Scalable community detection in massive social networks using MapReduce

J. Shi
W. Xue
W. Wang
Y. Zhang
B. Yang
J. Li

In this paper, we present a community-detection solution for massive-scale social networks using MapReduce, a parallel programming framework. We use a similarity metric to model the community probability, and the model is designed to be parallelizable and scalable in the MapReduce framework. More importantly, we propose a set of degree-based preprocessing and postprocessing techniques named DEPOLD (DElayed Processing of Large Degree nodes) that significantly improve both the community-detection accuracy and performance. With DEPOLD, delaying analysis of 1% of high-degree nodes to the postprocessing stage reduces both processing time and storage space by one order of magnitude. DEPOLD can be applied to other graph-clustering problems. Furthermore, we design and implement two similarity calculation algorithms using MapReduce with different computation and communication characteristics in order to adapt to various system configurations. Finally, we conduct experiments with publicly available datasets. Our evaluation demonstrates the effectiveness, efficiency, and scalability of the proposed solution.

Introduction

Networks (or graphs) are used to model complex real-world systems, such as social networks, biological interaction networks, coauthor networks, and linkage graphs. Unlike random networks, real-world networks have a hidden feature of community structure, i.e., the organization of vertices in clusters, with many edges joining vertices of the same cluster and comparatively few edges joining vertices of different cluster [1]. The community-detection problem has been widely addressed in the literature of sociology, biology, and computer science [1, 2]. Detecting communities has concrete applications. For example, mobile services providers can advertise specific offerings to users belonging to the same community. Online social networks (OSNs) can use community structure to recommend new friends or services to its users. The detected communities from OSNs are also useful to sociologists.

The emergence of massive social networks makes the community-detection problem challenging. The Checkfacebook site reports that there were 868 million active users in August 2012 [3]. China Mobile reached 670 million users by the end of May 2012. Traditional community-detection approaches have difficulty handling such massive-scale social networks.

First, traditional community-detection models are often designed for small-size networks with less than 10,000 vertices. For example, the well-known modularity-based approach [4, 5] performs well with small-size datasets such as Zachary's Karate Club [6] and America College Football [7]. However, these approaches are not suitable for processing networks with more than millions of nodes, due to accuracy issues.

Second, the efficiency of traditional community-detection algorithms is inadequate for processing very large social networks because they are not parallelizable. For example, traditional hierarchical clustering approaches [1] cannot be fit into parallel programming frameworks such as MapReduce [8]. (MapReduce is a programming model and an associated implementation for processing large datasets.) To obtain a reasonable community division, these methods often require $O(m)$ rounds of iterations to add or remove community forming edges (m is the number of edges). If these methods were implemented using

Digital Object Identifier: 10.1147/JRD.2013.2251982

©**Copyright** 2013 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied by any means or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of this paper must be obtained from the Editor.

0018-8646/13/$5.00 © 2013 IBM

MapReduce, we would further have several MapReduce jobs in each round of iteration based on the message passing model [9]. To achieve reasonable execution time and control overhead, practical MapReduce applications should have a low number of iterative jobs, usually less than a thousand jobs. PageRank [10] is a typical example.

The above issues motivate us to design, implement, and evaluate an efficient community-detection solution for large-scale social networks using MapReduce. We choose MapReduce because of its parallel and fault-tolerant features for the huge scale of many social networks. We propose a set of preprocessing and postprocessing algorithms named DEPOLD (DElayed Processing of Large Degree nodes) to effectively process vertices of very large degrees. Large-degree vertices commonly exist in social networks with a power-law distribution of node degrees [11, 12]. Applying DEPOLD to a social network from a publicly available OSN dataset (three million vertices) [12, 13] shows significant efficiency improvement compared with the method without applying DEPOLD. Our solution lowers both processing time and storage space by one order of magnitude of reduction. In addition, we also confirm our algorithms reduce the convergence time of community detection (usually within ten MapReduce iterations).

In this paper, we provide a complete community-detection solution for large social networks using MapReduce framework in a systematic fashion. The key contributions of this paper are threefold. First, the contribution is the *DEPOLD mechanism.* DEPOLD filters vertices with a large degree before applying core detection algorithms and processes the filtered nodes upon the output from the core-detection algorithms. DEPOLD can also be applied to other graph-clustering problems [14].

The second contribution involves a *MapReduce-oriented community-detection model and algorithms.* In particular, we designed and implemented the community-detection framework in MapReduce. The core-processing algorithms with an improved similarity metric have a good locality characteristic. (Locality means that all the information needed by the calculation can be exchanged within one or a few hops in a graph.) Each iterative step (including similarity calculation and topology updating) can be performed by two MapReduce jobs. Such algorithms converge fast while achieving better community divisions than those without applying DEPOLD.

The third contribution involves the *evaluation of system performance and consistency.* We conducted evaluations with real-world social network datasets from an OSN [12, 13] and a bulletin board [15]. As suggested, the results demonstrate the efficiency of our solution. We show that the control parameters of our detection model are easy to specify. The speedup trend indicates our implementation scales very well with increasing number of machines in a Hadoop** cluster.

Related work

The general problem of community detection has been extensively studied [1]. The early stage of research is in the form of graph partitioning. Authors in [16] proposed a benefit function to differentiate edges "within" and "between" communities. The algorithm starts from an initial partition with a specific number of groups and swaps the subsets of vertices in these groups to maximize the benefit function. The problem is that both the number and the size of communities must be specified as input parameters, which is impractical for any real community-detection applications in large social networks.

Hierarchical clustering algorithms include divisive and agglomerative approaches. The divisive approach iteratively computes "inter-communitiness" scores for each link and removes the link with the worst score [4]. The worst-case time complexity of this algorithm is $O(m^2 n)$ on a network with m edges and n vertices. It cannot be used to process more than 10^4 nodes. Improvements have been proposed for divisive algorithms [17, 18]. Among them, the fastest one [18] is still limited to processing 10^5 nodes.

Modularity has been proposed to measure the accuracy of community divisions [5]. Obtaining the optimal modularity value has been recently proved to be NP-complete [19]. Various approximating approaches have been proposed [1, 20, 21]. Unfortunately, the basic model of modularity is not applicable to large social networks. It has been shown that a modularity maximum does not necessarily mean the existence of community structure in large social networks [1]. In addition, existing solutions that optimize modularity cannot be parallelized to meet the scale of large social networks.

There are other alternative approaches for community structure mining, such as random walk methods [22]. However, the complexity of the most efficient random work approach so far is $O(n^2 \cdot \log n)$ on a network with n vertices [22]. Similar to a modularity-based approach, it is also difficult to parallelize the algorithm.

The locality characteristic of the community-detection algorithm usually determines whether it can be parallelized. Essentially, if the computing cannot be completed locally (e.g., within two hops), it is difficult to be executed simultaneously by parallel workers. The traditional community-detection algorithms mentioned above require global knowledge instead of local information, so they are difficult to parallelize.

A parallel approach to detect communities with propinquity dynamics is proposed in [23]. We improve the previous proposal to make it converge significantly faster with normalized metrics. More importantly, we propose our DEPOLD mechanism in this work, which achieves more than one order of magnitude of both execution time and storage space reduction. With this complete solution, we are able to design and implement the solution with MapReduce platforms and evaluate it with realistic large social data sets.

