# Efficient Anytime Anywhere Algorithms for Vertex Additions in Large and Dynamic Graphs

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*—***Over the past decade, there has been a dramatic increase in the availability of large and dynamic social network datasets. Conducting social network analysis (SNA) on these networks is critical for understanding underlying social phenomena. However, continuously evolving graph structures require massive recomputations and conducting SNA is infeasible if the computations have to be restarted for every change. Many recent works have proposed large-scale graph processing systems and frameworks, but less attention has been given to scalable SNA algorithm designs that can efficiently adapt to dynamic graph changes. Moreover, continuously adapting to dynamic graph changes such as node/vertex/actor additions/deletions in a parallel/distributed computational environment can skew the initial graph partitions, leading to load imbalance issues and performance degradation. Previous approaches that focus on computing SNA measures on dynamic graphs either ignore this critical load-balancing aspect or focus only on measures that are straightforward and inherently adjustable to changes in the graph topology. In this work, we have designed an anytime anywhere closeness centrality algorithm that can efficiently incorporate vertex additions while avoiding massive recomputations, by leveraging a generic framework for designing parallel/distributed algorithms called anytime anywhere. Furthermore, we have also performed an analysis of the effectiveness of various processor assignment strategies to mitigate the load imbalances caused by dynamic graph changes.***

*Keywords-social network analysis; parallel and distributed processing; centrality analysis; dynamic graphs; anytime anywhere algorithms; dynamic vertex addition*

## I. INTRODUCTION

Availability of large and dynamic social network information from mediums such as online social media, web graphs and sensor networks have dramatically increased in recent times. These large and dynamic datasets, coupled with social network analysis (SNA) techniques, are helping to extend our understanding of various underlying social processes and phenomena. However, applying static graph analysis methods to analyze the datasets require massive recomputations. Therefore this can become infeasible with large increases in network size and rate of network change. Since, typically, large graph datasets cannot be stored and analyzed in a single machine, they are distributed across multiple machines, and various graph partitioning schemes have been proposed to efficiently distribute the workload across these machines [1][2][3]. These approaches have tended to focus on two main aspects: evenly distributing the vertices across the machines and/or reducing the number of connections among these graph partitions. However, dynamic graph updates such as vertex/edge additions and deletions can skew these initial partitions and create computation and communication load imbalances. In this paper, we focus on efficiently performing vertex additions during the course of SNA. Vertex addition is a natural phenomenon observed across most real-world social networks. For instance, social changes such as new actors joining an online community, adding content to an online social media, and adding new publications to a citation network can all be represented as dynamic vertex additions.

One key challenge when performing dynamic vertex additions is in designing algorithms to efficiently incorporate the new vertices, without incurring massive recomputations. This is important as most analysis tools are built for static networks and require a restart when the underlying network changes. Another key challenge is maintaining the load balance across machines in a parallel/distributed computational environment. Although other graph updates such as edge changes can also induce some degree of load imbalance, the magnitude of the imbalance caused by vertex updates can be substantially higher. This is because vertex updates can affect both the number of vertices and edges in each machine. Furthermore, a single vertex change could initiate multiple edge changes. In addition, social network graphs exhibit scale-free property and have community structures [4][5]. Vertices belonging to the same community share more edges than the vertices belonging to different communities. Hence, assigning the vertices from the same community to a partition could reduce the number of edges between the graph partitions. We refer to such edges with endpoints in different graph partitions as cut-edges. In a number of parallel/distributed graph problems, the amount of communications required between the processors is determined by the total number of cut-edges between these processors. In this work, we present various strategies to assign processors to vertices and evaluate how they affect the number of new cut-edges introduced during vertex additions.

In our previous work [6]–[10], we proposed an overarching parallel/distributed anytime anywhere methodology that can be utilized to design algorithms for processing and analyzing information/data in various static and dynamic conditions. In the anytime anywhere methodology, the term *anytime* refers to the ability of the algorithm to provide non-trivial solutions when interrupted. The quality of these solutions improves in a monotonically non-decreasing manner with relation to the amount of computational resources available to the algorithm. The term *anywhere* refers to the ability of the algorithm to incorporate changes in the underlying data/information and also propagate changes in the analysis. In the domain of large and dynamic social network analysis, we have developed anytime anywhere algorithms; specifically for closeness centrality measurements that can handle edge additions [9] and edge deletions [10]; and for other SNA measurements [6][8]. In this work, we provide an anytime anywhere algorithm to efficiently handle vertex additions while dealing with the challenge of assigning processors to vertices. In addition, we analyze the performance of anytime anywhere algorithms for vertex additions during closeness centrality calculations and evaluate efficiency when combined with various processor assignment strategies. Furthermore, we study the effectiveness of the different processor assignment strategies and understand how they behave under various dynamic scenarios.

## II. Background

Large-scale graph analysis has received a lot of interest and importance in recent times. This has given rise to a number of tools [11][12], libraries [13][14], computational models [15][16], algorithms [17], [18] and load balancing studies [19][20] in this area. However, in these approaches, which are focused on providing systems based solutions, critical aspects such as designing efficient algorithms that can handle dynamic graph changes and load imbalances caused by these changes are often overlooked.

