SE-CDA: A Scalable and Efficient Community Detection Algorithm

Dhaval C. Lunagariya
Dept. of Computer Science and Engineering
National Institute of Technology
Warangal, India
er.dcpatel@gmail.com

D.V.L.N. Somayajulu
Dept. of Computer Science and Engineering
National Institute of Technology
Warangal, India
soma@nitw.ac.in

P. Radha Krishna
Infosys Labs
Infosys Limited
Hyderabad, India
radhakrishna_p@infosys.com

Abstract-- **Detecting communities is of great importance in various disciplines such as social media, biology and telephone networks, where systems are often represented as graphs. Community is formed by individuals such that those within a group interact with each other more frequently than with those outside the group. The communities have different properties such as node degree, betweenness, centrality, cluster coefficient and modularity. Discovering communities from social networks of big data scale on a single sequential machine is a tedious task. In this paper, we present a Scalable Community Detection Algorithm which relaxes the performance issues due to many I/Os.**

We adopt Girvan–Newman's modularity based hierarchical community detection algorithm in bottom up approach and proposed an approximation algorithm for community detection in a distributed environment. We developed our approach using MapReduce and Giraph computing platforms. Experimental results demonstrate that the proposed approach is more efficient than standard MapReduce approach and easily scaled to graph of any size.

Index Terms— **Community Detection, MapReduce, Giraph**

I. INTRODUCTION

Analyzing social graphs plays a big role in extracting relevant and personalized information for users, such as results from a search engine or news in an online social networking site. Due to increased use of social media and networks in a massive scale, the size of social graphs is also increasing at the same pace. Moreover, the businesses are demanding newer relationships and interactions through web and on-line social platforms. Many real-life applications (such as the telephone networks, the world-wide web, Internet, transportation networks, citation network and social interactions) produce a large amount of data which can be modeled as a graph. A large graph usually has millions of vertices with billions of edges.

Existing clustering algorithms and theory on graphs are only suitable for small graphs. With massive amounts of data continuously being collected and stored, many industries and researchers are becoming interested in finding community from large social graphs. The cost of time and space for processing large-scale graphs usually exceed the ability of a concentrated computing system. This necessitates parallel and distributed computing strategies for accelerating algorithms performance in order to discover communities from large graphs.

Most graph algorithms need iterative computations, which needs a chain of MapReduce (MR) jobs. In chain of MRs, transferring the graph data between two consecutive jobs during each iteration degrade the performance. In order to resolve this problem, Google developed a parallel graph processing system called Pregel [1] based on bulk synchronous parallel (BSP) model [2]. Two open source projects namely Giraph [3] and Hama [4] based on BSP are also developed. Apache Giraph is a large-scale graph processing platform, which is implemented on top of existing hadoop cluster. It uses MapReduce job without the *reduce* stage. Apache Hama is distributed computing framework for massive scientific computation.

To the best of our knowledge, the only Scalable Community Detection Algorithm was described in [5] which targets at web-scale graph data. Their motive is to relax scalability bottleneck with the help of cloud computing, more specifically, MapReduce programming model. For graph processing, the standard MapReduce programming model suffers from performance issues because the graph state has to be passed from one phase to the other in every iteration generating a lot of I/Os. This is because standard MapReduce lacks built-in support for the iterative process. To achieve better performance, we developed an efficient algorithm with the use of Giraph in this work.

In the literature, community detection approaches are broadly categorized into four: *node-centric*, *group-centric*, *network-centric* and *hierarchy-centric* [6]. In Node-Centric Community Detection technique, each node in a group satisfies certain properties such as Complete Mutuality, Reachability of members, Nodal degrees and Relative frequency of Within-Outside Ties. Finding cliques from graph falls under this technique. Clique Percolation Method (CPM) [7] allows overlap between the communities. Here, two k-cliques are adjacent if they share *k-1* nodes, and a community is equivalent to a percolation cluster of k-cliques, in which any k-clique can be reached from any other k-clique via sequences of k-clique adjacency.