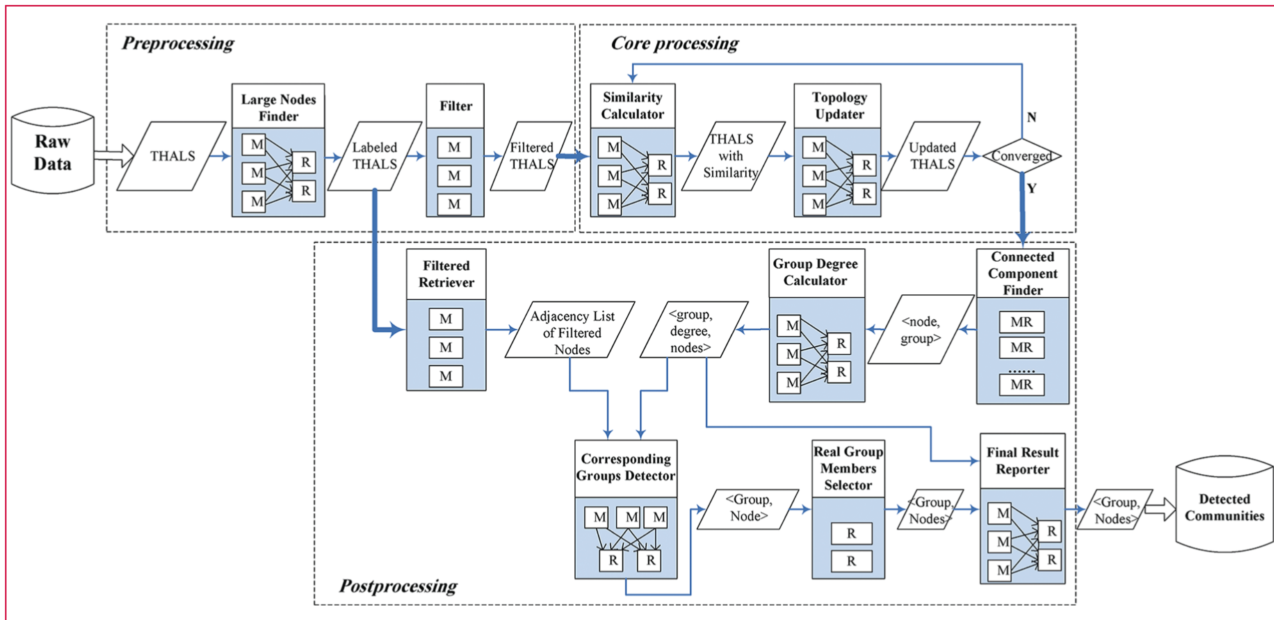


Figure 1

Framework overview and community-detection data flow in MapReduce. (THALS: two-hop adjacency list with similarity.)

Other authors have proposed an algorithm that recursively fingerprints massive graphs to extract dense sub-graphs [24]. We are different in that we specially treat high-degree nodes and detect communities in parallel.

Finally, Large Adjacency List Partitioning (LALP) partitions adjacency lists of high-degree vertices across nodes to reduce communication overhead [25]. However, because LALP does not change the processing order of high-degree vertices, its computation overhead is still very high.

Framework overview

In this section, we present the framework of the proposed community-detection solution. As shown in **Figure 1**, the solution consists of three stages: preprocessing, core processing, and postprocessing. M and R represent Mapper and Reducer, respectively. The similarity metric model is located in the core-processing module, with components of two iterative MapReduce jobs: *Similarity Calculator* (*S_Calculator*) and *Topology Updater* (*T_Updater*). The DEPOLD mechanism is implemented in the preprocessing and postprocessing stages. The similarity model designed in this paper applies the concept of computation locality. For parallel graph analysis algorithms with a message-passing framework [9], computation locality is crucial to reduce the number of iterations.

The key algorithm of the core-processing stage iteratively calculates the similarity metric of two linked vertices. The link will be removed if the similarity value is less than a specified threshold γ. When the change of the topology is less than a threshold φ, the algorithm has converged. Then extracting community division can be trivially done by discovering the *Weakly Connected Component* (WCC) [26]. Note that two vertices that are not directly connected will be grouped into the same community if they belong to the same *WCC* in the graph after the core-processing stage converges.

In the preprocessing stage, the nodes with a degree larger than the predefined threshold θ are labeled. The labeled nodes and their structure are stored in HDFS (Hadoop Distributed File System) for future usage by DEPOLD in the postprocessing phase. In the postprocessing phase, we first retrieve community structure detected in low-degree nodes, and then merge back the filtered high-degree nodes. For the merging, two factors are computed to determine whether a filtered node should be included in this detected community: 1) the degree the filtered node contributes to that community and 2) the average contributed degree of the detected community members. The *Real Group Member Selector* job uses these two values to determine whether the filtered node should belong to the detected community. Because filtering out high-degree nodes reduces the amount of inter-community noise, the accuracy and efficiency of the community-detection solution are improved by DEPOLD.

Community-detection model

Similarity metric calculation

We use a undirected non-weighted graph without duplicate edges or self-loops to model social networks. The graph

is represented by $G(V, E)$, where V is the vertex set and E is the edge set. We define the similarity metric as the probability that two connected vertices belong to the same community. The link in the social graph with low similarity value will be removed.

The numerator of similarity metric formulation consists of three parts. The first part is the link between the two nodes (e.g., the value is 1 if two nodes are connected). The second part of the similarity between v_i and v_j is the number of their common neighbors. We use $C(v_i, v_j)$ to denote this part. The set of the neighbors of node v_i is represented as $\pi(v_i)$. Thus, we have $C(v_i, v_j) = |\pi(v_i) \cap \pi(v_j)|$. The third part is the number of conjugate edges from their common neighbors (i.e., edges among their common neighbors). We have $\eta(v_i, v_j) = |E(\pi(v_i)) \cap E(\pi(v_j))|$.

It is obvious that the above similarity results depend heavily on the density of the social graph [11]. Choosing the right threshold for similarity probability for link removal becomes a trial-and-error task without value normalization. Thus, we define the similarity metric as a normalization value instead of a graph-dependent value.

Theoretically, the normalization factor $F(v_i, v_j)$ should be $F(v_i, v_j) = 1 + \min\{D(v_i) - 1, D(v_j) - 1\} + C(v_i, v_j) \cdot (C(v_i, v_j) - 1)$, where $D(v_i)$ represents the node degree of v_i. However, we find the simple factor $F(v_i, v_j) = D(v_i) + D(v_j)$ robust and adaptive to deal with various social networks in practice.

Thus, we have the similarity metric definition:

$$S(v_i, v_j) = \begin{cases} \frac{1 + C(v_i, v_j) + \eta(v_i, v_j)}{F(v_i, v_j)} & : \quad \text{if } \left|E(v_i, v_j)\right| = 1 \\ 0 & : \quad \text{otherwise.} \end{cases} \tag{1}$$

Considering a complete graph with four nodes $\{v_1, v_2, v_3, v_4\}$, we have $S(v_1, v_2) = (1 + 2 + 1)/(3 + 3) = 0.67$.

THALS data structure

We design a node data structure THALS (two-hop adjacency list with similarity) specifically for the purpose of our MapReduce-oriented solution. This node structure is used as the MapReduce input and output data format in parallel core-processing algorithms. The essential idea is to trade storage space for better parallelization, which is reasonable for computing intensive graph algorithms such as the community detection proposed in this paper. The first field of THALS is vertex ID. The second field of THALS is the adjacent list with similarity values, which includes neighbor vertex ID and the corresponding similarity value with that neighbor. The third field of THALS is the two-hop adjacent list. We store neighbors of a neighbor (i.e., two-hop neighbors) in this field using a hash table. The information is needed for parallel similarity calculation. The fourth field of THALS is a label, designed as extendable $\langle Name, Value \rangle$ pairs. It can be used to transmit any information from Mapper to Reducer. When we apply DEPOLD, the storage complexity is reduced from $O(n \cdot D^2)$ to $O(n)$ on a network with n vertices and average degree D, because the largest degree is bounded by the constant threshold θ.

Core processing

In this section, we describe the similarity calculating job *S_Calculator* and the topology updating job *T_Updator*, represented at upper right in Figure 1. In each round, we use a MapReduce job, *S_Calculator*, to compute the similarity metric, and then we execute another MapReduce job, *T_Updator*, to remove edges according to the computed similarity value. The core-processing algorithm is converged if the number of removed edges is less than a threshold value ϕ.

