IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 29, 2018, accepted January 21, 2019, date of publication February 7, 2019, date of current version February 27, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2897783

A Parallel Algorithm for Community Detection in Social Networks, Based on Path Analysis and Threaded Binary Trees

STAVROS SOURAVLAS[1], (Member, IEEE), ANGELO SIFALERAS[1], AND STEFANOS KATSAVOUNIS[2]

[1]Department of Applied Informatics, University of Macedonia, 546 36 Thessaloniki, Greece
[2]Department of Production and Management Engineering, Democritus University of Thrace, 671 00 Xanthi, Greece

Corresponding author: Stavros Souravlas (sourstav@uom.gr)

This work is supported by the project ''Algorithms and Applications in Social Networks and Big Data Systems'' which is funded by the Unified Insurance Fund of Independently Employed (ETAA), in Greece.

ABSTRACT Several synchronous applications are based on the graph-structured data; among them, a very important application of this kind is community detection. Since the number and size of the networks modeled by graphs grow larger and larger, some level of parallelism needs to be used, to reduce the computational costs of such massive applications. Social networking sites allow users to manually categorize their friends into social circles (referred to as *lists* on Facebook and Twitter), while users, based on their interests, place themselves into groups of interest. However, the community detection and is a very effortful procedure, and in addition, these communities need to be updated very often, resulting in more effort. In this paper, we combine parallel processing techniques with a typical data structure like threaded binary trees to detect communities in an efficient manner. Our strategy is implemented over weighted networks with irregular topologies and it is based on a stepwise path detection strategy, where each step finds a link that increases the overall strength of the path being detected. To verify the functionality and parallelism benefits of our scheme, we perform experiments on five real-world data sets: Facebook®, Twitter®, Google+®, Pokec, and LiveJournal.

INDEX TERMS Community detection, parallel algorithms, binary trees, social circles.

I. INTRODUCTION

Several systems of high interest to the scientific community can be represented as networks. Examples include the Internet, the World-Wide Web, and the social networks like Facebook (1.6 billion users), Instagram (400 million users), or Twitter (320 million users) [1], [2]. It is clear that these networks are so huge, that their modeling and study via graphs including very large numbers of nodes and edges can be quite cumbersome. Therefore, processing of such data is a very big challenge and it is one of the most ''hot'' topics discussed in the modern literature. Community detection has major importance in social networks.

Research on community detection is highly motivated by the fact that the traditional community detection algorithms fail to scale to the increasing number of users and the number and complexity of their relationships. More efficient algorithms are now necessary to analyze and even predict the user's behavior. This type of analysis is of great importance for organizations and companies, to plan their marketing or advertising policies or for political parties to keep track of user's opinions.

With the explosion of the aforementioned social networks, the problem of community detection has become rather difficult. Particularly, there are three issues that pose a big burden in community detection schemes: (1) The *increasing size* of social networks limits the ability of fast processing of the corresponding graphs. The computations performed should be organized very carefully due to the huge data volumes processed by community detection schemes, (2) the *network structure:* Generally, the aforementioned networks have irregular topologies, where there may be nodes with

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

very low or very high degrees, thus they can be accessed via a few or many different paths. Moreover, the number of nodes connecting different sub-networks can either be very large or very small, giving rise to serious computational issues (load imbalance and delays when many computations are handled by a single node), (3) *network updates:* the social networks are updated very frequently as new users join communities. Thus, network updating should be carefully scheduled to efficiently detect newly formed communities. These issues necessitate the use of highly parallelizable and distributed techniques, so that their burdening effects can be smoothed down.

In this work, we schedule and implement a new parallel community detection strategy, that can also detect overlaps over a large network. The new idea introduced in this strategy is the use of threaded binary trees for community detection and it is used in such a way that race conditions are avoided and load balancing between executing processors is assured. Moreover, the proposed scheme facilitates network updating (new nodes entrance). Our approach is implemented in three phases: In the first phase, each node creates a threaded binary tree of neighboring nodes, based on its connections. In the second phase, the trees generated are traversed in a parallelized step-wise strategy, in order to spot possible ‘‘stronger paths’’ between nodes. At the end of this phase, the paths found are compared against the strength of the existing communities, to determine node membership and detect overlaps. The third phase handles network updates.

The remainder of this paper is organized as follows: Section II describes the related work. Section III offers the theoretical background of the community detection problem and presents the criteria posed in our strategy to detect memberships and overlaps. Section IV presents the three phases of our strategy. Section V presents our experimental results and Section VI concludes the paper and offers aspects for future work.

II. RELATED WORK

Typically, the social networks are organized into groups of users [3]. These users join a network, create their own profiles, publish information and find other users with the same interests. In this way, groups of users are formed within networks. Such groups are referred to as *communities*. Although there is no universally acceptable definition of a community, one can define it as a set of nodes and links in a network, such that its internal connections are ‘‘stronger’’ than its external connections [4]. The nodes of a community are considered similar to each other, dissimilar to the other nodes of the network [5] and represent its users. The edges represent the similarity between the users of one community or between users of different communities. Put it in a different way, a community can be viewed as a sub-network of highly related users, within a huge network composed of a large number of such communities. Different sub-networks are connected since there can definitely exist some sort of relationship between members of different communities. In this sense, two or more communities can be partially *overlapping*, that is, they may have one or many common members. In cases where the common members are too many and very strongly connected, the two communities may be indeed considered as one.

This work addresses the problem of *community detection and overlapping* in large networks with irregular topology. Community detection has a lot of applications because communities are a more realistic approach of modern networks: researchers may belong to one or more scientific communities based on their research interests, medical communities are organized into sets of smaller communities based on specialities, and sport communities can be spotted via publications and comments on pages of related interest. Communities can either be disjoined or overlapped. Although there has been some work on disjoined communities (a good example is the work of Staudt and Meyerhenke [6]), the majority of the latest algorithms study the problem of *overlapped communities*. The main reason behind our choice is that, when absolutely no overlapping is considered, it follows that each node exclusively belongs to one community, which is quite restricting.

The study of community structures is generally related to the problem of *network partitioning* [7]. Typically, the network partitioning problem is defined as the partitioning of a network into a set of groups of approximately equal sizes with minimum number of edges [5]. The general idea is to let the network nodes represent computations and the edges represent communications. However, network partitioning is not the ideal method for the analysis of networks and for community detection. This is firstly due to the fact that in real networks, the communities formed rarely have approximately the same size and secondly because network partitioning does not consider the similarities between nodes (or users), which are inherited in a social network. Moreover, network partitioning is an NP-hard problem, thus heuristics need to be employed.

The *community detection methods* try to group the network nodes based on the relationships that hold among them, in order to form strongly linked subgraphs from the entire graph that represents the whole network [8]–[10]. Apparently, the community detection has turned out to be a graph problem and graph-based methods have been developed to solve the problem in an effective manner. In the remaining of this section, we categorize the community detection schemes found in the literature and briefly discuss the most representative strategies from each category.