Group-Centric Community Detection techniques require whole group to satisfy certain properties. Network-Centric Community Detection techniques partition the whole network into different disjoint sets. Clustering based on vertex similarity, Latent space models, Block model approximation, Spectral clustering and Modularity maximization are the approaches of this technique [6].

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

Hierarchy-Centric Community Detection technique constructs a hierarchical structure of communities. Girvan–Newman [8] algorithm is well known agglomerative algorithm for finding hierarchical community from graph dataset. Initially, the algorithm considers that all the nodes in a graph are single stand-alone communities. Then it calculates the modularity between every node pairs and repeatedly merges pair of two communities with largest modularity which results in single community. This algorithm follows greedy strategy and the community structures are built in a bottom-up fashion. The complexity of algorithm is O(n^3), where *n* is the number of nodes. So, it is not scalable for large graphs which include millions of edges. Another limitation of Girvan–Newman bottom-up approach is with respect to its too much processing time, since every iteration of the algorithm merges only one node pair. To address this issue, in this work, we propose an approximation algorithm for the same. In approximation approach, instead of merging single node pair during every iteration, it merges top *K* node pairs during every iteration.

In this paper, we develop a scalable and efficient community detection algorithm (SE-CDA) by extending the Girvan-Newman algorithm. We followed BSP model to design parallel algorithm. We considered Hadoop platform (*hadoop.apache.org*) along with MapReduce [9] and Giraph [3] in order to process massive data with better fault-tolerance and scalability.

The rest of the paper is organized as follows. In section II, we present the related work. Section III presents our scalable community detection algorithm. Experimental results are discussed in section IV and the paper concludes with section V.

II. Related Work

Varamesh et al [10] proposed Distributed Clique Percolation based Community Detection algorithm using MapReduce. Chen et al [5] proposed a parallel and distributed algorithm for the modularity based Girvan-Newman's community detection algorithm using MapReduce. Their approach has four MapReduce jobs in chain. MapReduce Job-1 and 2 calculates the modularity between every node pairs and finds the node pairs with highest modularity. Job-3 and 4 merges the node pair with the highest modularity and updates the edge weights of affected edges due to merging.

Community Detection by computing edge betweenness is described by Moon et al [11]. They proposed a Shortest Path Betweenness MapReduce Algorithm (SPB-MRA) which goes through four stages. Stage-1 calculates all the pairs with shortest path. This stage executes multiple times, and rest of the stages execute once during every iteration. Stage-2 finds edge betweenness of every pair of nodes. Stage-3 selects edges to be removed and stage-4 removes the selected edges. All four stages are implemented in Map Reduces and run in parallel.

Fast modularity optimization algorithm by Blondel et al [12] is an iterative multi-step method of identifying partitions and modularizing them so as to attain smaller weighted network with Newman Girvan modularity method. This method provides compromise between accuracy of the modularity maximum technique and computational complexity, essentially linear in the number of links of the graph.

Local resolution-limit-free Potts model by Ronhovde and Nussinov's approach [13] is based on Potts-like spin model where the spin state represents the membership of the node in a community. By different initial conditions, with the same resolution parameter, similarity of partitions is identified and thus been arrived at relevant scales. Stable/Relevant partitions are identified by peaks in similarity spectrum thus attained. The method being fast has slightly super-linear complexity.

All the above discussed approaches are not scalable and efficient for the data of big data scale mainly because of many IOs. In our proposed approach, we break down Girvan-Newman algorithm in such a way that it fits in Giraph programming model. Further, our approach takes advantage of localization and takes only few IOs during whole computation as Giraph keeps the graph state in memory during the whole computation.

III. SE-CDA Design

In this section, we first discuss the sequential algorithm given by Girvan-Newman to detect communities from a graph and then describe our proposed algorithm.