Similarity calculator

We implement two versions of *S_Calculator* because there is a tradeoff between computation complexity and shuffle overhead in MapReduce. The first implementation employs only Mapper to calculate the similarity value for each pair of linked nodes. We refer to it as CoMPuting Intensive (CMPI) *S_Calculator*. The second implementation employs both Mapper and Reducer to calculate the similarity value indirectly. Mapper emits intermediate messages to Reducer, which leads to disk and network input/output (I/O) overhead. We refer to this as I/O Intensive (IOI) *S_Calculator*. Note that the actual bound of a workload depends on the input data and hardware configurations of the cluster. We simply use CMPI and IOI to denote the two algorithms based on design choices.

Algorithm 1 illustrates the CMPI *S_Calculator*. Both input and output keys of Mapper are node ID. The input and output values of Mapper are THALS nodes. The information stored in THALS enables local calculation of similarity in parallel. We compute the similarity value for each neighbor of a node. $L(n)$ is a neighbor set of n. $T(n)$ is the two-hop neighbor set of n. $M(n, l)$ is the two-hop neighbor set of n with the intermediate node being l. $|L(n)|$ represents the number of members of set $L(n)$. Note that $L(n)$, $T(n)$, and $M(n, l)$ can be retrieved from the value v of n.

Algorithms 2 and **3** show the IOI *S_Calculator*. The calculation is implicitly split into the processing of individual nodes and edges. Considering a complete graph with four nodes $\{v_1, v_2, v_3, v_4\}$, node v_1 is the common neighbor of $\{v_2, v_3\}$, $\{v_3, v_4\}$, and $\{v_2, v_4\}$. Thus, the node contributes 1 unit to the number of common neighbors of these pairs. Likewise, $E(v_1, v_2)$ represents the conjugate edges of $\{v_3, v_4\}$. The edge contributes 1 unit to the number of conjugate edges of the pair. In fact, a node contributes 1 unit to all pairs of its neighbors as the common neighbor. An edge contributes 1 unit to all pairs of common neighbors of the two vertices of the edge as a conjugate edge. We determine the contribution units in Mapper and emit a notification message to Reducer for each unit. Finally, Reducer groups

Algorithm 1 Mapper of CMPI *S_Calculator*.

```
1: Function: MAP (Nid n, Node v)
2: for each l ∈ L(n) do
3:    p ← 1; d ← |L(n)|
4:    if l ∈ T(n) then
5:       C(l) ← M(n,l) ∩ L
6:       q ← 0; p ← |C(l)|; d ← d + |M(n,l)|
7:       for each c ∈ C(l) do
8:          for each k ∈ M(n,c) do
9:             if k ∈ C(l) then
10:               q ← q + 1
11:         p ← p + q/2
12:   else
13:      d ← d + 1
14:   set similarity value s ← p/d for l to v
15: EMIT(n, v)
```

Algorithm 2 Mapper of IOI *S_Calculator*.

```
1: Function: MAP (Nid n, Node v)
2: EMIT (n, v)
3: if |L(N)| ≥ 2 then
4:    for each {v_i, v_j} pair ∈ L(n) do
5:       if ∃E(v_i, v_j) then
6:          set f_p → src to v_j
7:          set f_p → val to 2
8:          EMIT (v_i, f_p)
9: for each l ∈ L(n) then
10:   if l ∈ T(n) then
11:      C(l) ← M(n,l) ∩ L(n)
12:      if C(l) ≥ 2 then
13:         for each {v_i, v_j} pair ∈ C(l) do
14:            if ∃E(v_i, v_j) then
15:               set f_p → src to v_j
16:               set f_p → val to 1
17:               EMIT (v_i, f_p)
18: for each l ∈ L(n) do
19:   set f_d → src to n
20:   set f_p → val to |L(n)|
21:   EMIT (l, f_d)
```

Algorithm 3 Reducer of IOI *S_Calculator*.

```
1: Function: REDUCE (Nid n, List<Node> V)
2: for each v ∈ V do
3:    parse F_p and F_d for v
4: for each l ∈ L(n) do
5:    M_p.put(l,2)
6: for each f_p ∈ F_p do
7:    if M_p contains key f_p → src then
8:       p ← M_p.get(f_p → src) + f_p → val
9:       M_p.put(f_p → src, p)
10: for each f_d ∈ F_d do
11:   M_d.put(f_d → src, f_d ← val)
12: for each l ∈ L(n) do
13:   s ← (M_p.get(l)/2) / (|L(n)| + M_d.get(l))
14:   set similarity value s for l to v'
15: EMIT (n, v')
```

and calculates the similarity based on received notification messages.

Algorithm 2 illustrates the Mapper of IOI *S_Calculator*, where the object type of *Node* is THALS. First, Mapper emits the node instance to Reducer for future usage. Second, Mapper emits the contribution of the node to each pair of its neighbors. The type of the notifier instance f_p is THALS. We use the *label* field (the fourth fields of THALS) to carry the payload of the message. $f_p \rightarrow src$ indicates from where the units are contributed. $f_p \rightarrow val$ indicates the contributed value that should be counted in Reducer. Third, Mapper emits the contribution of the edge as conjugate edges via f_p. In this phase, Mapper finds common neighbors for each neighbor. Finally, Mapper emits the degree of the node (via f_d) to each neighbor in Reducer. The notification message of degree is also carried by the *label* field of THALS.

Algorithm 3 illustrates the Reducer of IOI *S_Calculator*. V denotes the value list corresponding to key n received by the Reducer. F_p are the set of notification messages from Mapper with the similarity contribution information. $F_p \rightarrow src$ denotes the source vertex that sends the notification message. $F_p \rightarrow val$ denotes the value (i.e., similarity contribution) of the notification message. F_d are the set of notification messages containing the degree information. We first parse and cache the two types of messages and then calculate the similarity value. Hash map M_p and M_d store key-value pairs of source vertex ID with its contributed similarity and degree values, respectively.

Algorithm 4 Mapper of *T_Updater*.

1: **Function**: MAP (Nid *n,* Node *v*)
2: **for** each $l \in | L(n) |$ do
3: **if** $getSimilarity(l) < \gamma$ **then**
4: remove *l* from $| L(n) |$ of *v*
5: $RCounter++$
6: EMIT(*n*,*v*)
7: **if** $| L(n) | > 1$ **then**
8: set one hop id $f_t \rightarrow oid$ as *n*
9: set two hop neighbors $f_t \rightarrow TN$ as $L(n)$
10: **for** each $l \in L(n)$ **do**
11: set source vertex $f_t \rightarrow src$ as *l*
12: EMIT (l, f_t)

Algorithm 5 Reducer of *T_Updater*.

1: **Function**: REDUCE (Nid *n*, List<Node> *V*)
2: **for** each $v \in V$ **do**
3: set id of V_{out} as *n*
4: **if** *v* is labeled as f_t **then**
5: Remove *n* from $f_t \rightarrow TN$
6: Set two hop sets $\{f_t \rightarrow src, f_t \rightarrow TN\}$ in V_{out}
7: else
8: get $L(n)$ from *v*
9: add $L(n)$ as neighbors in V_{out}
10: **if** $| L(n) | > 0$ **then**
11: EMIT(*n*, V_{out})

Topology updater

T_Updater is implemented by a round of the MapReduce job. **Algorithm 4** illustrates the Mapper of *T_Updater*. Mapper first determines whether the edge should be removed on the basis of the latest similarity value and removing threshold γ. After removing the corresponding neighbors from the adjacent list, Mapper increases by 1 in the remove counter *RCounter* using the MapReduce's counter mechanism. Second, Mapper emits the neighbor information to each neighbor of the node in Reducer to update its two-hop structure. f_t denotes the notifier instance containing the neighbor information. **Algorithm 5** illustrates the Reducer of *T_Updater*. Reducer receives the topology updating message from Mapper and updates both neighbors and the two-hop neighbors list accordingly. The main loop of *T_Update* and *S_Calculator* is responsible for determining whether the algorithm converges based on *RCounter* and the threshold ϕ.