Generally, the community detection schemes can be categorized into three basic approaches: (a) top-down approach, which starts from the graph representing the entire network and try to divide it into communities, (b) the bottom-up approach that uses the local structures and tries to expand them to form communities, and (c) the data-structure-based approach, that tries to convert the entire network into a data structure, which is then processed to detect communities. In this section, we discuss the most representative papers from each category.

A. TOP-DOWN APPROACHES

The top-down approach is based on the idea of *graph or link partitioning*, that divides the overall network into small groups, in order to to detect communities. When the links connected to a node are found in more than one communities, this node is assumed to be overlapped. Prat-Perez *et al.* [11] proposed the *Weighted Community Clustering (WCC)*, which computes the level of cohesion of a set of nodes S. The idea behind this work is that good communities are those with a significant number of triangles well distributed among all the nodes. Therefore, to define a community, the WCC measures the ratio of triangles that a node x forms with nodes within a set S, as opposed to the number of triangles that x forms in the entire graph. Also, it computes the number of nodes that form at least one triangle with x, with respect to the union of such set and S. A community including x is well defined when set S includes the largest possible number of vertices that form triangles with x and the smallest possible number of nodes such that x does not form triangles. Ideally, x is well-considered as member of the community defined by S when it forms triangles with all the nodes (in pairs) in S and when the number of nodes such that x does not form triangles is 0. Experimental results have shown that the WCC indeed produces high-density well-defined communities. A similar triangle-based approach, called $k-$mutual-friend subgraph was also used in [12].

Chen *et al.* [13], calculate the node strength for each node at first, and then detect an initial community from the node with the largest node strength (the sum of weights of all the edges connected to the node). Then, the ''strongest'' node is selected and its belonging degree (a measure based on the coefficients of links) is measured against a threshold. The node belongs to a community if its belonging degree is less than the threshold. Apparently, the node's links may be connected to a number of communities, to which the node itself may belong to, according to its belonging degree and its threshold value.

A similar approach is found in [14], where the authors present a strategy that can detect both overlapping and non-overlapping communities and adds one node to a community in every *expanding step*. Specifically, each expanding step computes the *belonging degree* of the nodes to a community C and then, they select the one with the highest belonging degree. This node is temporarily attached to C, forming C'. Then, if the *conductance* of C' is lower compared to the conductance of C, the selected node is finally attached to C.

A newly introduced top-down strategy named picaso was introduced very recently by Qiao *et al.* [15] The proposed scheme detects communities using a modularity-based *mountain* model, which divides the network into chain groups (top-down) and sorts them by the weights of edges. Based on the community features, some edges fall down and others raise like mountains (hence the name). New communities are formed by the mountains produced. An update-modularities phase is also included.

Generally, the top-down approaches are an interesting and well-adopted idea to perform community detection. Their advantage is that they can easily detect overlapping communities, but sometimes this overlapping is too high, if a node is connected to large number of links distributed to many different communities. In such a scenario, processing delays may occur, so the algorithms should be cautious regarding the link connections they consider.

B. BOTTOM-UP APPROACHES

The second kind of approaches starts from local structures and expands to the overall network. During this process, various communities are formed. A number of different ideas is used to implement a bottom-up community detection approach.

Optimization is bottom-up approach that characterizes the quality of an interconnected part of the network. The community is considered as a subgraph identified by the maximization of the nodes fitness. This measure is based on the total internal and external degrees of the nodes of a group (or module). The aim of this optimization problem is to find a subgraph starting from a specified node such that, the inclusion of a new node, or the elimination of one node from the subgraph would lower the fitness value. Nascimento and Pitsoulis [16] presented a Greedy Randomized Adaptive Search Procedure (GRASP) with path relinking, for solving the modularity maximization problem in weighted graphs. A class of $\{0, 1\}$ was used to characterize the family of clusterings in the network. Clustering construction is the first stage of this algorithm and it provides a good starting solution for a local search. In the second stage, the exhaustive search is performed in the neighborhood of a given solution in order to get a local optimum. Finally, relinking is used, a search is performed to explore the space of solutions spanned by two good quality solutions, to find a better solution. Džamić *et al.* [17] proposed a method called Ascent-Descent VNDS, which combines local search with systematic changes of neighborhood structures to escape from local optima traps. This method is used to detect communities by modularity maximization. Scalable heuristics for modularity density are proposed in [18]. Newman [19] proposed the optimization of the ''modularity'' function on possible divisions of a network. The modularity was expressed in terms of the eigenvectors of a characteristic matrix for the network.

Another idea to implement a bottom-up approach is *Clique Percolation*. This method assumes that a community consists of fully connected subgraphs. Sets of such subgraphs may overlap. The community detection is based on searching and identifying neighboring cliques. Initially, it finds all cliques in the network, which are then represented in the graph by a vertex. If two cliques share a predefined number of members, then their corresponding vetrices are connected. Thus, connected vertices on the graph represent network communities. An interesting clique percolation technique was introduced by Farkas *et al*. [20]. In this algorithm, a predefined intensity

threshold was introduced, and only the cliques with intensity higher than the threshold get a community membership. In [21], the community detection process is divided into phases: In the first phase, cliques of $k-1$ members are detected by checking all cliques of $k-2$ members, residing in the neighborhood of two vertices. Then, the connected components of the $k-1$ cliques are checked, in order to form a community. In a more recent approach, Zhang *et al.* [22] address the problem of curse dimensionality from which the clique percolation techniques are suffering. They proposed the Salton index, to characterize node similarities and the weak cliques detected were merged into larger communities, whenever possible.

A new idea was introduced by Xiao *et al.* [23], which is based on the proximity of the nodes inside a network structure. This node location analysis is based one the estimation of the nodes' mass and spotting their location in the network. The Parallel Louvain Method with Refinement (PLMR) is based on the initial Parallel Louvain Method (PLM) introduced by Blondel *et al.* [24]. PLM is a bottom-up approach, that uses modularity as the objective function. In each step, the nodes are transferred to neighboring communities in such a way that the modularity is locally maximized. The process stops when all the communities are found to be stable. The refinement was added by Staudt and Meyerhenke [6] and it is an additional move phase following each prolongation. This move is needed to re-assess node assignments, in order to adapt to changes that have incurred.

Finally, another common idea for implementing a bottom-up approach is *label propagation*. Label propagation is a technique that assigns labels to previously unlabeled data points. Initially, a few nodes only have labels and as the algorithm proceeds, the nodes adopts the labels that most of its neighbors currently have [5].

Gregory [25] proposed an extension of the label propagation technique of Raghavan *et al.* [5] named Community Overlap PRopagation Algorithm (COPRA), in order to be able to detect overlapping communities. Specifically, the label and propagation steps are extended to include information about more than one community, thus each vertex can now belong to more communities. The network vertices that represent cliques have labels that propagate between neighboring vertices so that members of a community are aware of their community membership.