The steps in Newman algorithm's for community detection are:

1. Calculate modularity of every edge.
2. Edge with the highest modularity is merged into single node.
3. Update affected edge weights due to merging.
4. Repeat steps 1 to 4.

Modularity of edge is given by

$$Q_{ij} = 2\,(a_i * a_j - e_{ij})$$

where, $\boldsymbol{a_i}$ and $\boldsymbol{a_j}$ are the sum of all the edge weights connected to node $\boldsymbol{i}$ and $\boldsymbol{j}$ respectively and $\boldsymbol{e_{ij}}$ is edge weight between node $\boldsymbol{i}$ and $\boldsymbol{j}$. Here, $\boldsymbol{a_i = \Sigma_j\, e_{ij}}$. This is a greedy algorithm.

For given large dataset of graph, our goal is to find community on the basis of the relationships connecting those specific individuals. As Giraph is a vertex centric approach, in the proposed algorithm every step runs at available worker nodes in parallel.

The input data for this approach is in JSON format as given below:

```
[1, 4.3, [ [2, 2.1], [3, 0.7] ] ]
[2, 0.0, [ ] ]
[3, 1.8, [ [1, 0.7] ] ]
```

Giraph takes a directed/undirected graph and a user defined function Compute() as the input. Computation proceeds as a sequence of iterations, called supersteps in Giraph. During a superstep, Compute() function is invoked for every active vertex in parallel. Initially, every vertex is active. Each vertex is described in its own record, like vertex ID, vertex value and list of adjacency nodes with connecting edge weights. Vertex cannot access the values of other vertices. Vertex can send message to other vertices that will be received in the next super step, set/get the value of vertex, set/get edge weights from adjacency list and add/remove vertex from adjacency list. A vertex will de-active itself by voting to halt and keeps inactive until it receives a message. The execution stops if all vertices are inactive and no messages are in transit.

We used utility functions namely *combiners* and *aggregators* for message reduction and global communication. During a superstep, all vertices provide values to the aggregator. In the end of superstep, all these values aggregated by aggregator and available to all the vertices by next superstep.

Our merging scheme requires three parameters namely $\boldsymbol{a_i}$, $\boldsymbol{a_j}$ and $\boldsymbol{e_{ij}}$ to calculate modularity between every node pair. Every vertex has adjacency list with edge weights. Below we describe our procedure along with algorithms.

$SuperStep - 0$: $Calculate\ a_i$

$Input$:
$\langle VertexID, Adjacency\ list \rangle$

$Output$:
$sendMessage(TargetVertexID, \langle VetexID, a_i \rangle)$

$Procedure$:

```
1: $Initialize\ a_i = 0$
2: $for\ each\ edge\ in\ AdjacencyList\ do$
3:    $a_i = a_i + edge.weight;$
4: $end\ for$
5: $SetVertexValue\ (a_i)$
6: $for\ each\ edge\ in\ AdjacencyList\ do$
7:    $if\ edge.TargetVertex\ less\ than\ VertexID\ then$
8:       $sendMessage(edge.TargetVertex, \langle VetexID, a_i \rangle)$
9:    $end\ if$
10: $end\ for$
```

In the superstep-0, first all the vertices compute $\boldsymbol{a_i = \Sigma_j\ e_{ij}}$ (step-3), and then set $\boldsymbol{a_i}$ as their vertex value (step-5). Vertex value retain across barriers. It requires modularity calculation during next iteration. Next, it propagates $\boldsymbol{a_i}$ to its neighbors (steps 6 to 8). Here to avoid an overlapping of computation, it sends $\boldsymbol{a_i}$ to only those target nodes whose VertexID is lesser than its own VertexID.

As noticed, there is a barrier between consecutive supersteps. That is, the message sent in current superstep will be delivered to the destination vertices in the next superstep.