Mizan [11] is a Pregel [16] based graph load balancing system that applies vertex migration techniques for dynamic load balancing. Pregel based systems use the Bulk Synchronous Parallel (BSP) programming model, in which computation is performed in a sequence of supersteps separated by barriers for communication. The Mizan system monitors the runtime metrics for each vertex such as the number of outgoing messages, the number of incoming messages, and the response time for each superstep. Based on these metrics, the vertices are migrated to different processors to reduce the load imbalances. Although Mizan allows adding new vertices and edges at any superstep, it mainly focuses on performing load-balancing for static graph analysis. Furthermore, it does not provide any efficient processor assignment strategies. Instead, it is assumed that the load imbalances will be remedied in future steps using the vertex migration procedure. Vaquero *et al*. [19] proposed an adaptive load balancing system that is based on dynamic vertex migrations. This Pregel based system utilizes a label propagation heuristic. During every iteration, each vertex decides whether to stay in the current partition or to migrate to a different one based on the locality of the highest number of neighboring vertices. Similar to Mizan, this work does not consider any initial vertex placement strategies. Moreover, this work buffers the dynamic graph updates instead of continuously incorporating the graph changes. However, this can lead to outdated results especially when computing measures such as maximum clique and shortest path calculations. Hermes [20] is a lightweight graph repartitioner that utilizes a dynamic repartitioning algorithm to reduce the number of cut-edges and improve the co-location of vertices. However, their main emphasis is on providing graph data management rather than graph analysis. Furthermore, it does not provide any initial processor assignment strategies.

Although the methodologies discussed above provide general procedures for incorporating vertex/edge changes and for load balancing, they do not exploit the structure of particular graph problems to efficiently deal with dynamic graphs. Instead, these system requires the algorithm designers to grapple with such issues. In this work, our focus is to exploit the properties of social network analysis problems, such as closeness centrality, to formulate efficient ways to incorporate graph changes using an anytime anywhere framework. In our previous work [6]–[10] we have provided efficient anytime anywhere algorithms for closeness centrality analysis that can handle dynamic graph changes such as edge weight changes [6], edge additions [9], and edge deletions [10]. In this work, we provide algorithm designs to handle dynamic vertex additions that can incorporate dynamic graph changes continuously during the analysis. Moreover, we have provided efficient processor assignment strategies for reducing the load imbalances caused during dynamic vertex additions.

## III. Anytime Anywhere Methodology

The anytime anywhere methodology [6]–[10] offers a scalable parallel/distributed framework to design algorithms for processing both static and dynamic information/data. One specific application domain for this methodology is in social networks analysis (SNA), such as in the formulation of SNA algorithms that can handle large-scale dynamic graphs. One possible technique is for the large input graph to be decomposed into smaller sub-graphs and assigned to different processors. Results obtained by analyzing these sub-graphs individually are further combined and refined in multiple steps to obtain the final solution. Moreover, dynamic graph updates such as node additions are efficiently incorporated and the effects of these changes are propagated to the entire network with minimal recomputations. The anytime anywhere methodology performs these tasks in three phases: domain decomposition (DD), initial approximation (IA) and recombination (RC). The significance of these phases, with an emphasis on social network analysis, are discussed below and a more detailed discussion can be found in our previous work [6]–[10].

### A. *Domain Decomposition*

The large input graph is decomposed into manageable sub-graphs (graph partitions) and assigned to different processors in the DD phase. The quality of the initial partition affects the quality of the partial results and the amount of communication required in the successive steps. One possible way to design algorithms in domain decomposition is to minimize the number of cut-edges in each sub-graph. Cut-edges have endpoints belonging to different sub-graphs. Cut-size of a sub-graph is the number of cut-edges in that sub-graph. Sub-graphs with higher cut-size will share more edges with other sub-graphs and could increase the amount of communication performed in the successive steps. Various graph partitioning algorithms which minimize the number of cut-edges can be considered in this phase.

### B. *Initial Approximation*

A preliminary approximation is computed in the IA phase, possibly by individually analyzing the sub-graphs obtained from the DD phase. Algorithms employed in the IA phase should support anytime anywhere properties so that the partial results obtained here can be combined and refined in the RC phase.

### C. *Recombination*

In this phase, each processor iteratively computes the final solution by combining and refining its values based on the information received from neighboring processors. Dynamic graph changes are continuously incorporated into the sub-graphs and the effects of these changes are periodically propagated to the entire network. Furthermore, dynamic network changes especially node additions can skew the number of nodes and cut-edges in each sub-graph. Such network updates can consequently lead to load imbalances. In this work, we provide various processor assignment strategies to handle these issues.

## IV. Anytime Anywhere Algorithm Design and Analysis for Closeness Centrality

Centrality measures are key SNA metrics for identifying important actors/nodes in a social network. Some widely used centrality measures are degree centrality, betweenness centrality, closeness centrality, and eigenvector centrality [21][22]. In this work, we focus on designing efficient algorithms for closeness centrality computations in large and dynamic graphs. Closeness centrality of an actor is the inverse of the sum of all shortest path distances from the actor to all other actors in the social network and therefore requires computation of all pairs shortest paths (APSPs). The computational challenge of calculating APSPs in a large network that is continuously evolving makes closeness centrality computation a particularly challenging and interesting problem.

Given a graph $G(V, E)$ where $V$ is the set of vertices in the graph and $E$ is the set of edges such that $|V| = n$ and $|E| = m$, let $d(u, v)$ represent the shortest path distance between vertices $u$ and $v$, then closeness centrality $C_c$ of a vertex $u \in V$ is given by:

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

Dynamic graphs can undergo various forms of changes such as edge weight changes, edge additions/deletions, and node additions/deletions. In our previous work, we designed and analyzed algorithms to handle edge weight changes [7], edge additions [9], and edge deletions [10]. In this work, we focus on vertex additions. Vertex additions possess similar challenges encountered in edge additions since a vertex addition can consist of one or more edge additions. However, the key challenge that needs to be addressed during vertex additions is that the newly added vertices could skew the initial graph partitions, and this can lead to substantial load imbalances. Furthermore, the magnitude of the change is not always proportional to the degree of load-imbalance. For instance, uniform node additions across all processors would have a lower degree of imbalance when compared to the same number of node changes distributed across only a few processors. Additionally, adding vertices of a higher degree will have a substantial impact on load imbalance when compared to vertices with a lower degree. In addition to the degree of a vertex, the number of cut-edges created by a vertex addition will also be based on the processor to which it is assigned. For instance, assigning a new vertex to a sub-graph that have most of its neighboring vertices will reduce the total number of cut-edges introduced by the vertex addition. In order to handle these challenges, performing vertex additions require efficient strategies such as processor assignment, vertex addition and repartitioning strategies.