Generally, the bottom-up approaches have the advantage that, in many cases, their complexity is linear (for optimization and label propagation techniques). However, strategies that belong to these sub-categories often fail to detect very small communities, even in cases they are well-defined. This generally happens because the initial local structures do not capture these small communities from scratch, and the expansion method used fails to incorporate node-members of a community. The clique percolation strategies suffer high computational costs, as they need to continuously check every pair of cliques detected, to determine if they can belong to a larger clique.

C. DATA STRUCTURE BASED APPROACH

The data structure based approach is based on the idea of forming a network to some type of data structure (usually in a tree form), which is then analyzed in a way to detect communities. Here, we briefly describe some of the most representative examples of strategies of this type.

Ahn *et al.* [26] use the metric of Jaccard index to compute the similarity for any given pair of links connected to a node. Based on similarities, they build a link dendrogram, which is then cut at some threshold to produce the communities. The Overlapping Community Algorithm (OCA) algorithm [27] is based on the idea of mapping each node to a multi-dimensional vector. Each node subset is then defined as the sum of individual vectors in this set. The fitness function is defined as the directed Laplacian on function O, where O is the squared Euclidean length of a subset vector. The algorithm tries to remove or add a node that results in maximizing the value of the tness function.

Agglomeration algorithms build tree hierarchies starting from small clusters and expanding to larger ones. Clauset *et al.* [28] presented an algorithm that starts from single nodes and build the dendrogram structure which describes the community structure, while maintaining the changes in modularity. A slightly different approach, *matrix blocking* [29] constructs an hierarchy tree by recognizing matrix column similarities between nodes. Then, partial clustering is computed in the graph, where each node is not necessarily considered as community member (in contrast to complete clustering). Finally, another interesting data structure approach is presented in [30]. The scheme is called graph-skeleton-based clustering (gSkeletonClu) and its idea is to projecting an undirected network to its maximal spanning tree. Then, the optimized clusters on the tree are detected.

Generally, the idea of converting a network to a tree structure is an interesting one, however, this translation should be implemented carefully, as it may be very expensive in terms of computation costs, especially when processing networks of millions nodes or edges. Careful parallelism is the solution for this issue. This work belongs to the category of data structure based techniques. Its novel idea is that it converts the network into threaded binary tree structure using carefully designed parallel operations, to reduce the computational costs.

III. OUR COMMUNITY MEMBERSHIP CRITERIA

Several criteria have been proposed to detect community membership. Before we introduce our criteria, we briefly present some of the representative criteria found in the papers presented in the Related Work Section. More criteria can be found in the papers cited.

1. *Node Location Analysis [23]:* The node mass is evaluated and the affiliation between nodes is determined based on their position in the structure.
2. *Conductance Function [14]:* The weights of the cut-edges of a network divided by the weight of all network edges. When the conductance is low, more nodes lie within a community, which is considered stronger.

3. *Salton Index [22]:* It is based on the section of the neighboring sets of two nodes. The larger this index, the more common neighbors the two nodes have, so it is more likely that they are members of the same community. The Jaccard and Sørensen index have also been referred in this work as possible candidates.
4. *Belonging Coefficient [25]:* It indicates the strength of a node's membership to a community. Each propagation step sets the nodes label to the union of its neighbors' labels, sums the belonging coefficients of the communities over all neighbors, and normalizes the result.
5. *Modularity [19]:* The number of edges falling within groups minus the expected number in an equivalent network with edges placed at random.

This paragraph will provide a few details regarding our community membership criteria. Some of these aspects have been discussed in our previous work [31], but we will briefly discuss them here for completeness.

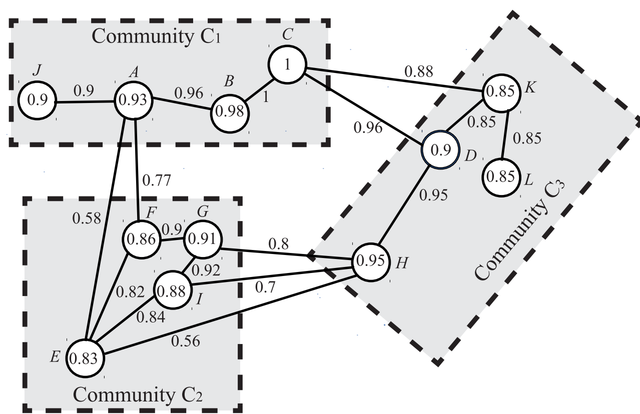

FIGURE 1. An irregular network with 3 communities.

Consider a small network of irregular topology, such as the one shown in Fig. 1, where there are already three communities C_1, C_2, and C_3. Also, assume that the links between nodes indicate that there exists some relationship between them, in the sense that they have similar opinions on topics or have the same interests and likes [31]. Real data sets, in which some similarities can be found are freely available (for example, visit: http://snap.stanford.edu/data/index.html).

Fig. 1 reveals some aspects than need to be considered:

1) User A is related to user F that belongs to C_2. Thus, there is a chance that A also belongs to C_2.
2) User A is indirectly related to C_3 via E or via B, C. Thus, there is a chance that A belongs to C_3 as well. In this case, nodes E or B, C may also belong to C_2.
3) Continuing Topic 2, if there is a high percentage of overlapping between two communities, chances are that they are actually one community.
4) Not all users of a network are necessarily related (e.g., two Facebook users although they are not related, they may have common friends). In Fig. 1, A is not directly to H, but it is related to E. In its turn, E is related to H. To continue with the Facebook analogy, this is a case that a user (for example A) is unaware of community C_3 until he/she visits the page of another user that has some relationship to members of this community (like E). In other words, the path relating A and H may be strong enough, so the two nodes can be members of the same community.
5) Newly connected users should also be examined and, based on their behavior, characterized as members of one or more communities.

Let $G = (V, E)$ be a weighted, undirected graph, where V and E are the sets of nodes and edges, respectively. Nodes represent users and edges represent the relationship between two users (e.g., friendship in Facebook). The *similarity* $w_{i,j}$ between users i and j is the weight of the edge that connects i and j. This value lies in the interval $[0\ldots1]$. As will be described in the Experimental Results and Discussion section, this value is computed (see Eq. 8) based on the real data collected for various networks. For example, a value of 0.78 shows that the likes or opinions of i and j can be considered quite converging, at a percentage of 78%. Bu *et al.* [3] presented a table of user phrases that indicate supportive or opposing attitude towards a comment or opinion, etc. These phrases are accompanied by a value that shows the degree of support.

The network nodes are also weighted: the weight of a node i indicates its *Network Connectivity Degree* (NCD), i.e., how well the preferences, likes, views of a user are fitted to a community. The network connectivity degree is also between $[0\ldots1]$. In a network representation, the letters are the node names, the edge values are the similarities and the node values indicate network connectivity degrees. The network connectivity degree of a user i, NCD_i, is computed as follows:

$$NCD_i = \frac{\sum w_{i,j}}{n_e} \tag{1}$$

where $w_{i,j}$ is the weight of any edge that relates user i with any user j that lies in the same community and n_e is the number of such edges. For example, consider community C_1. User A has two internal links, namely with users J and B and two external links, with users E and F. To compute $NCD_{\mathtt{A}}$, we only consider the internal links and we have $NCD_{\mathtt{A}} = \frac{(0.9+0.96)}{2} = 0.93$.