DEPOLD: Preprocessing and postprocessing

In this section, we present our DEPOLD algorithm targeted for processing massive social networks.

Preprocessing: Why filtering of nodes?

The degree of distribution of social networks follows a power law. The most notable characteristic is the relative commonness of vertices with a degree that greatly exceeds the average. Moving the computation of high degree nodes to the postprocessing stage improves both efficiency and accuracy.

We find that nodes of large degree cause the performance bottleneck when we directly apply the similarity-based algorithm to perform community detection on large social networks (i.e., without DEPOLD). For the CMPI algorithm, processing nodes with large degree dominate the CPU consumption. For the IOI algorithm, nodes with large degree cause a large disk I/O overhead. Storing THALS consumes a significant amount of disk for these high-degree nodes. In addition, nodes of large degree also increase the network I/O consumption.

Filtering high-degree nodes can also significantly improve the detection accuracy. We find that nodes with a large degree do not reside in one community. They often lay across several communities as overlapping nodes. Such overlapping phenomena are common in social networks [27]. These nodes cause a substantial amount of noise in community detection and sometimes even result in detection failure. The DEPOLD mechanism effectively addresses the above concerns.

Implementation details of DEPOLD

We specify the threshold θ as the filtering algorithm input. A node is filtered in the preprocessing phase if its degree is larger than θ. Note that it is easy to extend the input parameter to be a percentage of largest degree nodes rather than an absolute degree value. We store the topology (i.e., adjacency list) of filtered nodes in HDFS for future DEPOLD usage.

After the core-processing stage, we use the MapReduce-oriented WCC implemented in our prior work [15] to retrieve $\langle node, group \rangle$ pairs, shown as *Connected Component Finder* in Figure 1. Then, we start the postprocessing stage of DEPOLD.

We post-detect the community for filtered nodes utilizing the detected results from the unfiltered ones. Because the adjacency list of each filtered node is stored and the community division of its neighbors in the adjacency list can also be retrieved, we can measure whether the filtered node belongs to the community of its neighbors. If the

Algorithm 6 Reducer of Real Group Members Selector.

1: **Function**: REDUCE (Group g , List<Node> V)
2: parse group name g_n from g
3: parse group degree g_d from g
4: **for** each vertex $v \in V$ **do**
5: $c \leftarrow M_n.get(v)$
6: $c{+}{+}$
7: $M_n.put(v,c)$
8: **for** each key $k \in M_p$ **do**
9: **if** $M_p.get(k) > g_d$ **then**
10: EMIT (g_n, k)

number of neighbors of the filtered node belonging to a community (i.e., the degree the filtered node contributes to that community) is larger than that of average contributed degree to the community, we add the filtered node to that community.

The postprocessing stage component in Figure 1 shows the data flow. *Group Degree Calculator* calculates the average contributed degree of members for each detected community. Because the adjacency list is available in the output of *Connected Component Finder*, this calculation is trivial. The output key of *Group Degree Calculator* is the group name with the average degree. The *Corresponding Groups Detector* maps the filtered nodes to potential corresponding groups based on their adjacency list. It emits $\langle group, filtered \rangle$ pairs to the corresponding group for each neighbor. For example, we suppose the filtered node is v_f with unfiltered neighbors v_1, v_2, v_3, v_4, and v_5. Nodes v_1 and v_2 are assigned to g_1, and nodes v_3, v_4, and v_5 are assigned to g_2. Then, the *Corresponding Groups Detector* emits two $\langle g_1, v_f \rangle$ to g_1 and three $\langle g_2, v_f \rangle$ to g_2. For each filtered node, the number of emitted messages to a group indicates the number of its neighbors fall into that group. Thus, we have the *Real Group Members Selector* described in **Algorithm 6** to finally detect the community for filtered nodes. M_n denotes the hash map that stores the connected degree of the node. Each resulted group is a detected community.

Experiments

We conduct experiments [28] on real-world datasets to evaluate the accuracy and efficiency of our solution.

Experimental environments

The Hadoop [29] cluster (version 0.20.2) with ten POWER* 730 Express machines is deployed on two 10-Gbps Ethernets. The ten machines are homogeneous. Each machine has two sockets with six cores each. We set simultaneous multithreading (SMT) to two such that we have 24 hardware threads per machine. Each machine has 64 GB of memory. We have one HomeRun Storage Expansion per machine with twenty-four 600-GB 10-Krpm SAS (Serial Attached SCSI [Small Computer System Interface]) disk storage drives. Thus, we have the cluster with 120 cores, 240 hardware threads, 640 GB of DRAM (dynamic random access memory), and 144-TB of SAS disk storage. We also tested the algorithms on a cluster with eight x86 nodes. We present the results of the ten-node POWER7* cluster because it can have more than 200 concurrent Map/Reduce workers to demonstrate the scalability of our solution.

All of the ten nodes serve as *Task Tracker* and *Data Node*. One master node also serves as *Job Tracker* and *Name Node*. We assign 20 concurrent maximal Map or Reduce tasks (i.e., "task slots") for the master and 24 for the other nodes. Then, both the Map and Reduce task slots capacity is 236. This assignment enables the cluster to utilize all the CPU resources when the workload is bounded by CPU. We set the maximum of 236 Reduce tasks for jobs. Then, the copy and sort phase of Reduce tasks can be paralleled with Map tasks. The block size of HDFS is set to 256 MB. The JVM** (Java Virtual Machine**) heap size of the child task process is set to 900 MB. For IOI *S_Calculator*, we compress Map output to save network I/O. We use Ganglia [30] to monitor the cluster-wide resource utilization.

Datasets

We evaluated our solution on several actual social network datasets such as a publicly available OSN dataset [12, 13], Twitter** [31], an enterprise social network, an online bulletin board system (BBS), etc. Because of the limited space, we only present the results on OSN and BBS data.

The BBS dataset is described in [15]. We choose to present the BBS dataset with limited size for the comparison between the similarity-based algorithm with and without DEPOLD, because we could not reliably obtain the results on the OSN dataset without DEPOLD filtering, at least with our hardware.

The publicly available OSN dataset includes 3,097,166 nodes and 28,377,481 edges [12, 13]. Each line of the raw data includes two user IDs and represents friend relation between these two users IDs. The user IDs are sequential integers (from 1 to 3,097,166) to save storage space. We also remove redundancy in the raw data and generate undirected graphs that have no loops and no more than one edge between any two vertices.

Detection accuracy demonstration

We first demonstrate the accuracy of our detection algorithm on small synthetic datasets with 15 vertices and 24 edges.