Processor assignment strategies provide the ability to assign a newly added vertex to an appropriate sub-graph that aims to minimize load imbalances. Moreover, our anywhere approach for vertex addition efficiently utilizes the processor assignment strategies to incorporate dynamic changes to the network. Two examples of such processor assignment strategies are: round robin based strategy and cut-edge optimization based strategy. Round robin based processor assignment strategies focuses only on distributing the new vertices equally while the cut-edge optimization based processor assignment strategies also look at the relationships between the new vertices to minimize the number of cut-edges. However, for a larger number of changes, the overhead involved in incorporating the dynamic updates using anywhere approach for vertex additions could increase substantially. Therefore, in such cases, it may be better to repartition the whole graph rather than applying dynamic changes. However, instead of restarting the analysis from scratch we can utilize the anytime property of the algorithm and reuse the partial results calculated thus far. The anytime anywhere algorithm for closeness centrality consists of three phases: 1) Domain decomposition (DD), 2) Initial approximation (IA), and 3) Recombination (RC). A more detailed discussion about the DD, IA, and RC phases can also be found in our previous work [6]–[10].

In this section, we also provide analyses for different components of the anytime anywhere algorithms. Some analyses presented here have also appeared in our previous work [9][10], and in those instances, are appropriately cited. The LogP [23] distributed memory model was utilized to analyze the runtime for various phases of the algorithm.

## *A. Domain Decomposition*

The input graph $(G)$ is decomposed into balanced sub-graphs and distributed across the processors in the DD phase. The quality of these partitions in terms of the number of cut-edges and the number of vertices assigned to each processor affects the load balancing and the quality of the partial results obtained in the successive steps. Given a set of processors $P = \{P_1, P_2, \dots, P_P\}$, the vertex set $V$ is partitioned into $P$ distinct sub-sets of vertices $\{V_i\}_{i=1}^{P}$ during the DD phase. Let $G(V, E)$ be the graph where $V$ is the set of vertices and $E$ is the set of edges such that $|V| = n$ and $|E| = m$. Let $G_i(V_i \cup B_i, E_i)$ represent the local sub-graph assigned to each processor $P_i$, where, $V_i \subseteq V$ is the set of vertices assigned to processor $P_i$, $E_i \subseteq E$ is the set of edges that have at least one endpoint(vertex) in $V_i$ , and $B_i \subseteq V$ is set of external boundary vertices for processor $P_i$ . External boundary vertices act as bridges that connect the neighboring sub-graphs to the vertices $V_i$ in the local sub-graph. Any cut-edge optimization based graph partitioning algorithm can be used in this phase. Therefore the runtime for this phase depends upon the algorithm being used.

## *B. Initial Approximation*

The sub-graphs obtained from the DD phase are analyzed individually in the IA phase to obtain the first set of partial results. These partial results provide a preliminary approximation of the entire network. For the closeness centrality computation, each processor computes APSP values for its local sub-graph. In the IA phase, each processor $P_i$ computes the partial results based on the information contained in its local sub-graph $G_i$.

A possible algorithm to implement the IA for closeness centrality analysis is Dijkstra's single source shortest path algorithm. In our previous work [9][10], we applied a multi-threaded version of Dijkstra's single source shortest path algorithm to speed up this computation and leverage the multiple cores, and it takes $O\left(\frac{n^3 \log\frac{n}{P}}{\tau P^3}\right)$, where $\tau$ is the number of threads used. The partial results obtained using Dijkstra's algorithm in the IA phase can be further refined in the RC phase.

## *C. Recombination*

```
1.  INPUT: $\{P_1, \dots P_i, \dots P_P\}$ //set of processors assigned to the problem
2.  INPUT: $n$ //number of vertices in the input graph G
3.  INPUT: $G_i$ //local sub-graph in processor $P_i$
4.  INPUT: $DV_i^0$ //Distance Vectors $(|V_i| \times n)$ for sub-graph $G_i$ generated in the 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 $i \neq j$
11.                 RECV DVs of external boundary vertices from processor $P_j$ in messages of size $\alpha$.
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 $\alpha$.
15.         END FOR
16.         Choose Recombination strategy(ies) based on the constraints
17.         Perform Recombination strategy(ies)
18.     UNTIL $k = P - 1$ OR no more updates in any processor
19. END FOR
```

*Figure 1. Pseudo-code template of the recombination phase for closeness centrality [9][10]*

In the RC phase, each processor refines its partial results by incorporating the updates received from the neighboring processors. This process is repeated in iterative steps until the final solution is obtained. One way of performing this process is by using Distance Vector Routing (DVR) algorithm [24].

In this work, we utilized the recombination algorithm for closeness centrality computation developed in our previous work [9][10], where we applied DVR algorithm to perform incremental graph updates across the processors during the APSP computation. Each vertex in a sub-graph maintains a Distance Vector (DV) to store the current shortest path distances to other vertices in the graph. Boundary DVs are the distance vectors of the boundary vertices. Boundary vertices act as bridges connecting sub-graphs belonging to different processors. During each RC step, the boundary vertices in each processor receive updates from its neighboring processors. Therefore, when propagating updates, it is sufficient to send only the updated values of the boundary DVs and this significantly reduces the amount of communications. We used a personalized all-to-all communication schedule that ensures only one message traverses the network at any given time in order to prevent network flooding and obtain predictable performance. Although our communication schedule takes $O(P^2)$ steps for $P$ processors, it mitigates network flooding. For static graphs, the number of RC steps required to compute final APSP values is bounded by the number of processors $P$. However, the number of RC steps required for dynamic graphs is based on the step at which dynamic changes are incorporated. Hence, during dynamic graph analysis, the refinement of the partial results is continued until there are no more updates to be exchanged between processors.