Therefore, the *Average Community Connectivity* (ACC) for a community C is defined as the average of the NCDs of all the N nodes in the C, that is:

$$ACC_C = \frac{\sum_{i=1}^{n} NCD_i}{N} \tag{2}$$

This measure indicates how strongly the members of a community are related.

In this work, we will use a double criterion to identify a user's membership to a community.

1. A user can be considered as a community member, either *directly*, through a relationship with one or more users of the community, or *indirectly*, through a relationship

with a user that is related to a member of the community. To determine a user's relationship to a community and possible overlaps, we need to detect the existence of "stronger" in terms of weight paths, within "weaker" that already determine the membership to a community. We introduce the notion of *path strength (PS)* as the *sum* of weights on a path that connects two *not directly connected* users i, j, divided by the number of hops on this path

$$PS_{i,j} = \frac{(\sum_{i=1}^{k} w_{i,r_{i-1}}) + w_{r_i,j}}{k} \tag{3}$$

where r_i denotes one of the k intermediate nodes j and j. Apparently, a criterion to determine the membership of a user U to a community, would be the detection of a path starting from U, which is stronger than the ACC of this community.

2. The length of the paths found to satisfy the first criterion should not exceed the *diameter* d of community C, that is, the longest path that can be found among the nodes of C. This condition is necessary because, given a very long path, chances are that the path similarity would be reduced. Moreover, the ACC values are quite large, thus condition 1 may never be satisfied. The diameter of C gives a reasonable bound for the acceptable path lengths. To summarize the conditions:

Mathematically, the conditions described above are expressed by the following inequalities:

$$PS_{i,j} \geq ACC_C \tag{4}$$

$$\delta_{i,j} \leq d_C \tag{5}$$

Before concluding this paragraph, let us illustrate this double criterion with a brief example. The Average Community Connectivity for community C_1 of Fig. 1 is the sum of the NCDs for nodes J, A, B and C divided by 4, that is, $\frac{0.9+0.93+0.98+1}{4} = \frac{0,95}{4} \approx 0.95$. Now, the path strength of the path connecting nodes I and C through nodes G, H, and D is $\frac{0.92+0.8+0.95+0.96}{4} = 0.9$. In this case, $PS_{IC} < ACC_{C1}$ so the criterion of Eq.4 is not satisfied. Moreover, $\delta_{IC} = 4$ and $d_{C1} = 3$ so the criterion of Eq. 5 is also not satisfied. So, if user I can't be determined as a member of C_1 by examining the path from I to C through nodes G, H, and D. On the other hand, if we consider the membership of node H to community C_1 through node D, we have $PS_{HC} = \frac{0.95+0.96}{2} = 0.955 > ACC_{C1}$. Moreover, $\delta_{HC} = 2 < 3$, so the criteria of Eq. (4) and Eq.(5) are satisfied and H can be determined as a member of C_1 by examining the path from H to C, through node D.

IV. OUR COMMUNITY DETECTION ALGORITHM

Our community detection strategy includes three phases: (A) Threaded binary tree generation (B) Path analysis, and (C) Network updating. In this section, we present each phase in detail and perform analysis of the computations required at node lever, which is important to analyze the scaling (strong and weak) of the proposed scheme.

A. THREADED BINARY TREE GENERATION

The threaded binary tree generation is an expanding process that starts from some initially formed small communities and it keeps expanding as new nodes enter the system. This updating process is the last phase of the proposed scheme and will be described in Section IV.C. For each node, which is the *tree root*, we map the root's subnetwork-of-neighbors to special type of binary tree, which has a root and a right subtree, but no left subtree. Moreover, each node of this tree, except the root, has at most two threaded links: a *right thread* (RT) linked to the root and a *left thread* (LT) linked to a carefully selected sibling, as we describe later.

The root's neighbors are first sorted in a list in ascending order, based on the weight value that connects them to the root. Now, if $T = T_1, T_2, \ldots T_m$ is the set of a node's m neighbors (which are siblings since they have the same parent, the root), then the threaded binary tree $B(Root)$ can be rigorously defined as follows:

a) If $m = 0$, then $B(Root)$ is empty.

b) If $m > 0$, the right subtree of $B(Root)$ is $B(T_1), B(T_2), \ldots B(T_m)$, that is, the right subtree is formed by the subtrees of the roots's neighboring nodes. More specifically:

  b1) The children are placed, one at each level of the tree, according to their order in the sorted list. The right link of each child points to the next level sibling, that is the next element of the sorted list.

  b2) The left link of each child is its own subnetwork-of-neighbors, in other words a special tree with this child as the root.

c) The right thread of each child links to the root. The right thread is represented by an arrow pointing from children to the root vice versa.

d) The left thread of each child T_i, $i \in [1 \ldots m]$ links to a sibling $T_{i'}$, $i' \in [1 \ldots m]$, $i \neq i'$, such that:

$$\frac{w_{T_i,T_{i'}} + w_{T_{i'},root}}{2} > w_{root,T_i}. \tag{6}$$

Inequality (6) states that the indirect relationship between T_i and the root through $T_{i'}$ is stronger compared to their direct connection. Not all the children have a left thread. To locate a left thread, we search for a relationship between T_i and a sibling $T_{i'}$, such that (6) is satisfied. Note that $T_{i'}$ can only be found at a level "lower" than T_i's, because if $T_{i'}$ was located "above" T_i, then $w_{T_{i'},root} < w_{T_i,root}$ and (6) would not be satisfied. The left thread is represented by an arrow pointing from T_i to $T_{i'}$

For example, consider the neighborhood of node A. Node A is the root and placed at level 0 of the tree. The sorted list of neighbors is | E | F | J | B | , because $w_{A,E} = 0.58$, $w_{A,F} = 0.77$, $w_{A,J} = 0.9$, and $w_{A,B} = 0.96$. Now, the children are placed, one at each level in ascending order, thus, E is at level 1, F is at level 2, J is at level 3 and B is at level 4. The right subtree of B(A) is now completed