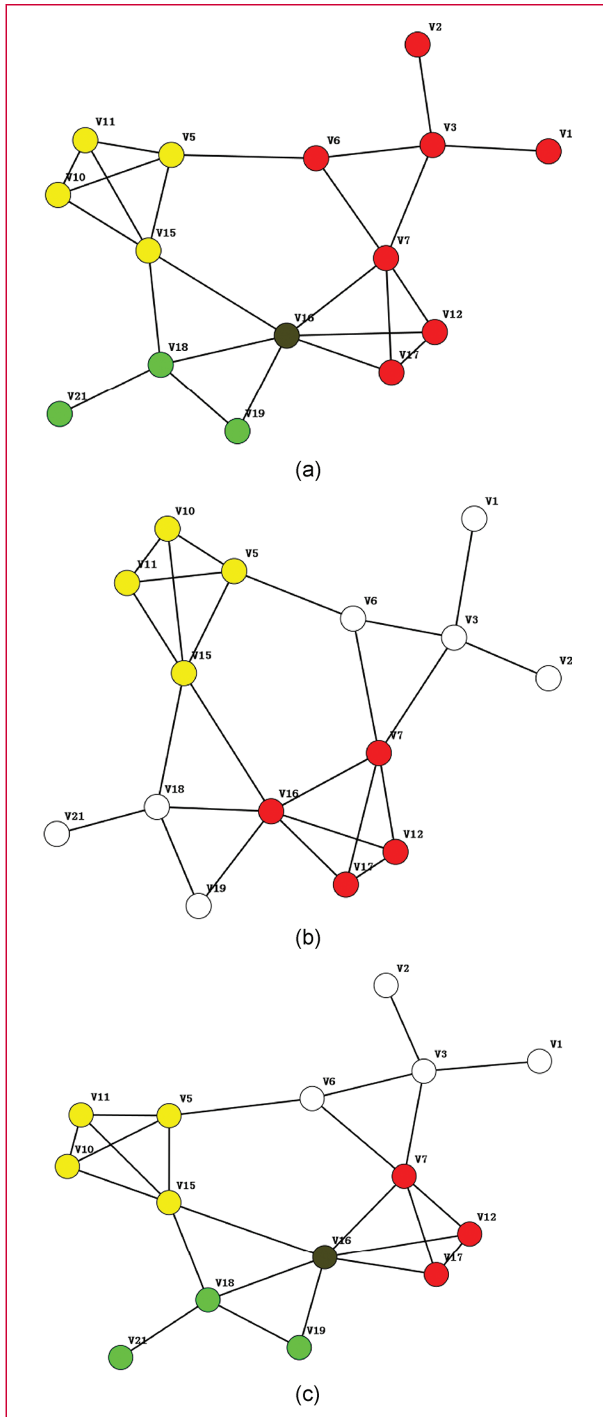


Figure 2

Detection results of the sample: (a) $\gamma = 0.2$; $\theta = 5$; (b) $\gamma = 0.3$; $\theta = 5$; (c) $\gamma = 0.3$; $\theta = 4$.

Figure 2(a) shows the detection result when $\gamma = 0.2$ and $\theta = 5$. Node v_{16} is filtered by DEPOLD during the preprocessing stage. The core-processing stage converges in two rounds. Two edges $E(v_5, v_6)$ and $E(v_{15}, v_{18})$ are removed in the core-processing stage. Node v_{16} is detected to be a member of both group $\{v_{16}, v_{17}, v_2, v_1, v_7, v_6, v_{12}, v_3\}$ and group $\{v_{18}, v_{21}, v_{19}, v_{16}\}$ by DEPOLD during the postprocessing stage. In Figure 2(a), members in the same community are labeled with the same color. The result indicates that the algorithm is able to find inter-community edges and remove them.

Evaluating accuracy of community detection is application-specific for large-scale social networks, as there is no explicit accuracy metric. Usually the evaluation is based on whether the detection result meets the application requirement. To address this problem, we have studied the effect of key parameters of our solution on detection results and consequently recommend applications to tune the provided parameters for desired community-detection effect. Therefore, our method is adaptive to different applications and datasets.

Figure 2(b) shows the detection result when $\gamma = 0.3$ and $\theta = 5$. Here, v_{16} is filtered. The core-processing stage also converges in two rounds. Because the similarity threshold becomes stricter, nine edges are removed. Finally, only two communities are detected, and seven nodes become isolated points. We label them white in the figure, indicating that they belong to no community. This result means that when γ increases, only members of a strongly connected community can be detected.

Figure 2(c) shows the detection result when $\gamma = 0.3$ and $\theta = 4$. Node v_7, v_{15}, and v_{16} are filtered in preprocessing. The core-processing stage converges in the same number of rounds as previous experiments. Compared with Figure 2(b), filtering v_{15} helps to detect the community of $\{v_{18}, v_{21}, v_{19}, v_{16}\}$. The result demonstrates that large-degree nodes could confuse the community structure, and filtering them out for special processing can improve the accuracy of detection results.

Impact of filtering and its threshold

Now, we evaluate the impact of DEPOLD filtering and the filtering threshold on BBS datasets. **Table 1** shows the impact of filtering on CMPI core processing. Since the first CMPI *S_Calculator* is the "bottleneck job," we report its execution time, and HDFS read and write information. Because the job is Map only, there is no intermediate result for Reduce shuffle. The baseline is the algorithm without filtering. The result shows 64.6% of running time reduction on the bottleneck job if we filter the top 1.87% of nodes with the largest degree. It also indicates 92.9% and 92.8% of HDFS read and write reduction, respectively, by the filtering.

Table 2 shows the impact of filtering on IOI core processing. In addition, we report the size of Map outputs (before compression) and Reduce shuffle (compressed). The result shows 81.5% of running time reduction on

Table 1 Impact of filtering on CMPI core processing for the BBS dataset.

θ	*% of filtered nodes*	*Core processing* (s)	*First CMPI S_Calculator* (s)	*First CMPI S_Calculator HDFS_READ* (bytes)	*First CMPI S_Calculator HDFS_WRITE* (bytes)
150	1.8687%	117	29	736,534,956	726,156,287
200	1.2080%	123	32	1,237,251,745	1,220,239,660
250	0.8504%	141	34	1,757,533,725	1,730,978,214
300	0.6194%	142	34	2,321,596,357	2,284,938,717
500	0.2121%	195	44	4,510,689,173	4,430,630,253
1,000	0.0298%	216	59	7,933,898,447	7,786,525,229
3,000	0.0005%	236	71	10,111,306,187	9,883,104,719
∞	0	244	82	10,381,332,957	10,144,927,844

Table 2 Impact of filtering on IOI core processing for the BBS dataset.

θ	*% of filtered nodes*	*Core processing* (s)	*First CMPI S_Calculator* (s)	*First IOI S_Calculator HDFS Read* (bytes)	*First IOI S_Calculator HDFS Write* (bytes)	*First IOI S_Calculator Map Output* (bytes)	*First IOI S_Calculator Reduce Shuffle* (bytes)
150	1.8687%	141	53	736,534,956	726,098,985	1,441,526,399	437,280,605
200	1.2080%	151	58	1,237,251,745	1,220,181,966	2,687,896,849	749,431,784
250	0.8504%	214	80	1,757,533,725	1,730,924,293	4,184,667,406	1,094,761,213
300	0.6194%	215	83	2,321,596,357	2,284,885,793	5,854,477,736	1,466,287,789
500	0.2121%	247	100	4,510,689,173	4,430,577,309	13,209,174,634	2,955,513,014
1,000	0.0298%	309	152	7,933,898,447	7,786,471,953	28,306,779,134	5,562,723,172
3,000	0.0005%	404	239	10,111,306,187	9,883,051,496	40,500,050,883	7,394,933,290
∞	0	452	287	10,381,332,957	10,144,874,568	41,731,934,524	7,604,540,964

the bottleneck job if we filter the top 1.87% of nodes with the largest degree. The reduction of HDFS read and write is the same as that in CMPI core processing. It also indicates 96.5% and 94.2% of Map output size and Reduce shuffle size reduction, respectively, by the filtering.

We also conduct experiments on the eight-node x86 cluster with commodity processors and disks to evaluate the execution time of core-processing stage with and without DEPOLD filtering. More jobs are bounded by system resources such as CPU or disk I/O in the x86 cluster. The result shows 92.3% of total running time reduction if we filter the top 0.85% of nodes with the largest degree. The impact on CMPI-based core-processing is even more impressive: 98.4% of total running time savings with 0.85% of top degree nodes filtered.