In previous work [9][10], we provided the runtime analysis for the RC phase, when DVR algorithm is applied to perform incremental updates. Let $C_i$ be the number of local boundary vertices in the processor $P_i$ and $\gamma_i$ be the maximum number of cut-edges for any boundary vertex in $V_i$. Based on previous studies [4][5][25], for the networks with scale-free property, we can approximate $\gamma_i \leq \frac{n}{\log n}$. Since our focus is on social network graphs and as these graphs exhibit scale-free property we use this bound for $\gamma_i$ in our analysis. During each RC step, the processors send and receive DVs of size $O(nC_i)$, where $C_i = O\left(\frac{n}{P}\right)$. Using the information received from the neighboring processors the boundary vertices are updated and this takes $O(n\gamma_i C_i)$. Therefore, the time to share the information and update the boundary nodes at each recombination step takes $O\left(\frac{n^3}{P\log n} + \frac{n^2}{\alpha}PL + n^2 Pg\right)$. When there are no dynamic graph updates the number of RC steps required is bounded by the number of processors $P$. This is because the longest processor chain could be of length $P-1$ and therefore the number of RC steps is bounded by $P-1$ steps. The total runtime to complete the communication and boundary vertices updates is: $O\left(\frac{n^3}{\log n} + \frac{n^2}{\alpha}P^2L + n^2P^2g\right)$. Here $L$ and $g$ represent the latency and gap from the LogP model. The maximum size of a single message exchanged between the processors is represented by $\alpha$. Maximum message size $\alpha$ is bounded by the memory capacity of the processor and is chosen such that the network remains lightly loaded during communications.

*1) Recombination Strategies*

In addition to the information sharing with the neighboring processors, one of the key steps in the RC phase is to perform the recombination strategy(ies). Recombination strategies provide the capability to perform various updates and computations on the graph. These strategies include (but not limited to) static graph updates, dynamic changes such as vertex/edge additions and deletions, processor assignments and load balancing.

The recombination strategy(ies) performed in each iteration (on each processor) varies based on a set of constraints. These constraints help us guide the choice of strategy(ies) based on the requirements during the recombination step. For instance, constraints may include information such as user defined values, system specified thresholds, dynamic change requirements and static refinements. Moreover, these constraints can be expanded based on evolving requirements. Let $\phi$ be this set of constraints. Let $\mathcal{A}_{Con}$ be the algorithm used to choose the recombination strategy, which takes in various inputs including the set of constraints $\phi$. The runtime of $\mathcal{A}_{Con}$ at a recombination step is represented as $T(\mathcal{A}_{Con})$.The algorithm used to perform the recombination strategy is represented by $\mathcal{A}_{RC}$. Depending on the design of the recombination strategy, Algorithm $\mathcal{A}_{RC}$ may take in various inputs including the graph $G$ and dynamic changes to G, such as set of new vertices $V'$ of size $n'$ and set of new edges $E'$ of size $m'$. There are various ways to implement Algorithm $\mathcal{A}_{RC}$, including as a distributed algorithm across the processors. The run time of the recombination strategy at a recombination step is represented as $T(\mathcal{A}_{RC})$.

An example of a basic recombination strategy is the static graph analysis for closeness centrality calculations that was described earlier in section IV.C. In this strategy, the partial results are iteratively combined and refined. In a slight modification to the strategy, the newly obtained values on the boundary DVs can be used to update the DVs of the local vertices using Floyd-Warshall's algorithm, as shown in previous work [9][10]. This will help in providing more up-to-date partial results to the user without having to depend on future recombination steps. Performing updates to the local DVs require an all-pairs shortest path calculation within the local sub-graph. In this case the time taken to perform the local DV updates at a recombination step is $O\left(\frac{n^3}{P^2}\right)$. Therefore, the overall run time of the recombination strategy at a recombination step is:

$$T(\mathcal{A}_{RC}) = O\left(\frac{n^3}{P^2} + \frac{n^3}{P\log n} + \frac{n^2}{\alpha}PL + n^2Pg\right)$$

As mentioned before, in the worst case, the maximum number of RC steps required is $O(P)$. The overall running time of the recombination phase is $O(T(\mathcal{A}_{Con})P + T(\mathcal{A}_{RC})P)$. After substituting for $T(\mathcal{A}_{RC})$, we get:

$$O\left(T(\mathcal{A}_{Con})P + \frac{n^3}{P} + \frac{n^3}{\log n} + \frac{n^2}{\alpha}P^2L + n^2P^2g\right)$$

*a) Vertex Addition Strategy*

The recombination strategy also checks for dynamic changes and incorporates these changes at the end of the refinements during each RC step. In particular, in this work we focus on handling vertex additions. Vertex addition consists of two key steps, determining the processor assignments to the new vertices and incorporating the new information into the existing graph. Figure 2 shows the pseudo-code template for a vertex addition strategy. This recombination strategy is performed on line 17 of the recombination algorithm presented in Figure 1.

```
1. Read dynamic changes input
2. Perform processor placement strategy
3. Perform vertex addition strategy
```

*Figure 2. Pseudo-code template for vertex addition strategy*

When a set of vertices is added during the course of analysis, it is initially assigned to a particular processor based on a processor assignment strategy. Processor assignment strategies are critical to prevent computation and communication load imbalances during vertex additions. Although there are many ways to approach this issue, to demonstrate the capability of our framework we focus on the following two key factors that will affect the running time, the number of vertices and the number of cut-edges assigned to each processor. If the number of vertices increases in one or more processors compared to others it can lead to a computational load imbalance. Similarly, if the number of cut-edges increases in one or more processors compared to others then it can result in a communication load imbalance. Both computation and communication load imbalances will lead to an increase in overall run-time.