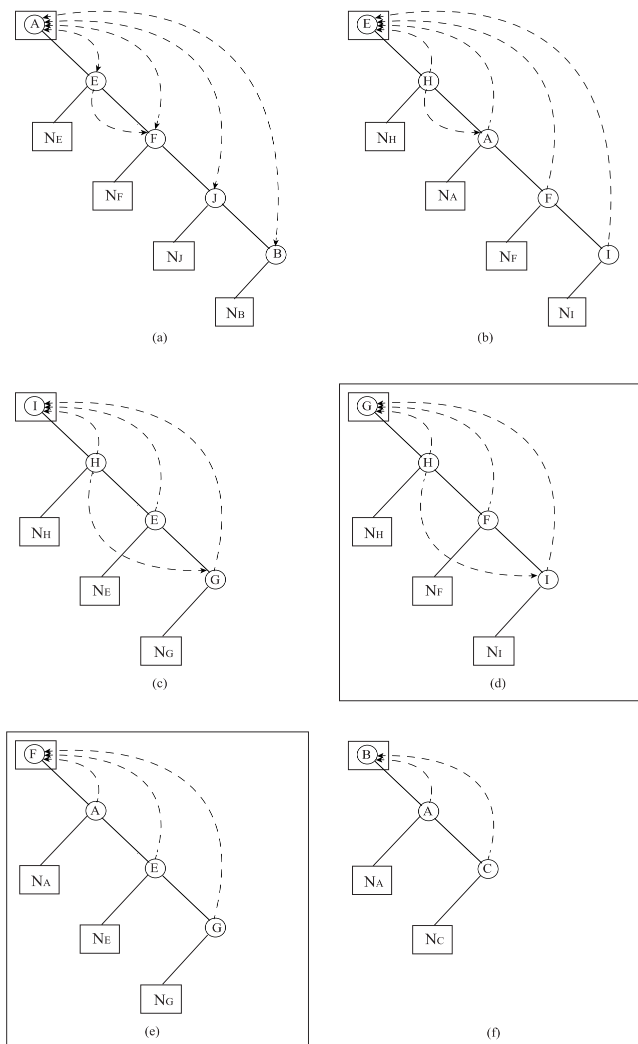

FIGURE 2. Threaded trees.

by adding the corresponding neighborhoods to the left links of all the children. Now, there remains to add the threads. The right threads are links to the root. The left thread is used to find a stronger path from the root to a child vice versa (triangle handling). For example, E is linked to the root with a weight value of 0.58. However, there exists a sibling of E at a lower level of the tree, namely F, such that $w_{E,F} = 0.82 > w_{E,A}$. Moreover, $w_{F,A} = 0.77$ and $(w_{E,F} + w_{F,A})/2 = (0.82 + 0.77)/2 = 0.795 > w_{E,A}$. Thus, we place a left thread for node E, which links to F, indicating that the relationship between E and the root A is stronger if it is considered as an indirect relationship through F rather than as a direct relationship. The left thread is shown by an arrow from E to F. Fig. 2 shows the binary threaded trees for nodes A, E, I, G, F, and B, for the network of Fig. 1.

For example, for the subnetwork of A, the memory word that stores the links of node E would look like:

E	F	F	A

for the left link, right link, left thread, and right thread values, respectively.

1) PARALLEL IMPLEMENTATION OF THE THREADED BINARY TREE GENERATION

As explained, the neighbors of a node in its subnetwork-of-neighbors are represented as a special type of a threaded binary tree. Actually, these nodes form a group and they can be kept in computer memory together. Only one pointer to the sorted list of these nodes is necessary for reference to them. All the operations on data, which are necessary to form the threaded binary trees can be expressed as a sequence of parallel operations on groups of nodes that belong to different subnetworks (trees). Thus, it is possible to introduce a parallel iterator, that takes as an argument a pointer to a set of nodes and applies the proper operations on the data set (sorting and tree generation).

Although the use of such iterators can conceal details concerning load balancing and race conditions, we need to comment on these two important issues. First, load balancing can be widely achieved by grouping the parallel tasks executed on several processors, based on the number of neighboring nodes that exist in the trees. For example, if the iterator executes in parallel for the lists of nodes A, E, and H (all have 4 neighbors), then the load is balanced over the three participated processors. Note that the trees for these three nodes are identical, with the exception of the presence/absence of left threads. So, a more effective usage of the parallel system can be achieved if the nodes are separated into groups of equal number of neighbors. Second, the parallelization strategy incurs race conditions. This is obvious, since, for example, by the time a node's T_i left thread value is evaluated when constructing the threaded binary tree for a root node $T_{i''}$ on one processor, another processor may be evaluating the left thread of $T_{i''}$ when constructing the threaded binary tree for root node T_i. However, this seemingly difficult situation is overcome by the fact that, in both cases, the left thread will finally end up to be the same node $T_{i'}$, although intermediate results may differ. For example, note the threaded binary tree for A: the left thread of E is F. Then, note the threaded binary tree for E: the left thread of A, again, is F.

Algorithm 1 presents the parallel version of the threaded binary tree generation. The sorting iterator SORT takes advantage of the fact that the unordered lists of the neighbors of each node can be grouped based on the number of elements. Thus, all the lists with the same number of elements can be sorted in a pipeline fashion, to avoid having idle processors. This pipelining is shown in Fig. 3, for $P = 16$. With no loss of generation, we can assume that we have $P = 2^e$ nodes. For example, in Fig. 3, we have $e = 4$. Also, assume that we work with a group of lists, which all include the same number of elements, m, corresponding to all users with m directly connected users (neighbors). Each list is initially divided into $\frac{m}{P/2}$ elements, which are delivered to $P/2$ processors, to create the first pipeline stage. These processors sort these sub-lists in parallel and forward the sorted sub-lists "one level up" to the next $P/4$ processors. In the meantime they send a notification that they can receive

Algorithm 1 Parallel Threaded Binary Tree Generation

input : A graph $G = (V, E)$ and the weight values
output: A threaded binary tree for each node

```
1  SORT(*T) //parallel iterator with pointer T
2                  // (unordered list of neighbors) as input
3  while all nodes v ∈ V not exhausted
4      for i ∈ 1 to ρ − 1 do
5          create pipeline_task_i with m/2^i elements
6          in parallel sort the elements in ascending
7          order of weight values
8          move the sorted list to next P^{i+1} processors
9          // one level up the tree
10         input list from next node v ∈ V to set up a
11         new pipeline_task_{i+1}
12     end for;
13     L_v ← [T_1, T_2, ... T_m] // ordered list
14 end while;
15 end SORT;
16
17 TBTGEN(*L_v) //parallel iterator with pointer L_v
18                 // as input
19 begin
20 for v ∈ V in parallel do // v is root
21     for i ∈ 1 to m
22       LL_i ← community_ref // Left link value
23       RL_i ← L_{i+1} // Right link value
24       RT_i = v // Right thread value
25       max = w_{i,v}; // Left thread computations
26       for j ∈ i + 1 to m
27         if ((w_{i,j} + w_{j,v})/2 > max)
28         then
29           max ← (w_{i,j} + w_{j,v})/2
30           LT_i = j; // Left thread value
31         else LT_i = ∅;
32       end for;
33
34 Save data for v : | LL_v | RL_v | LT_v | RT_v |.
35 end for;
36 end TBTGEN;
```

the next list. Thus, at any given time, each level of processors finishes a sorting operation, forwards the result one level up and receives the next sub-list. The data structure we use affords this type of pipelining, because groups of equal (or almost equal) sized lists can be chosen for processing each time.