$SuperStep - 1$: $Calculates\ Modularity$

$Input$:
$\langle VertexID, Adjacency\ list \rangle, List\langle VertexID, a_i \rangle$

$Output$:
$Aggregate\langle$i, j, Modularity$\rangle$

$Procedure$:

```
1: $Initialize\ Modularity = 0.$
2: $for\ each\ received\ message\ do$
3:    $a_i = message.a_i$
4:    $a_j = vertex.getValue\ ()$
5:    $Modularity = 2 * (a_i * a_j - e_{ij});$
6:    $Aggregate(MOD\_MAX, \langle i, j, Modularity \rangle)$
7: $end\ for$
```

In the superstep-1, all vertices receive $\boldsymbol{a_i}$ from adjacent vertices. Vertex also has $\boldsymbol{e_{ij}}$ in adjacency list and $\boldsymbol{a_j}$, which is computed and set as vertex value during previous superstep. In step-5, it calculates modularity $\boldsymbol{Q}$ and sends $<$ ***i, j, Q*** $>$ to aggregator to find node pair with highest modularity.

$Aggregator\ (MOD\_MAX)$: *Find "TopList"*

$Input$:
$\langle i, j, Modularity \rangle$

$Output$:
$SendToAll(TopList\langle i, j \rangle)$

$Procedure$:

```
1: $Set\ K\ with\ constant.$
2: $Create\ ArrayList\ TopList;$
3: $Initialize\ toplist\ with\ AggregatedList$
4: $if\ check(i, j, Modularity) == true\ then$
5:    $Add(TopList, \langle i, j, Modularity \rangle)$
6: $end\ if$
7: $Update\ AggregatedList$
```

Aggregator receives $<$ ***i, j, Q*** $>$ pairs from every vertex. In step-1, it initializes $\boldsymbol{K}$ with constant, which indicates the number of pairs in TopList. For standard approach the value of $\boldsymbol{K}$ is ***1***. In our approximation approach, we consider $\boldsymbol{K} > 1$, in order to find top node pairs with highest modularity. The results of aggregator will become available to all vertices in the following superstep-2.

$SuperStep - 2$: Merge node pairs in "TopList"

```
Input:
        ⟨VertexID, AdjacencyList⟩, TopList⟨i, j⟩

Output:
        SendMessage⟨I, ⟨ TargetVertexID, Edge Weight⟩ ⟩

Procedure:

1:  for each item in TopList do
2:    if VertexID == j then
3:      for each edge in AdjacencyList do
4:        sendMessage (i, ⟨TargetVertexID, EdgeWeight⟩)
5:      end for
6:      voteToHalt ();
7:    else if node j in AdjacencyList then
8:      removeEdges (j);
9:    end if
10: end for
```

Superstep-2 receives the node pairs with highest modularity. It reads one by one node pair from TopList and merges this node pair into a single node. Then, it makes other nodes in node pair inactive permanently (step-6) and also removes all the edges connected to that node (step-8). These merging of nodes affect the weight edges of nearby edges. Finally, Superstep-2 sends update to affected nodes and makes it available to vertices during next Superstep.

$SuperStep - 3$: Update Edge Weights − 1

```
Input:
⟨ VertexID, Adjacency list⟩ ,
                    List⟨ TargetVertexID, EdgeWeight⟩

Output:
SendMessage⟨ TargetVertex, ⟨ VertexID, EdgeWeight⟩⟩
Procedure:

1:  for each received message do
2:    Initialize weight = 0
3:    if getEdgeValue (TargetVertexID) != null then
4:      weight = getEdgeValue (TargetVertexID)
5:      weight = weight + message.EdgeWeight
6:      setEdgeValue (TargetVertexID, weight)
7:      sendMessage (TargetVertexID, weight)
8:    else
9:      addEdge (TargetVertexID, message.EdgeWeight)
10:     sendMessage (TargetVertexID, weight)
11:   end if
12: end for
```

In the superstep-3, vertices receive list of ***<TargetVertexID, EdgeWeight >***. It updates the weights of affected edges (steps 3-11). As input graph is undirected graph, it needs to update the weight at other end. So in step-7 and 10, updated weights are sent to vertices with ***TargetVertexID***, which will be available at vertices during next superstep.