For the remainder of this paper, all the experiments are performed with the OSN dataset (including 3 million vertices) using DEPOLD on the ten-node POWER7 cluster. We first evaluate the impact of the filtering threshold θ. Nodes with a degree larger than θ are filtered in the preprocessing stage. We set θ to $\{250, 200, 150, 100\}$ to obtain $\{0.71\%, 1.15\%, 1.92\%, 3.75\%\}$ of filtered nodes, respectively. **Figure 3(a)** shows the Cumulative Distribution Function (CDF) of community size, varying θ. The result indicates that the size of most of detected communities is small. This validates the rule of *Scale-free Networks* [32] such that the clustering coefficient decreases as the node degree increases. It also indicates that filtering threshold θ does not significantly affect the size distribution of detected communities. Figure 3(a) indicates that with more nodes being filtered, more small groups are detected. Because filtering out more high-degree nodes reduces the amount of noise caused by their links, more communities of small size become detectable.

We note that with 1.92% of filtered nodes, the maximal community size is 2,763 for the OSN data. Such large communities can be divided into sub-communities by the micro-clustering heuristic available in [23]. When the percentage of filtered nodes goes beyond 3.57%, the maximal community size becomes smaller, less than 171. Detailed

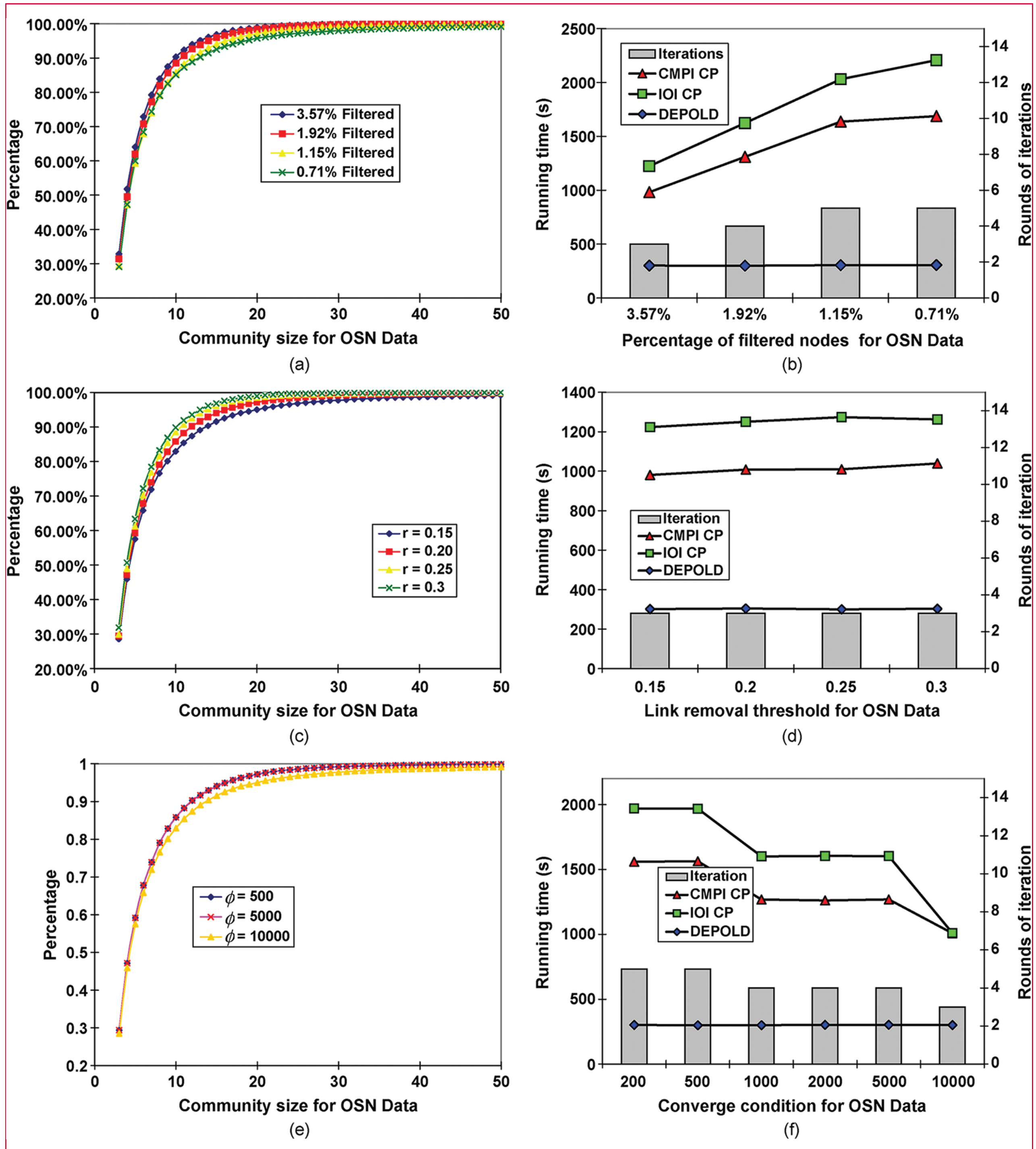


Figure 3

Impact of algorithm parameters. (a) Impact of the filtering threshold θ on detection result for OSN data, $\gamma = 0.2$, $\varphi = 10,000$, $m = 236$, and $r = 236$. (b) Impact of the filtering threshold θ on running time for OSN data, $\gamma = 0.2$, $\varphi = 10,000$, $m = 236$, $r = 236$. (c) Impact of the link removal threshold γ on detection result for OSN data, $\theta = 100$, $\varphi = 10,000$, $m = 236$, and $r = 236$. (d) Impact of the link removal threshold γ on running time for OSN data, $\theta = 100$, $\varphi = 10,000$, $m = 236$, $r = 236$. (e) Impact of the converging condition φ on detection result for OSN data, $\theta = 100$, $\gamma = 0.2$, $m = 236$, and $r = 236$. (f) Impact of the converging condition φ on running time for OSN data, $\theta = 100$, $\gamma = 0.2$, $m = 236$, and $r = 236$.

investigation shows that the large communities previously detected are split into smaller communities, resulting in more smaller communities. This actually achieves the purpose of micro-clustering and obtains similar results.

Figure 3(b) shows the impact of filtering threshold θ on the running time of the core-processing (CP) stage and DEPOLD. The DEPOLD time is the total processing time introduced by its components in preprocessing and postprocessing stages, such as filtering and merging. It does not include the time to extract community structure from WCC because it is necessary with or without DEPOLD. The result indicates that the round of iteration is reduced from 5 to 3 when the percentage of filtered nodes increases from 0.71% to 3.57%. Overall, the number of iterations remains low in all cases. The running time is also significantly reduced because the community structure becomes clearer when a larger percentage of nodes is filtered. We also see that DEPOLD does not consume much running time compared with the core-processing stage.

Impact of link removal threshold

We evaluate the impact of edge-removal threshold γ used in the core-processing stage. We set γ to $\{0.15, 0.2, 0.25, 0.3\}$. **Figure 3(c)** shows the CDF of community size, varying γ. The result indicates that the threshold does not significantly affect the degree distribution of detected communities. It indicates that when γ increases, more small-size communities are detected. We also observe that lower γ value leads to larger detected groups. When $\gamma = 0.15$, the maximal detected group has 2,572 members. That size becomes 62 when we change γ to 0.3. That is because larger γ means more strict requirements in community intra-connectivity. **Figure 3(d)** shows the impact of removal threshold γ on the running time of core-processing and DEPOLD. The result shows that this parameter does not distinctly affect the running time or iterative rounds.

Impact of converging condition

Now, we evaluate the impact of the converging condition φ. We still present the CDF of community size to study the result of detection. The list of values we used are $\{200, 500, 1{,}000, 2{,}000, 5{,}000, 10{,}000\}$. **Figure 3(e)** shows the CDF of community size, varying φ. The result indicates that the threshold does not significantly affect the degree distribution of detected communities. **Figure 3(f)** shows the impact of the converging condition φ on the running time of the core-processing stage and DEPOLD. The result shows that when φ increases, fewer rounds of iteration are performed and the running time decreases. Therefore, we can increase φ, in practice, to save the running time.