The parallel Threaded Binary Tree Generator (TBTGEN) begins at line 17. It takes as an argument a pointer to an ordered list of neighbors of a node v and applies necessary operations to generate the tree for the given data (lines 20 to 34). The variable *max* (line 25) is updated (line 29) each time (6) is satisfied.

There are two factors that can significantly affect the performance of SORT and TBTGEN, as will also be clear

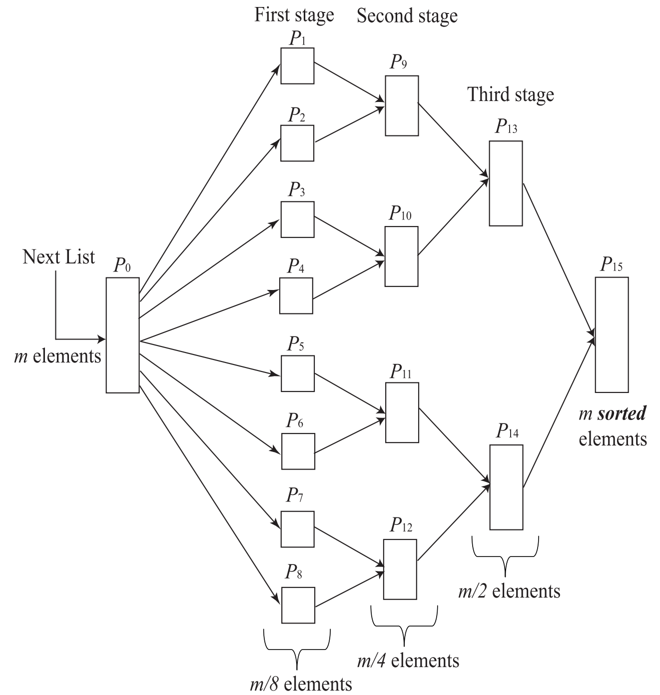

FIGURE 3. Example of generating sorted sub-lists in a pipeline fashion.

from the experiments presented later: (1) The SORT iterator is mainly affected by the large number of inter-processor communications (overheads) especially for networks with large number of links like Google plus, and (2) The TBTGEN is mostly affected by hyper-threading, since the threads used are more memory-bound (many writings to memory). Moreover, the use of larger number of threads does not improve performance, because of the often context switches that may be required.

2) COMPLEXITY ANALYSIS

To analyze the complexity of the threaded binary tree generation phase, we consider separately the time required to execute the two parallel iterators. The first iterator works in a pipeline fashion for P processors (with no loss of generality, we assume that sorting is organized in e pipeline stages, where $e = log_2 P$). In the first stage, each of the $\frac{P}{2}$ processor sorts $\frac{m}{P/2} = \frac{2m}{P}$ elements, with average time $\frac{2m}{P} \log_2(\frac{2m}{P})$. During the second stage, $\frac{P}{4}$ processors sort $\frac{4m}{P}$. Each of the $\frac{P}{4}$ processors receives two sorted lists of $\frac{2m}{P}$. Thus, it inserts a list of $\frac{2m}{P}$ elements into another sorted list of $\frac{2m}{P}$ elements and this insertion can be implemented, on the average in $\frac{2m}{P}$ (each element added from one list to the other reduces by half the comparisons required for the remaining elements, which are placed in their position after a small constant number of comparisons). Similarly, in the second stage $\frac{P}{8}$ processors sorts $\frac{8m}{P}$ elements by inserting a list of $\frac{4m}{P}$ elements into another sorted list of $\frac{4m}{P}$ elements. This can be implemented in $\frac{4m}{P}$. Finally, during the last stage, a processor sorts two lists of $\frac{m}{2}$ elements by inserting $\frac{m}{2}$ elements into a sorted

list of $\frac{m}{2}$ elements in $\frac{m}{2}$ time. During the last pipeline stage, the other pipeline stages process the weights of the links that connect the next node $v \in V$ to all its neighbors. This means that, in the worst case, the total time required to complete the sorting process is $\frac{2m}{P} \times e$. A single processor would require $m \log_2 m$ to sort m elements. Thus, the theoretical speedup is $\frac{m \log_2 m}{\frac{2m}{P} \times e}$. However, as our results show, such a speedup is not achievable, due to overheads incurred by the large number of inter-processor communications.

The time required by second iterator TBTGEN is highly dictated by the number of comparisons needed to compute the left thread of each node. Here, in each of the n in total neighborhoods, we need to compute at most $m-1$ left thread values (the largest element will necessarily have a null left-thread value). To compute the left thread of the minimum element of the list, we need $m - 1$ comparisons, for the next element we need $m - 2$ comparisons, etc. So, we need $(m-1)+(m-2)+\cdots+2+1 = m \times \frac{m}{2} = \frac{m^2}{2}$ comparisons for each neighborhood. Thus, this iterator is more computationally intensive, compared to SORT, and moreover requires temporary storage of intermediate results.

The memory required by the TBTGEN iterator to complete the binary tree representation depends on m, the number of neighbors per node. For m neighbors per node, the threaded binary tree structure can be saved into m 4-byte memory words per node, where m is the number of the node's neighbors. These four bytes are used for the left link, the right link, the left thread, and the right thread. Thus, each processor that processes a set of N nodes in total, would require $4Nm$, words, 4 bytes per word, for a total of $16Nm$ memory space for storage. Moreover, one more word is required to save the intermediate *max* values. So, $17Nm$ is the total memory required per processor, to perform the TBTGEN.

B. PATH ANALYSIS

The path analysis phase is used to detect community memberships for certain nodes and to uncover community overlaps. Generally, [32] overlapping occurs when "stronger" paths are found "within" "weaker" ones. In this sense, the major concern of the proposed stepwise path analysis described here is to detect "ever-increasing" paths, that is, paths that, in every step, they become stronger. At the end of this procedure, inequalities (4) and (5) are evaluated to determine node memberships to a desired community and to detect community overlaps. The path analysis is described in Algorithm 2.