Specifically in this paper, we choose two processor assignment strategies, namely round robin based processor assignment strategy (RoundRobin-PS) and cut edge optimization based processor assignment strategy (CutEdge-PS). The RoundRobin-PS, which represents the straight forward and easy to implement approach for load balancing, distributes the vertices evenly across the processors and has minimal overhead. However, it does not consider the relationships or connections between the newly added vertices. Consequently, when vertices with community structure are added, the RoundRobin-PS method can create a higher number of cut-edges when compared to the second approach, CutEdge-PS. The CutEdge-PS approach is a more sophisticated approach, which considers the newly added vertices and the edges between these vertices as an independent graph. This graph is partitioned into sub-graphs based on the number of processors such that it minimizes the number of cut-edges between these new partitions. Let $\mathcal{A}_{Plac}$ denote the algorithm used for the processor assignment strategy in a recombination step, with a run time of $T(\mathcal{A}_{Plac})$. In RoundRobin-PS, new vertices are assigned to the processors in a circular fashion and therefore the run time of the processor algorithm strategy is $O(n')$. However, any cut edge optimization based graph partitioning algorithm can be substituted for CutEdge-PS and therefore the runtime of CutEdge-PS will be dependent on this choice.

```
1.  INPUT: $P = \{P_1, ... P_i, ... P_P\}$ //set of processors assigned to the problem
2.  INPUT: n //number of vertices in the input graph G
3.  INPUT: $G_i$ //sub-graph assigned to processor $P_i$
4.  INPUT: $ADJ_i$ //Adjacency list for $V_i$ in sub-graph $G_i$
5.  INPUT: $DV_i$ //Distance Vectors $(|V_i| \times n)$ for sub-graph $G_i$
6.  INPUT: $\rho$ //set of new vertices and edges to be added
7.  INPUT: $W' = \{\langle w_{l,1}, ..., w_{l,j}, ..., w_{l,m''}\rangle\}$ //weights of the new edges to be added
8.  INPUT: $P' = \{P'_1, ..., P'_j, ..., P'_{n'}\}$ //processors where the new vertices need to be added
9.  INPUT: $P'' = \{\langle P''_{l,1}, ..., P''_{l,j}, ..., P''_{l,m''}\rangle\}$ //processors containing the target vertices of the new edges to be added
10. FOR EACH processor $P_i$ do in parallel
11.   FOR EACH vertex $a_l$ to be added
12.     IF $a_l$ has to be added to sub-graph $G_i$
13.       ADD vertex $a_l$ to $ADJ_i$
14.       ADD new row and column to $DV_i$ and initialize to $\infty$
15.     ELSE
16.       ADD new column to $DV_i$ and initialize it to $\infty$
17.     END IF
18.   END FOR
19.   FOR EACH new vertex $a_l$
20.     FOR EACH edge $(a_l, b_j)$ to be added
21.       IF $b_j \in V_i$
22.         SEND row $DV_i[b_j]$ to all other processors //using tree broadcast
23.       ELSE
24.         RECV row $DV_i[b_j]$ from processor $P''_j$
25.       END IF
26.       IF $w_{l,j} < DV_i[a_l][b_j]$
27.         FOR EACH $u \in V_i$
28.           FOR EACH $v \in V$
29.             IF $DV_i[u][v] > DV_i[u][a_l] + w_{l,j} + DV_i[b_j][v]$
30.               $DV_i[u][v] = DV_i[u][a_l] + w_{l,j} + DV_i[b_j][v]$
31.             END IF
32.           END FOR
33.         END FOR
34.       END IF
35.       IF $a_l \in V_i$ AND $b_j \in V_i$ //update adj. list after adding edges
36.         ADD edge $(a_l, b_j)$ to $ADJ_i[a_l]$ and $ADJ_i[b_j]$
37.       ELSE IF $a_l \in V_i$ AND $b_j \notin V_i$
38.         ADD edge $(a_l, b_j)$ to $ADJ_i[a_l]$
39.         Notify $P''_j$ to start sending DV of $b_j$ to $P'_j$
40.       ELSE IF $a_l \notin V_i$ AND $b_j \in V_i$ ADD edge $(a_l, b_j)$ to $ADJ_i[b_j]$
41.         Notify $P'_j$ to start sending DV of $a_l$ to $P''_{l,j}$
42.       END IF
43.     END FOR
44.   END FOR
45. END FOR
```

*Figure 3. Pseudo-code 3 Anywhere approach for vertex addition*

Vertex additions consist of two key steps. First, the new vertices are added to the existing sub-graphs. Second, the edges corresponding to the new vertices are added. Let $a_l$ be the new vertex that is added to a sub-graph $G_i$ in processor $P_i$. The vertex $a_l$ is added to the DV of processor $P_i$. Since each existing vertex can now have a path to the new vertex, DVs of existing vertices across all processors are extended to store this new value (Figure 3, line 11 - 18). The vertex additions are simultaneously performed across all processors. Once the new vertices are added, we perform edge additions based on the anytime anywhere edge addition algorithm described in our previous work [9]. Let $(a_l, b_j)$ be the new edge that has to be added. The edge addition algorithm first examines whether the new edge has affected any previously computed shortest path values (Figure 3, line 28 - 32). This is done by evaluating the inequality $DV_i[u][v] > DV_i[u][a_l] + w_{l,j} + DV_i[b_j][v]$, where $u \in V_i$ and $v \in V$. The paths affected by the new edge addition are updated and propagated to the neighboring processors.