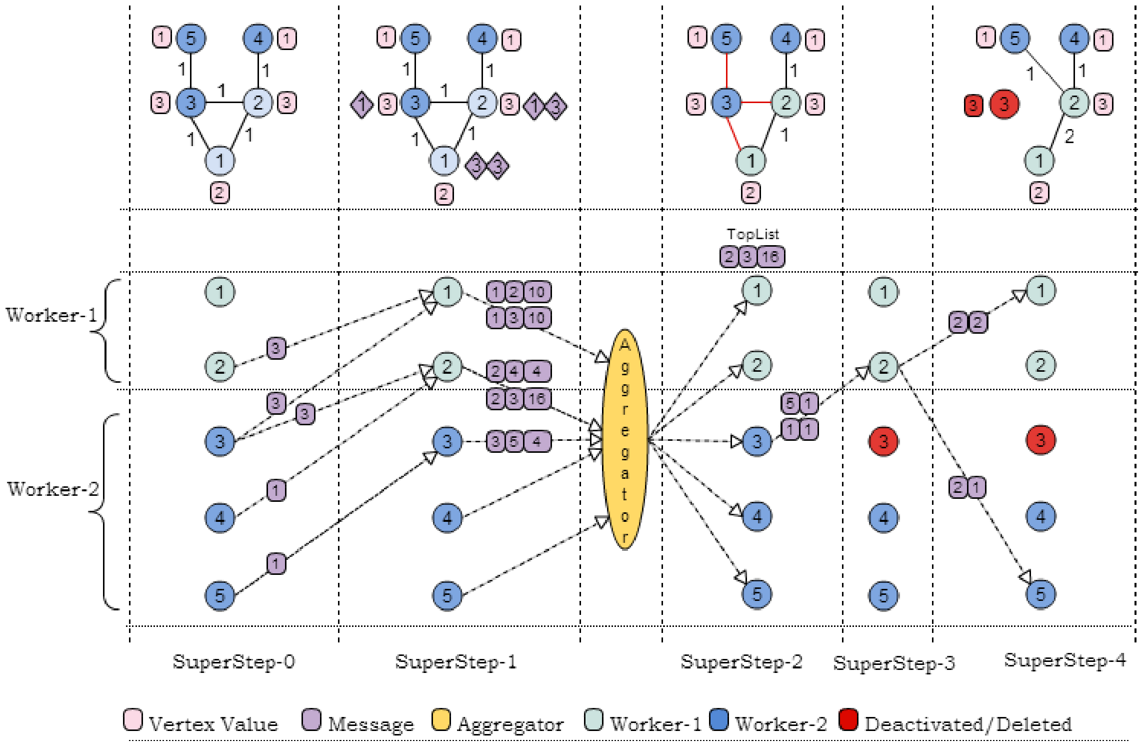

Fig. 1. Data flow of proposed approach

```
SuperStep − 4: Update Edge Weight − 2

Input:
        ⟨VertexID, Adjacency list⟩,
          List⟨TargetVertexID, EdgeWeight⟩

Procedure:

for each received message do
  if getEdgeValue (i) != null then
    setEdgeValue (i, message.EdgeWeight)
  else
    vertex.addEdge (i, message.EdgeWeight)
end for
```

The updated List <***TargetVertexID, EdgeWeight*** > during superstep-3 is input to the superstep-4. If ***TargetVertexID*** exists in adjacency list then it will update the weight, otherwise it adds new edge into adjacency list. Then, the procedure starts execution again from superstep-0.