Recommendation for control parameters

We have studied the impact of the three parameters of our solution. It is quite important to make recommendations for their settings, as different applications may have different requirements on the detected communities. In **Table 3**, we use "Strict," "Normal," and "Loose" to denote these different requirements on the intra-connectivity of detected communities. We have shown that our solution is robust with respect to variation in parameter settings. Thus, it is easy for us to make the concrete suggestions for parameter values in Table 3. We believe these recommendations are very useful guidelines in practice.

Table 3 Recommendation of algorithm parameters. E is the number of edges.

Requirements	θ	γ	φ
Strict	>3%	>0.3	1
Normal	1%–3%	0.1–0.3	$1 \sim \sqrt{E}$
Loose	<1%	<0.1	$> \sqrt{E}$

Scalability

We select heavily loaded jobs to test the speedup. After monitoring overall resource usage of the preprocessing, core-processing, and postprocessing algorithms, we found that the first round of iteration (*T_Updater* and *S_Calculator*) dominates the resource usage. Most of the topology updating (99.4%) completed in the first round of iteration.

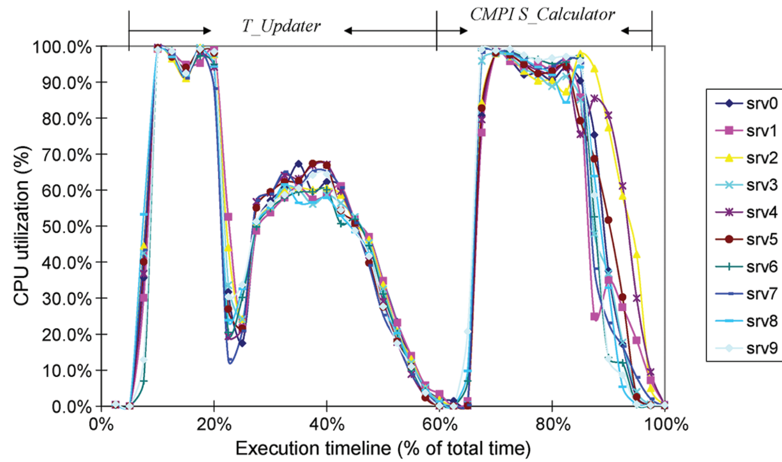
We further noted that when we set filtering threshold θ to 250, the first iteration step of core processing (*T_Updater* and *S_Calculator*) achieves high cluster-wide resource utilization. Thus, we choose the first iteration of the core-processing stage to evaluate the speedup of our solution with respect to increasing hardware allocation. The evaluation is performed through changing the number of active Hadoop slave nodes in our testing cluster.

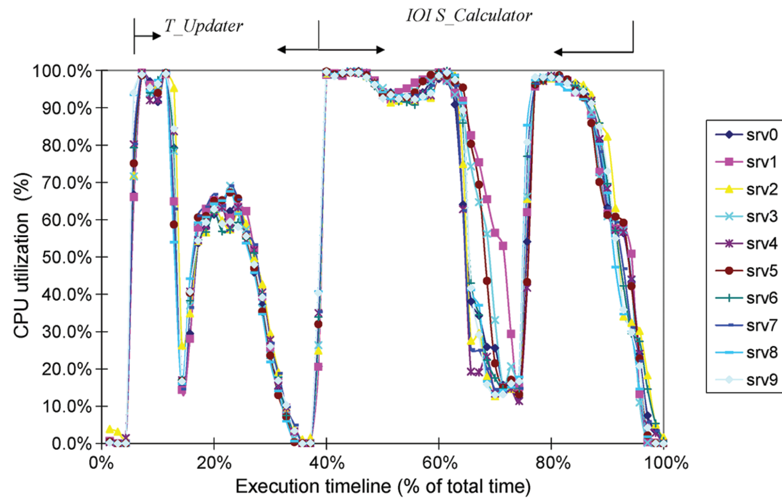
Figure 4(a) and **(b)** show the CPU utilization of first CMPI and IOI-based core-processing iteration for the ten nodes (srv0 to srv9), respectively. The result indicates both that *S_Calculator* is well parallelized and can fully utilize the heavy-threaded multi-core cluster in this bottleneck iteration.

Figure 4(c) shows the speedup test result. The speedup of CMPI is 1.98 when the number of machines increases from four to ten. We obtain 1.94 times speedup for IOI as the number of machines increases from four to ten. Because both CMPI and IOI approaches can be fully distributed to the task workers in the cluster, we expect the community-detection speed of the cluster is nearly linearly scalable with respect to the capacity of resources.

Discussion and future work

Most real-world social networks are *Scale-free Networks* with power-law properties [32]. The most notable characteristic is





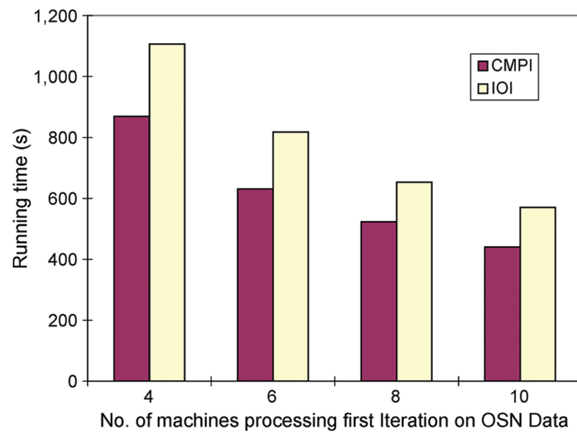


Figure 4

Scalability of the proposed algorithms. (a) Cluster-wide CPU utilization of the first CMPI core-processing iteration $\gamma = 0.2$, $\theta = 250$, $\varphi = 10{,}000$, $m = 236$, and $r = 236$. (b) Cluster-wide CPU utilization of first IOI core-processing iteration $\gamma = 0.2$, $\theta = 250$, $\varphi = 10{,}000$, $m = 236$, and $r = 236$. (c) Speedups of the first core-processing iteration for OSN data, $\gamma = 0.2$, $\theta = 250$, $m = 236$, and $r = 236$.

the long-tail distribution in node degree. Some large-degree nodes are hubs that connect multiple communities formed by nodes with small degree and consequently make community structure unclear. Thus, the clustering coefficient decreases as the node degree increases. DEPOLD is able to filter the noise and computing overhead caused by hub.

Therefore, the principle of DEPOLD can be applied to more graph-clustering algorithms in social networks [14]. The only variable to specify is the approach to merge filtered nodes in the postprocessing stage. Importantly, partial clustering results of unfiltered nodes are made use of to reduce computation overhead of large-degree vertices. The degree of unfiltered vertices is bounded by the constant threshold. With the help of these partial clustering results, merging back filtered vertices could be very fast. We will abstract DEPOLD as a common *Graph-Clustering Accelerator* in the future.

Conclusion

We designed and implemented an efficient community-detection solution for large social networks using MapReduce. A set of preprocessing and postprocessing methods named DEPOLD is proposed to significantly save running time and storage space while improve the detection accuracy. We use a similarity metric to achieve low convergence time and increase robustness in practice. Experimental results demonstrate that our solution achieves one order of magnitude improvement in both processing time and storage space. Furthermore, our experiments indicate a great scaling characteristic with more machines in a MapReduce cluster. Our evaluation shows that the input parameters of our detection solution are robust for real-world social network data.

Acknowledgments

We thank Professor Ben Y. Zhao from University of California, Santa Barbara, for help regarding datasets.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Apache Software Foundation, Sun Microsystems, or Twitter, Inc., in the United States, other countries, or both.