Let $m''$ denote the degree of some new vertex added to the graph. Since the existing vertices could have a path to the newly added vertex, the DVs of the existing vertices are updated. Based on the add edge algorithm [9], to add O $(m'')$ edges it takes:

$$O\left(m''L\log P + m''ng\log P + m''\frac{n^2}{P}\right)$$

To add $n'$ vertices, there is also an additional cost to resize and maintain the DVs. Assuming that the size of the vector is doubled every time the resize takes place, this operation has a cost of $O\left(\frac{(n+n')^2}{P}\right)$. Therefore, the overall running time for vertex addition strategy to add $n'$ vertices and $m'$ edges, at a recombination step is:

$$O\left(T(A_{Plac}) + m'L\log P + m'ng\log P + m'\frac{n^2}{P} + \frac{(n+n')^2}{P}\right)$$

*b) Repartition Strategy*

In this subsection, we describe a strategy for dealing with scenarios where large network changes can render the previously discussed vertex addition strategy inefficient. In such cases, it may be more efficient to repartition the entire graph rather than using our vertex addition strategy. However, restarting the entire analysis from scratch incurs significant overhead. In our approach, we can reduce this overhead by leveraging the properties of the algorithm and reusing the partial results computed in previous recombination steps. In our repartitioning strategy (Repartition-S), the entire graph along with the newly added vertices are repartitioned. Although repartitioning the entire graph has an overhead, this method can effectively reduce the total number of cut-edges when compared to other methods. Note that, in Repartition-S, we do not perform the vertex addition strategy used in RoundRobin-PS and CutEdge-PS.

Any cut-edge optimization based graph partitioning algorithm can be used to repartition the graph and therefore the time take to repartition the graph is dependent on the algorithm used. Here, repartitioning could reassign an existing vertex to a different processor and this requires communicating the partial results to the appropriate processor. This communication step uses the communication schedule discussed in section IV.C. It should be noted that in the Repartition-S, the DVs of the existing vertices are not immediately updated based on the new vertex additions as in RoundRobin-PS or CutEdge-PS. This can lead to additional RC steps in the Repartition-S method.

## V. Experimental Results

We evaluated the performance of our approach on a distributed Linux cluster with 32 compute nodes connected over 1 Gb/s Ethernet network. Each compute node has dual Intel Xeon E5 (1.8 GHz) processors with 8 core per processor and 32 GB of memory. We implemented our algorithms using the C++ programming language, MPICH's Message Passing Interface (MPI) library was used for inter-processor communications and OpenMP for implementing the multithreaded aspects of our algorithm. All our experiments were conducted on 16 processors with graphs of 50,000 vertices. Since real-world social networks exhibit scale-free property, we generated undirected scale-free graphs for our evaluations using the Pajek[1] network analysis tool.

### A. Experimental Setup

Based on the theoretical analysis described in section IV.C, we expect both the number of vertex additions and the stage of analysis, at which the new vertices are added, to have an impact on the performance of our anytime anywhere vertex addition algorithm. The number of vertices added affects the time taken to perform dynamic updates. In particular, for a larger number of vertex additions the overhead to incorporate these changes can be substantially higher. Based on our analysis, this dynamic vertex addition cost could be avoided by performing the repartition strategy. We evaluate the performance of our recombination strategies to understand how they behave under different dynamic conditions.

In our implementation, we carefully selected algorithms and strategies for the various components of our framework to evaluate the impact of vertex additions and load balancing strategies on the overall run time. Moreover, the algorithms used in DD, IA and the recombination strategies can be modified without affecting other components of the framework. This gives us the flexibility to use various algorithms and communication schedules to better understand the efficiency of our framework.

We implemented the DD phase using Parallel Graph Partitioning and Fill-reducing Matrix Ordering (ParMETIS) algorithm [2]. ParMETIS is a prominent parallel graph partitioning algorithm that produces balanced partitions while minimizing the total number of cut-edges across the partitions. Since Repartition-S performs graph partitioning on the entire graph, we reused the algorithm from the DD phase. For the experiments, we implemented CutEdge-PS using METIS [26] serial graph partitioning algorithm. Each processor computes the METIS partition for the newly added vertices and the partition with the lower number of cut-edges is chosen for processor assignment. Also, it must be noted that CutEdge-PS is designed to consider only the new vertices and the edges between them. Considering both the existing and new vertices during processor assignment may require relocation of the existing vertices to other processors. Relocating existing vertices would require communicating the vertex information and its partial results, which in turn would increase the vertex additions overhead. Therefore, in this paper, we do not consider migrating the existing vertices and their edges in this approach, but will consider this in future work.

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

## B. Results

In this section, we report the performance results for the anytime anywhere vertex addition algorithm during closeness centrality computations. As mentioned in the previous section, both CutEdge-PS and Repartition-S attempt to minimize the number of cut-edges and also balance the number of vertices across the processors. The following experiments were conducted to compare the performance of various strategies.

### 1) Anytime Anywhere and Baseline Restart

In a large-scale social network analysis, incorporating dynamic changes during the course of the analysis is critical since static graph analysis methods that restart the computation from scratch for every change can take a significant amount of time and lead to obsolete results. In this experiment, we compare the performance of the anytime anywhere approach to a baseline restart method that restarts the computation from scratch for every change. Figure 4 shows the results for the baseline restart and the anytime anywhere approach with RoundRobin-PS. The RoundRobin-PS is a simple approach that evenly distributes the newly added vertices to the sub-graphs without considering the edges or the relationships between them. By comparing the baseline restart method with the RoundRobin-PS we can see the effectiveness of the anytime anywhere methodology even when a simple processor assignment strategy is adopted. The baseline restart method does not have an anytime property (e.g. cannot reuse any partial results). However, the anytime anywhere approach (RoundRobin-PS) reuses the partial results and updates only the parts of the network that are affected by the dynamic graph changes. This shows the efficiency of the anytime anywhere algorithm for performing dynamic vertex additions.