Here, described algorithm is direct translation of Girvan-Newman algorithm which runs in parallel in a distributed environment. It takes gigantic processing time to generate complete hierarchical clustering tree, since every iteration of algorithm merges only one node pair. To address this issue, we proposed an approximation algorithm for the same. As we know, in case of detecting community structure from large graph, sequence of merging nodes is not important. Instead of merging single node pair during every iteration, our approximation approach merges top ***K*** nodes with highest modularity during every iteration so as to improve the performance significantly.

Figure 1 shows the data flow for our approach. As shown in the figure, input graph splits and load into available workers. Here, we took five vertices and two workers. So vertices 1 and 2 load into worker-1, and vertices 3, 4 and 5 load into worker-2.

As stated earlier, we require three parameters $\boldsymbol{a_i}$, $\boldsymbol{a_j}$ and $\boldsymbol{e_{ij}}$ to calculate modularity. Every vertex has adjacency list with edge weights of all its connected nodes.

First, superstep-0 calculates $\boldsymbol{a_i = \Sigma_j e_{ij}}$ and set as vertex value as shown ▢. Now to calculate modularity vertices require $\boldsymbol{a_j}$. So it sends $\boldsymbol{a_i}$ to all neighbor nodes whose node ID lesser than its own node ID. All these values are available to vertices during next superstep. So $\boldsymbol{a_i}$, $\boldsymbol{a_j}$ and $\boldsymbol{e_{ij}}$ available at superstep-1. So it calculates modularity and send ***<i, j, Modularity>*** to aggregator, which is shown as ▢▢▢. Aggregator finds top-k pairs with highest modularity and distribute to every vertex. So list with top-***K*** highest modularity are available at every vertex during next superstep. Here, in figure 1, we set ***K*** as 1 for simplicity. As shown, 2 3 16 is pair in TopList with highest modularity. It saves the selected pairs into file as output.

Now, superstep-2 merges the node pairs in TopList. So, as shown in figure, it merges 2 and 3 and sends adjacency list of 3 to node 2 and deactivate the node-3 and its connected edges, which highlighted with red color.

Superstep 3 and 4 updates the edge weights of affected edges. So, here, node pair 1-3 and 1-2 merges and final weight of 1-2 node pair updates to 2. After superstep-4, execution starts again from superstep-0.

IV. Experiments

A. Dataset & Environment

Proposed algorithm experimented on system with 64-core 2.13 GHz CPU (64-bit architecture, 128 Hyper threading), 512 GB RAM, 3 TB hard drive space. We have conducted experiments on sample of 1,189K unique users collected by Breadth-First-Search traversal on Facebook, released by [14].

B. Standard Approach

The performance of different datasets is analyzed with above mentioned environment. Datasets are tested for sizes up to 2 million vertices to demonstrate the performance of the algorithm. Figure 2 shows the performance graph of Hadoop versus Giraph.

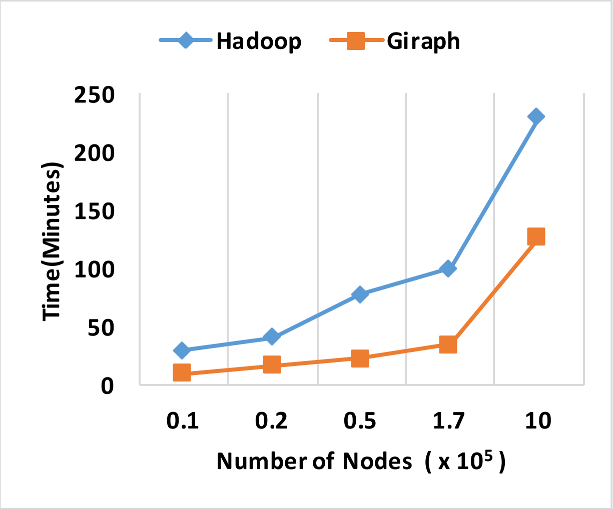

Fig 2. Performance Graph Hadoop vs Giraph

C. Approximation Approach

The performance of approximation algorithm on different number of node pairs merged during every iteration is also experimented. Figure 2 shows the performance comparison of Standard and Approximation approaches with ***K*** = 6 and ***K*** = 10. The results illustrate that approximation approach gives better performance over standard approach for the same dataset.