References

1. S. Fortunato, "Community detection in graphs," *Phys. Rep.*, vol. 486, no. 3–5, pp. 75–174, 2010.
2. M. E. J. Newman, "Detecting community structure in networks," *Eur. Phys. J. B—Condens. Matter Complex Syst.*, vol. 38, no. 2, pp. 321–330, Mar. 2004.
3. Facebook Data Tracker. [Online]. Available: http://www.checkfacebook.com
4. M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Phys. Rev. E*, vol. 69, no. 2, pp. 026113-1–026113-15, Feb. 2004.
5. M. E. J. Newman, "Modularity and community structure in networks," *Proc. Nat. Acad. Sci. USA*, vol. 103, no. 23, pp. 8577–8582, Jun. 2006.
6. W. W. Zachary, "An information flow model for conflict and fission in small groups," *J. Anthropol. Res.*, vol. 33, no. 4, pp. 452–473, 1977.
7. M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proc. Nat. Acad. Sci. USA*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002.
8. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
9. J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in MapReduce," in *Proc. ACM 8th Workshop Mining Learn. Graphs*, 2010, pp. 78–85.
10. S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine*1," *Comput. Netw. ISDN Syst*, vol. 30, no. 1–7, pp. 107–117, Apr. 1998.
11. A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Conf. Internet Meas.*, San Diego, CA, USA, Oct. 2007, pp. 29–42.
12. C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, Nuremberg, Germany, Mar. 2009, pp. 205–218.
13. The OSN Data Set. [Online]. Available: http://current.cs.ucsb.edu/facebook/index.html
14. S. E. Schaeffer, "Graph clustering," *Comput. Sci. Rev.*, vol. 1, no. 1, pp. 27–64, Aug. 2007.
15. W. Xue, J. Shi, and B. Yang, "X-RIME: Cloud-based large scale social network analysis," in *Proc. IEEE Int. Conf. Services Comput.*, 2010, pp. 506–513.
16. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 1, pp. 291–307, 1970.
17. J. R. Tyler, D. M. Wilkinson, and B. A. Huberman, "E-mail as spectroscopy: Automated discovery of community structure within organizations," *Inform. Soc.*, vol. 21, no. 2, pp. 143–153, 2005.
18. M. J. Rattigan, M. Maier, and D. Jensen, "Using structure indices for efficient approximation of network properties," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2006, pp. 357–366.
19. U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Trans. Knowl. Data Eng.*, vol. 20, no. 2, pp. 172–188, Feb. 2008.
20. N. P. Nguyen, T. N. Dinh, Y. Xuan, and M. T. Thai, "Adaptive algorithms for detecting community structure in dynamic social networks," in *Proc. IEEE INFOCOM*, 2011, pp. 2282–2290.
21. L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas, "Comparing community structure identification," *J. Stat. Mech.: Theory Exp.*, vol. 2005, p. P09 008, Sep. 2005.
22. P. Pons and M. Latapy, "Computing communities in large networks using random walks," in *Proc. ISCIS*, 2005, pp. 284–293.
23. Y. Zhang, J. Wang, Y. Wang, and L. Zhou, "Parallel community detection on large networks with propinquity dynamics," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2009, pp. 997–1006.
24. D. Gibson, R. Kumar, and A. Tomkins, "Discovering large dense subgraphs in massive graphs," in *Proc. 31st Int. Conf. VLDB Endowment*, 2005, pp. 721–732.
25. GPS: A Graph Processing System. [Online]. Available: http://infolab.stanford.edu/gps/
26. W. Mclendon Iii, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *J. Parallel Distrib. Comput.*, vol. 65, no. 8, pp. 901–910, Aug. 2005.
27. G. Palla, I. Der'enyi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature

and society," *Nature*, vol. 435, no. 7043, pp. 814–818, Jun. 2005.
28. X-rime: Hadoop Based Large Scale Social Network Analysis. [Online]. Available: http://xrime.sourceforge.net/
29. T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, 2009.
30. Ganglia Monitor System. [Online]. Available: http://ganglia.info/
31. H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 591–600.
32. R. Pastor-Satorras and A. Vespignani, "Epidemic spreading in scale-free networks," *Phys. Rev. Lett.*, vol. 86, no. 14, pp. 3200–3203, Apr. 2001.

Received July 16, 2012; accepted for publication August 14, 2012

Juwei Shi *IBM Research - China, Haidian District, Beijing, 100193 China (jwshi@cn.ibm.com).* Mr. Shi is a Research Staff Member in the Information Management department at IBM Research - China. He received his B.S. and M.S. degrees in electrical engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2005 and 2008, respectively. He subsequently joined IBM Research - China where he worked on Big Data analytics, such as social network analysis using MapReduce, Hadoop performance on PowerPC*, and data management and analytics applications across industries. He also worked in Microsoft Research Asia as a visiting student in 2005. Mr. Shi has more than 10 papers published and 15 patents filed.

Wei Xue *Tencent, Inc., China Technology Exchange Building, Haidian District, Beijing, 100080 China (weixue@tencent.com).* Dr. Xue a Staff Research Member in Data Platform department at Tencent, Inc. He received his B.S. and Ph.D. degrees in computer science from Beihang University, Beijing, China, in 1998 and 2006, respectively. He subsequently joined IBM Research - China where he worked on distributed systems, cloud computing, and distributed and parallel large-scale data analytics. He joined Tencent, Inc., in 2011. His current focus is on architecture, algorithms, and performance tuning for online recommendation system. This publication work was done while he was with IBM Research - China.

Wenjie Wang *Shanghai Synacast Media Tech (PPLive), Inc., PuDong New District Shanghai, 201203 China (wenjiewang@pplive.com).* Dr. Wang is the P2P (peer-to-peer) Chief Architect at PPLive. He received his Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 2006. He co-founded a peer-to-peer live streaming company, Zattoo, in 2005 based on his Ph.D. thesis. After that, he served as a Research Staff Member at IBM Research - China. He also worked at Microsoft Research China and Oracle China as a visiting student and consultant intern. This publication work was done while he was with IBM Research - China.

Yuzhou Zhang *Qihoo 360 Technology Company Limited (yuzhou.zh@gmail.com).* Dr. Zhang is a Senior Engineer at Qihoo. He received his Ph.D. degree in computer science and technology at the Department of Tsinghua University in 2010. He served as a researcher at IBM Research - China for 1.5 years after graduation. He also worked for Google China as a research intern for approximately 2 years before graduation. Dr. Zhang was a 2010 Sieble Scholar. His research interest includes data mining, machine learning, distributed large-scale data processing system, and application of these technologies in modern search engines. This publication work was done while he was with IBM Research - China.

Bo Yang *IBM Software Group, China Development Laboratory, Beijing, 100193 China (boyang@cn.ibm.com).* Dr. Yang is a Lead Architect in the Emerging Technology Institute of IBM China Development Laboratory. He received B.S. and Ph.D. degrees in 1997 and 2001, respectively, in computer science from Tsinghua University. He subsequently joined IBM Research - China where he worked on distributed systems, cloud computing, and next-generation communications. He moved to IBM China Development Laboratory in 2012 and is now assuming the architect role for cloud computing. Dr. Yang is an IBM Master Inventor and has filed more than 30 patents and has authored or coauthored more than 20 technical papers.

Jian Li *IBM Research - Austin, Austin, TX 78758 USA (jianli@us.ibm.com).* Dr. Li is a Research Staff Member at IBM Research - Austin. He holds a Ph.D. degree in electrical and computer engineering from Cornell University. His current research centers on Big Data analytics systems and their applications, such as Apache Hadoop, Twitter STORM, IBM BigInsights, and InfoSphere Streams on x86, PowerPC, and other platforms. He has worked in the areas of architectural support for power- and variation-aware computing, interconnection network design for high-performance computing systems, workload-driven three-dimensional integration (3DI) architecture, cost-driven 3DI, architectural applications of non-volatile memory (NVM) and storage class memory (SCM), energy-efficient interconnection networks, data center networks, workload-optimized systems, and Big Data analytics. He holds an adjunct position at the Texas A&M University.