The algorithm is executed in parallel for multiple threaded binary trees and it is implemented in a straightforward way. To keep obtaining stronger paths at each step, the left threads are used to increase the current path strength, in cases where there exists a sibling of T_i, such that inequality (6) holds. The path updates are incremented by two (one from the root to the left thread of T_i, $LT[T_i]$, and one from $LT[T_i]$ to T_i (line 11). To restrict unnecessary iterations, we deactivate the

Algorithm 2 Binary Threaded Tree Traversals

input : Per node threaded binary tree
output: Largest path similarities

1. Path/Sim(*$Root$) *// parallel iterator with pointer T*
2. *// (threaded binary tree) as input*
3. cur_path $\leftarrow \emptyset$
4. set active_list and Root
5. $Root \leftarrow^* T_{act}$ *// pointer to next node processed*
6. $updates \leftarrow 0$ *// number of path updates*
7. **for all** nodes **in** active_list **do in parallel**
8. Read_active_list;
9. $T_i \leftarrow RL[*T_{act}]$
10. **if** $LT[T_i] \neq \emptyset$ **then** {
11. $\ell_{cur_path} \leftarrow \ell_{cur_path} + w_{root,LT[T_i]} + w_{LT[T_i],T_i}$
12. updates $\leftarrow$ updates + 2
13. cur_path = cur_path $+\{^*T_{act} \rightarrow LT[T_i] \rightarrow T_i\}$
14. $LT[T_i] \leftarrow^* T_{act}$
15. $\{active_list\} \leftarrow T_i(\{active_list\})-$
16. $\{cur_path\} - \{T_i \ldots LT[T_i]\}$
17. *// Remove current path and "lower" neighbors*
18. *// from active set*
19. **else**
20. {
21. $\ell_{cur_path} \leftarrow \ell_{cur_path} + w_{root,T_i}$
22. updates $\leftarrow$ updates + 1
23. cur_path = cur_path $+\{T \rightarrow T_i\}$
24. $T_i \leftarrow^* T_{act}$
25. $\{active_list\} \leftarrow T_i(\{active_list\})-$
26. $\{cur_path\}$
27. }
28. $Root \leftarrow T_i$ *// change neighborhood*
29. **if** Root is in C **then terminate**
30. }
31. }
32. path_strength $= \ell_{T_i} \div$ updates

nodes "above" $LT[T_i]$, so that they are not visited when the threaded tree rooted at T_i is examined (line 12). The nodes deactivated are impossible to increase the path strength in a next step. The most important structure in Algorithm 2 is the active_list. Each time a new node T_i is processed, the active_list is updated, so that it includes the nodes that our current path can follow, if we remove from T_i's list of neighbors, $T_i\{(active_list)\}$, all the nodes already in the path, and all the nodes deactivated as unable to increase the current path strength. The path strength ℓ_{cut_path} is updated by the sum of the weights of the two aforementioned links (see line 13).

For example, let us use the threaded trees of Fig. 2 to test the membership of node A to C_3. In this case, we need to start with a threaded tree similar to Fig. 2(a), where *Root* is A. We can assign the necessary traversals to four processors, namely, p_0 to p_3. The active lists for the four processors are set differently, as indicated below:

TABLE 1. Finding "stronger" paths for the network of Fig. 1.

processor	Iteration	Root	cur_path	active_list	Next node to be processed
p_0	–	A	$\emptyset$	{E, F, J, B}	A
	1	A	A → F → E	{I}	I
	2	E	A → F → E → I	{I, H, G, }	I
	3	I	A → F → E → I → G → H	{$\emptyset$}	H
p_1	1	A	$\emptyset$	{F, J, B}	A
	2	F	A → F	{G}	G
	3	G	A → F → G	{H, I}	G
	4	G	A → F → G → I → H	{$\emptyset$}	H

- For p_0: {E, F, J, B}.
- For p_1: {F, J, B} (all nodes in A's neighborhood would be in the paths found except E).
- For p_2 {J, B}, (only nodes J, B from A's neighborhood will be in the paths found).
- For p_3 {B}, (only node B from A's neighborhood will be in the paths found).

Let us see how Processor p_0 will find the path. Processing starts from the right link (RL) of the node pointed by $*T_{active}$, that is, the right link of A, which is E (line 9). Since the left thread of E is F, it follows (line 10) that $\ell_{cur_path} = w_{A,F} + w_{A,E} = 0.77 + 0.58 = 1.35$ (line 11) and updates = 2 (line 12). The current_path will be A→F→E (line 13). Pointer $*T_{active}$ will now point to F (line 14). The new active list will be node E's active list (see Fig.2b), that is E, H, A, F, I minus the nodes in the current path (A, F, E), minus the links above $LT[T_i]$ =F (in this case E, H, A, see Fig.2b). Note that the use of link E→H can't increase the path strength. Thus, the new active list includes I. Finally, node T_i =E will become the root by the end of Path/Sim routine (line 29). Now, the second iteration is about to start, and we will continue processing from the right link of F (line 10), which is I, inside the root node's, E, neighborhood. Because I has no left thread (line 10), the block starting from line 21 executes and $\ell_{cur_path} = 1.35 + 0.84 = 2.19$ and the number of path updates becomes $2 + 1 = 3$ (line 22). The cur_path becomes A→F→E→I. Now, the pointer $*T_{act}$ will point to the new node to be processed, again $T_i = I$, but this time processing T_i =I becomes the root (line 28) and neighborhood changes, that is, processing continues from I's neighborhood and the active_list will include I, H, E, G (see Fig.2c), minus the nodes already in the path A, F, E, I=H, G. Note that the previously excluded node H is now included in the active_list, as we have moved to a different neighborhood. Processing starts from I's right link, which is node H. Because H has a left thread equal to G, we add $\ell_{cur_path} = 2.19 + w_{I,G} + w_{G,H} = 2.19 + 0.92 + 0.8 = 3.91$ and we add two more path updates, thus updates becomes $3 + 2 = 5$. Now, the current path becomes A→F→E→I→G→H. The next node being processed is now H, but we have already reached $C = C_3$. This means that processor p_0 has found an ever-increasing (stronger) path A, through E, with path strength $\ell_{T_i} \div$updates $= 3.91 \div 5 \approx 0.78$ (line 32). Table 1 shows the computations performed by processors 0 and 1.

1) COMPLEXITY ANALYSIS

One can easily see that we can develop the binary tree of Fig.2 by replacing the squared nodes with their neighboring nodes that have not been deactivated or processed yet. The resulting tree can be further developed if for each node:

(S1) Add the first element (not deactivated or processed yet) of its sorted list of neighbors as left link

(S2) Add the remaining neighbors (not deactivated or processed yet) of the node selected in (1), as the right link.

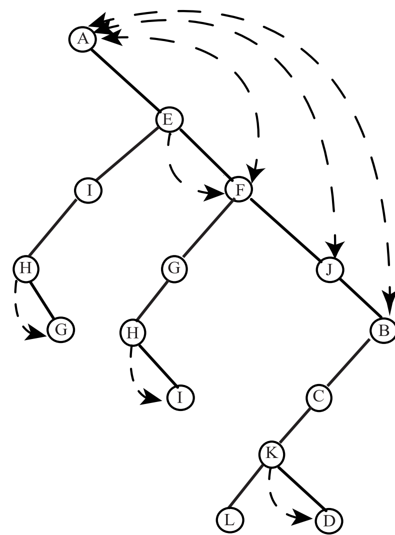

FIGURE 4. Development of binary tree when executing Algorithm 2.

This development is shown in Fig. 4. The tree formed can clearly be separated into subtrees $T_1, \ldots T_4$, each rooted at one of A's neighbors. Each of these trees is traversed in the following way:

(1) The left links are visited once.