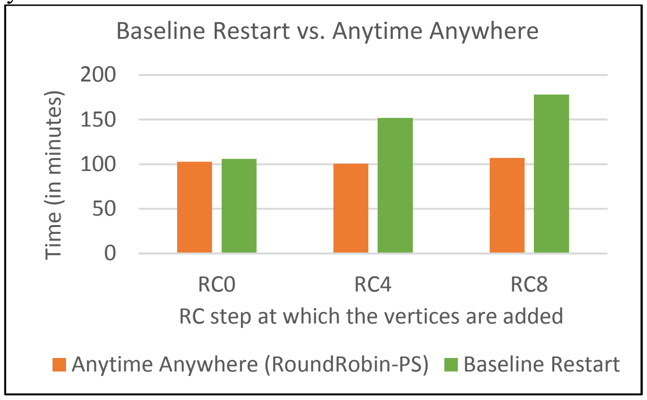

*Figure 4. Performance comparison of anytime anywhere and baseline restart methods for 512 vertex additions using 16 processors on a graph with 50,000 vertices*

### 2) Vertex Additions at Single RC Step

One of the key challenges when performing dynamic vertex additions is to reduce the computation/communication load imbalances among the processors. Our previous experiment showed the efficiency of the anytime anywhere algorithm in performing vertex additions, however, it does not go in detail into the effects of the load balancing strategies. In this experiment, a set of vertices ranging between 500 and 6000 are added at different stages of the analysis using CutEdge-PS, RoundRobin-PS, and Repartition-S. To test the effects of CutEdge-PS, the new vertices that are added in our experiments were extracted from a larger graph using Pajek's Louvain[2] community extraction method.

Figure 5 shows the performance results for vertex additions during the initial stages of the analysis (RC0). For a smaller number of vertex additions, the time taken to propagate the dynamic updates is less compared to the time taken for partial result communication and partitioning, therefore RoundRobin-PS and CutEdge-PS perform better than Repartition-S. However, for a substantially higher number of changes, the overhead to propagate dynamic updates increases, leading to Repartition-S outperforming both RoundRobin-PS and CutEdge-PS. Repartition-S utilizes the anytime property of the algorithm to reduce the number of recomputations and display better performance when there is a substantial change in the underlying network. However, the downside for using Repartition-S is that additional recombination steps may be needed. Figure *6* shows the performance comparison of different strategies for vertex additions happening at the later stages of the analysis (RC8). Similar to the previous results both CutEdge-PS and RoundRobin-PS perform better when the number of changes is low. However, as the number of changes increases the overhead to perform anywhere vertex addition also increases, therefore Repartition-S performs better when the number of changes is high.

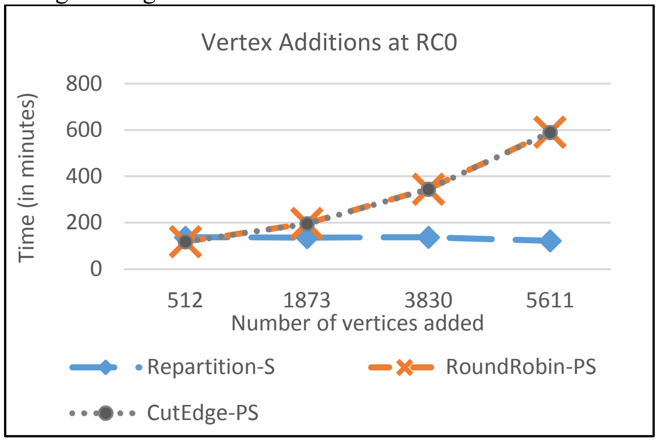

*Figure 5 . Performance comparison of different strategies for vertex additions at recombination step 0 (RC0) using 16 processors on a graph with 50,000 vertices*

Even though the performance gain from the CutEdge-PS when compared to the RoundRobin-PS is small, analyzing the final network/graph produced after adding the new vertices shows that the number of new cut-edges (Figure 7)

[2] http://mrvar.fdv.uni-lj.si/pajek/community/LouvainVOS.htm

created by CutEdge-PS was less when compared to the number of new cut-edges from the RoundRobin-PS. Therefore, when compared to RoundRobin-PS, for a large set of vertex additions with multiple communities CutEdge-PS may effectively reduce the number of cut-edges and therefore reduces the running time.

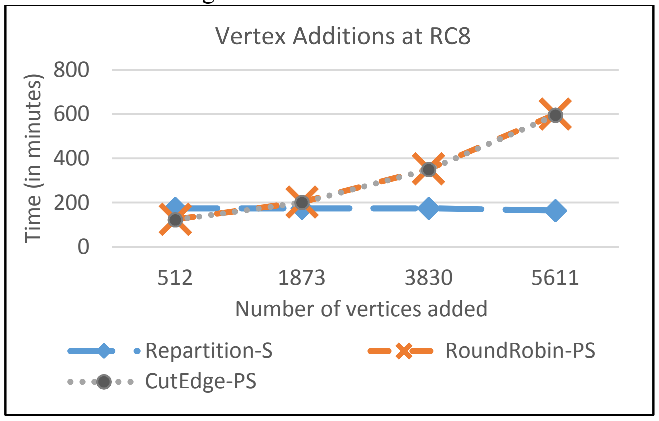

*Figure 6. Performance comparison of different strategies for vertex additions at recombination step 8 (RC8) using 16 processors on a graph with 50,000 vertices*