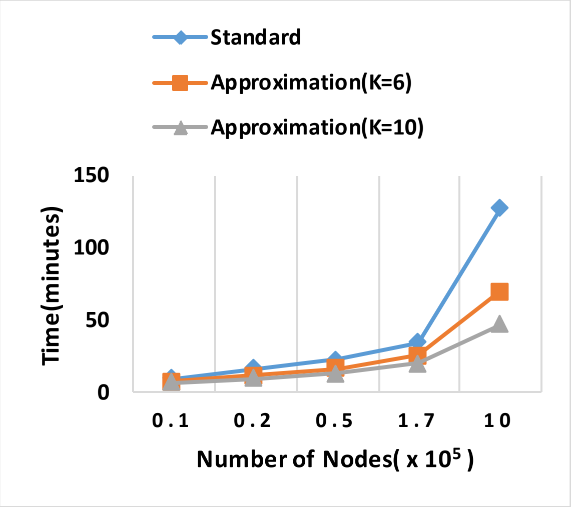

Fig. 3. Performance Graph: Standard vs Approximation Approaches

V. Conclusion

In this paper, we addressed discovery of community structures from a given large size graph. We adopted Girvan–Newman's modularity based hierarchical community detection algorithm in bottom up approach and extended it to distributed environment. We also proposed an approximation algorithm to detect the community structures. We performed experiments with a dataset with varying sizes. Our results show the viability of the proposed approach.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135-146*, 2010.

[2] L. G. Valiant, "A bridging model for parallel computation," in *Communications of the ACM*, vol. 33 (8), pp. 103-111, 1990.

[3] Ching, "Giraph: Large-scale graph processing infrastruction on Hadoop," in *Proceedings of Hadoop Summit*, Santa Clara, USA, 2011.

[4] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim and S. Maeng, "HAMA: An efficient matrix computation with the mapreduce framework," In Proce. of *2010 IEEE Second International Conference Cloud Computing Technology and Science (CloudCom)*, pp. 721-726, 2010.

[5] Y. Chen, Yingying, C. Huang and K. Zhai, "Scalable Community Detection Algorithm with MapReduce," *Commun. of ACM, vol. 53*, pp. 359-366, 2009.

[6] J. Han, M. Kamber and J. Pei, Data mining: concepts and techniques. Morgan Kaufmann, 2006.

[7] Derényi, Imre, G. Palla and T. Vicsek, "Clique percolation in random networks," Physical review letters, vo. 94 (16), pp. 160-202, 2005.

[8] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Physical review, E 69*, 066133, 2004.

[9] Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Communications of the ACM*, vol. 51 (1), pp. 107-113, 2008.

[10] Varamesh, M. K. Akbari, M. Fereiduni, S. Sharifian and A. Bagheri, "Distributed Clique Percolation based community detection on social networks using MapReduce," 5th Conference on *Information and Knowledge Technology (IKT) , IEEE Explore*, pp. 478 - 483, 2013.

[11] S. Moon, J.-G. Lee and M. Kang, "Scalable Community Detection from Networks by Computing Edge Betweenness on MapReduce," In *Proc. 2014 Int'l Conf. on Big Data and Smart Computing (BigComp), Bangkok, Thailand, pp. 145-148* ,2014.

[12] D. Blondel, J.-L. Guillaume, R. Lambiotte and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, P 10008, 2008.

[13] P. Ronhovde and Z. Nussinov, "Local resolution-limit-free Potts model for community detection", Physical Review, E 81, 046114, 2010.

[14] M. Gjoka, M. Kurant, C. T. Butts and A. Markopoulou, "Walking in facebook: A case study of unbiased sampling of osns," In Proc. of *INFOCOM 2010, IEEE* Explore, pp. 1-9, 2010.