our experimental results (see Figure 5). This also helps to further explain the weaker performance of our approach during RC0 step and its improved performance in the later steps (RC5 and RC9) in the first experiment (see Figure 4).

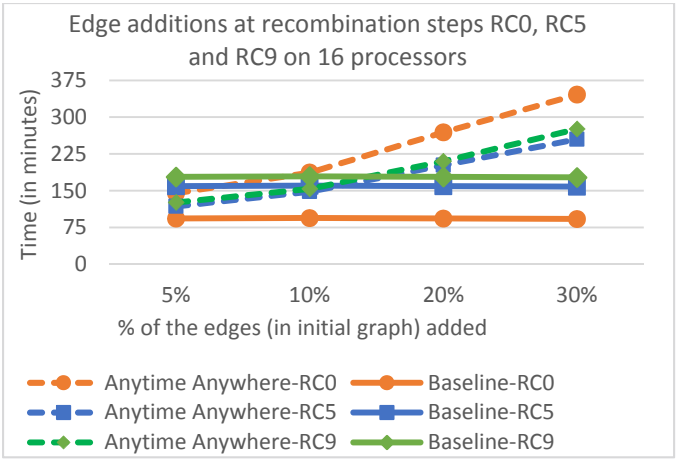

Figure 4. Performance comparison of anytime anywhere vs. baseline for edge additions at recombination steps 0, 5 and 9 using 16 processors on graph with 50,000 vertices

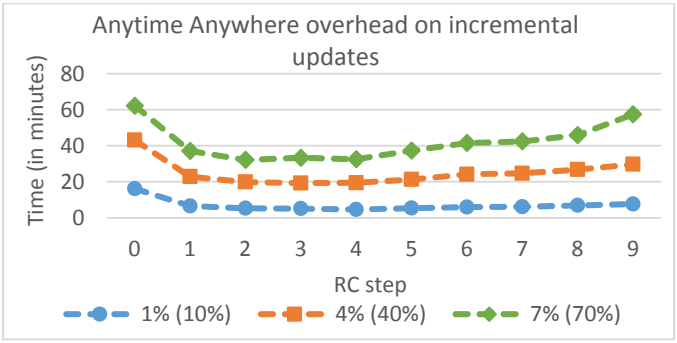

Figure 5. Overhead involved in anytime anywhere algorithm for adding 1%, 4% and 7% of the total number of edges on initial graph for 10 recombination steps

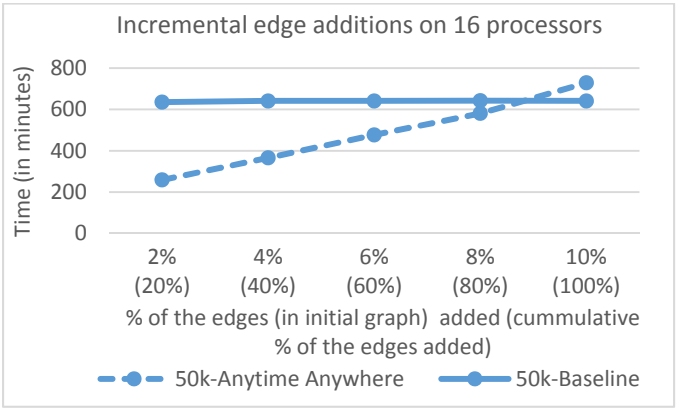

Figure 6. Performance comparison of anytime anywhere vs. baseline incremental edge additions using 16 processors on graph with 50,000 vertices

Our first experiment considers edge changes at a single point of time during analysis. However, in real-world scenarios, changes can be spread across multiple time steps. Increased frequency in edge additions, can be detrimental for the baseline approach as it would have to restart multiple times. In our next experiment (see Figure 6) we performed edge additions which

were spread across all 10 recombination steps. For instance, adding 1% (of the edges in the original graph) edges in each iteration will result in a cumulative increase in the number of edges by 10%. At each step, the changes were equally divided among the processors. As expected, the performance of the anytime anywhere approach is better than the baseline method. In fact, the anytime anywhere approach performs better even for cumulative changes as high as 80%.

VII. Conclusion

In this paper, we presented an anytime anywhere algorithm to efficiently handle new relationship formations during closeness centrality measurements in large and dynamic social networks. Our edge addition algorithm incorporates new edges during the course of analysis and utilizes the partial results to reduce recomputations. Theoretical analysis and empirical evaluations clearly show that the anytime anywhere algorithm performs better on real-world scenarios when the updates occur more dynamically, across multiple time steps. Our experimental results provided important insights into understanding tipping points at which an anytime anywhere approach may start to become inefficient and when it may be better to restart the graph processing. In future work, we plan to continue to explore higher rates of edge addition dynamism to understand tradeoffs for utilizing our methodology for closeness centrality analysis. We also plan to design anytime anywhere algorithms to handle other network dynamisms such as node additions and deletions. In addition, we are interested in investigating an anytime anywhere approach for handling load-imbalances during events such as dynamic graph changes, network congestion, and system failures.

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 IEEE International Conference on Systems, Man and Cybernetics , 2008. SMC 2008, 2008, pp. 3529–3535.

[3] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," Commun. ACM, vol. 51, no. 1, pp. 1–13, 2008.

[4] 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.

[5] 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.

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

[7] 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.

[8] E. E. Santos, J. Korah, V. Murugappan, and S. Subramanian, "Efficient Anytime Anywhere Algorithms for Closeness Centrality in Large and Dynamic Graphs," in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 1821–1830.

[9] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," Proc. 2010 Int. Conf. Manag. Data - SIGMOD '10, pp. 1123–1126, 2010.

[10] 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.

[11] 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.

[12] O. Green, R. McColl, and D. A. Bader, "A fast algorithm for streaming betweenness centrality," 2012 Int. Conf. Privacy, Secur. Risk Trust 2012 Int. Conf. Soc. Comput., pp. 11–20, 2012.

[13] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating Betweenness Centrality," in Algorithms and Models for the Web-Graph, Berlin Heidelberg: Springer, 2007, pp. 124–137.

[14] M. Lee, J. Lee, and J. Park, "QUBE: a Quick algorithm for Updating Betweenness centrality," in Proceedings of the 21st international conference on World Wide Web, 2012, pp. 351–360.

[15] U. Brandes, "A faster algorithm for betweenness centrality," J. Math. Sociol., vol. 25, no. 2, pp. 163–177, 2001.

[16] A. E. Sariyuce, K. Kaya, E. Saule, and U. V. Catalyiirek, "Incremental algorithms for closeness centrality," in 2013 IEEE International Conference on Big Data, 2013, pp. 487–492.

[17] N. Kourtellis, G. De Francisci Morales, and F. Bonchi, "Scalable Online Betweenness Centrality in Evolving Graphs," IEEE Trans. Knowl. Data Eng., vol. 27, no. 9, pp. 2494–2506, 2015.

[18] M. E. J. Newman, "Modularity and community structure in networks," Proc. Natl. Acad. Sci. U. S. A., vol. 103, no. 23, pp. 8577–8582, 2006.

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

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

[21] 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.

[22] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, "LogP: Towards a realistic model of parallel computation," in Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 1993, pp. 1–12.

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

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

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