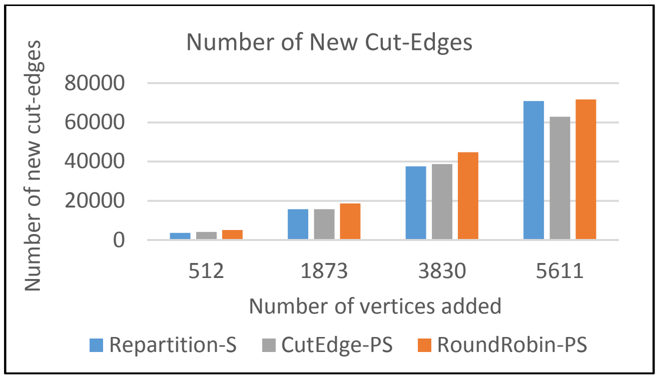

*Figure 7. Number of new cut-edges created by different strategies on a graph with 50,000 vertices using 16 processors*

### 3) *Incremental Vertex Additions*

In real-world scenarios, the networks evolve continuously, resulting in smaller number of changes over the course of analysis instead of one large update at a particular stage of the analysis. In order to capture this effect, in this experiment, we perform vertex additions distributed across multiple stages (10 RC steps) of the analysis. For instance, in the 5611 vertex additions experiment, at each RC step, approximately 560 vertices are added. Figure 8 shows the results for incremental vertex additions on a graph with 50,000 vertices. Baseline restart approach has to restart the computation for every update and therefore performs significantly slower than other methods. For a smaller number of updates, both CutEdge-PS and RoundRobin-PS performs better than Repartition-S. This is due to the fact that the Repartition-S has to do partitioning and redistribute the partial results every time there is an update, this takes much longer than the overhead involved in CutEdge-PS and RoundRobin-PS. However, as the number of vertex additions increases, the overhead to perform anywhere vertex addition algorithm in both CutEdge-PS and RoundRobin-PS increases. Therefore, in these experiments, we can see that for a large number of vertex additions, Repartition-S performed better.

### 4) *Summary*

Our results show that anytime anywhere methodology can be used to design efficient algorithms for dynamic vertex additions. However, it is also clear that there is no one strategy, such as the processor assignment strategy, that is efficient for all situations. From our experimental results, we provide the following insights:

- For a relatively smaller number of vertex additions and for higher rates of network changes, the anytime anywhere approach of utilizing only the new information and incorporating the changes as and when the changes occur provides the most efficient and accurate results (as shown in Figure 8).
- For a larger number of changes happening at a single step of the analysis, the overhead involved in updating and changing the existing results increase considerably. Therefore, the Repartition-S uses the anytime property to provide an efficient middle ground, between restarting from scratch and using the vertex addition strategy, which yields better results when the number of vertex additions are large, as seen in Figure 5 and Figure *6*.

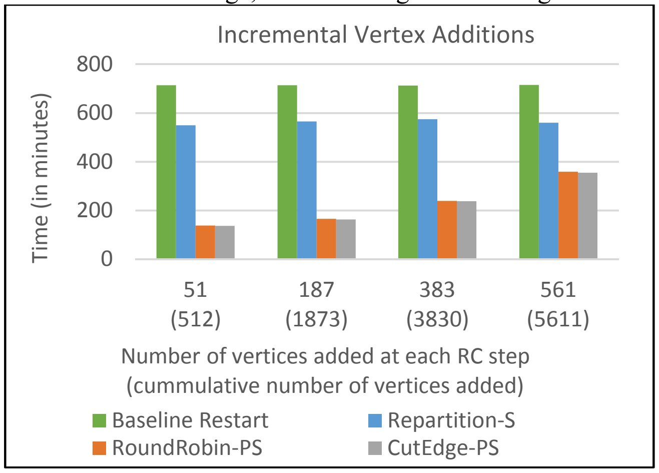

*Figure 8. Performance comparison of different strategies for incremental vertex additions using 16 processors on a graph with 50,000 vertices*

## VI. CONCLUSION

In this paper, we proposed an anytime anywhere algorithm for handling vertex additions during closeness centrality computations. Both theoretical analysis and empirical evaluations illustrate the effectiveness of the anytime anywhere approach in handling dynamic vertex updates. Furthermore, we analyzed the effectiveness of different processor assignment strategies and presented their performance for different forms of vertex additions such as single step and incremental additions. The RoundRobin-PS and the CutEdge-PS performed better for real-world scenarios, where the dynamic updates happen continuously over time. However, Repartition-S performed better when a large number of vertex updates happen at a single stage of the analysis. This illustrates how various processor assignment and repartitioning strategies can be combined with the anytime anywhere vertex addition approach to handle different forms of vertex additions. In the future, we plan to design anytime anywhere algorithms to also handle vertex deletions and develop graph rebalancing strategies to deal with load imbalances caused by these changes. We also plan to investigate anytime anywhere methodologies to handle issues such as fault tolerance in the cloud and other parallel/distributed platforms.

## REFERENCES

[1] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1. pp. 359–392, 1998.

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

[3] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, 2006, p. 10 pp.

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

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

[6] 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.

[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] 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.

[9] E. E. Santos, J. Korah, V. Murugappan, and S. Subramanian, "Effectively Handling New Relationship Formations in Closeness Centrality Analysis of Social Networks Using Anytime Anywhere Methodology," *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*. pp. 354–361, 2016.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] G. Malewicz *et al.*, "Pregel : A System for Large-Scale Graph Processing," in *SIGMOD'10*, 2010, pp. 135–145.

[17] 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.

[18] 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.

[19] L. M. Vaquero and C. Martella, "Adaptive Partitioning of Large-Scale Dynamic Graphs," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, 2013, p. 35:1--35:2.

[20] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases," in *EDBT*, 2015.

[21] M. E. J. Newman, *Networks. An introduction.* New York, NY, USA, 2010.

[22] K. Okamoto, W. Chen, and X. Y. Li, "Ranking of closeness centrality for large-scale social networks," in *International Workshop on Frontiers in Algorithmics*, 2008, vol. 5059 LNCS, pp. 186–195.

[23] D. Culler *et al.*, "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.

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

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

[26] 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.