(2) Each visit to a right link may take one additional step (using the left thread, to increase the path strength). Such examples, are nodes H (in I's neighborhood, visited through G), H (in G's neighborhood, visited through I), E (in A's neighborhood, visited through F) and D (in C's neighborhood, visited through K).

Assume that we start from a node with m neighbors as root. On the average, for every node added as a left link (S1), there are $\mathcal{M}$ unprocessed and active neighbors, added to its

right subtree. Also, on the average, the value of $\mathcal{M}$ is halved each time we add a new left link due to deactivations occurred in the previous steps. Each added node at level h increases the subtree height to $\mathcal{M} + h$. Assuming a maximum height of $\mathcal{M}$ (caused by the addition of $\mathcal{M}$ unprocessed and active neighbors, added to a right subtree rooted at an "upper" level node) for the resulting tree and that the total number of nodes of this tree is $\mathcal{N}$, the internal path length for each subtree would be at most [33]:

$$(\mathcal{N} + 1)\mathcal{M} - 2^{\mathcal{M}+1} + 2, \quad \text{where} \quad \mathcal{M} = \lfloor \log_2(\mathcal{N} + 1) \rfloor \tag{7}$$

In the example of Fig. 4, the maximum value of $\mathcal{M}$ found was three (three active, unprocessed nodes), once for node E (I was added as left link, followed by H and G) and once for node B (C was added as left link, followed by K and D). Thus, the height of the tree rooted at A is 3+4 (level of B)=7. Applying Eq.(7), we find that this tree can have at most $16\times3-2^4+2 = 48-18 = 30$ internal paths. In this example, the internal paths are 25. Also, note that the external nodes of this tree are the nodes of community C_3, except I and G, which are the left threads of H, so a visit to H is implemented through the threads.

The second phase of our scheme is by far the most computationally intensive, as it includes traversals to subtrees that can grow quite large, especially in networks with large number of edges (although the deactivations incur somehow control this growth, as explained previously) and demands temporary storage of current paths found. The results presented in the next section will show that path analysis benefits from parallelism, however the speedups are low.

2) COMMUNITY OVERLAPS

The algorithm described above detects a number of ever-increasing (or stronger) paths from a root to a community C. To decide if this root can be considered a member of the target community C, we compare the strengths of the paths found to the ACC_C. Also, we compare the path lengths with the diameter of C [see Inequalities (4) and (5)]. In case (4) and (5) are satisfied by at least one of the paths found, then the node examined can be considered as member of C. A stricter approach would require that a percentage, say 50%, of the paths found satisfies (4) and (5). In our experimental results, we use this strict approach.

C. NETWORK UPDATING

One of the main problems in the analysis of social networks is that their structure changes in a rapid way, from moment to moment. To handle these changes, we consider two cases: (1) a new node enters the network with just one relation, and (2) a new node enters the network with multiple relations.

1) NEW ENTRY WITH ONE CONNECTION:

In normal situations, this is not the case. However, if a new node I enters with just one relation to a node, say A, then its membership to a community can be determined using the computations for A. In the paths found, we just add $w_{I,A}$ and examine inequalities (4) and (5).

2) NEW ENTRY WITH MULTIPLE CONNECTIONS:

In such a scenario, we generate a new threaded binary tree for the new node and perform path analysis. Also, this new-coming node has to be included in the threaded trees of all its new neighbors. However, this process is equivalent to an insertion to a sorted list, which can be implemented in a binary fashion, in $\log_m$ time.

It is important to notice that, since the algorithms presented are fully parallelizable, multiple new insertions can be parallelized in a straightforward way, using Algorithms 1 and 2 presented above. To prevent possible race conditions when multiple insertions incur, we group the new-coming nodes based on their connections. For example, a number of new connections to node A are processed in parallel using the threaded trees formed for A, while another group of new connections to node F are processed in parallel using the threaded trees formed for A. Again, for the new entry case, the number of parallel processes depends on the system available.

Let us analyze the updating process. Initially, all the nodes are members of a single community and each node has its own sorted list of neighbors. Communities are expanded by adding new nodes. A node may have a number of relations inside a community (internal links) and a number of relations outside a community (external links). The *local clustering coefficient* (see [34], denoted as *lcc* is the ratio of external degree/total degree for each node. Specifically, each node shares *lcc* of its links with the members of its community and $1 - lcc$ with members of other communities. Apparently, $lcc \in [0 \ldots 1]$. As new nodes enter the network, the *lcc* changes. When the newly inserted node has most of its links to nodes of the same community, thus its $lcc \rightarrow 1$, then the algorithm performance decreases, as the following proposition describes.

Proposition 2 (Complexity Analysis for Network Update Phase): The complexity of the network update phase depends on the lcc of each node. The performance of the network update phase increases when the lcc tends to 1 and decreases when the lcc tends to 0.

Proof: Assume that, for a newly inserted node i, all its links, say m in total, lie in community C, thus $lcc = 1$. In this case, Algorithm 1 will generate a threaded binary tree for i with m neighbors. Then to examine i's membership to another community C', we need *m messages passed* to the processors that store the stronger paths from i's neighbors to C'. The path strength values will be updated by the weight values of w_{ij_ϱ}, where j_ϱ are the neighbors of i, $\varrho \in [0 \ldots m-1]$. Thus, the complexity is $O(m) + \Delta$, where Δ is the overhead of message passing required. Now, suppose that there exists just one neighbor of i, say i', which lies outside C. Then, if μ' is the number of links of i' to the neighbors of i inside C, in the worst scenario, we need $m\mu'$ more computations (for the paths formed by i' and the neighbors of i, which have not been computed). In this case, the overall complexity would be

TABLE 2. Statistic properties of the datasets for the five social networks selected for experiments.

Network	Num. of Nodes	Num. of Edges	Num. of Triangles (longest shortest path)	Diameter	Average local clustering coefficient (*alcc*)
Facebook	4,039	88,234	1.612.010	8	0.6055
Twitter	81,306	1,768,149	13.082.506	7	0.5653
Google+	107,614	13,673,453	1.073.677.742	6	0.4901
Pokec	1,632,803	30,622,564	32.557.458	11	0.1094
LiveJournal	4,847,571	68,993,773	285.730.264	16	0.2742

$O(m)+\Delta+m\mu'$. Without loss of generality, let us assume that each neighbor of i outside C has, on the average, M links to the neighbors of i that lie in C. Then, the overall complexity would be $lcc[O(m)+\Delta]+(1-lcc)(mM)$. Consequently, the overall complexity decreases as $lcc \to 1$ and increases as $lcc \to 0$.

It must be noted that a network update may also include the case that some nodes are eliminated from the network. This scenario requires careful removal of nodes from the trees and it is a matter of future research.

V. EXPERIMENTAL RESULTS AND DISCUSSION

The community detection algorithm was implemented using an object-oriented environment, where each node is implemented as an object where its adjacencies, memberships and neighborhood (in the form of a threaded tree) are stored. As it can be clear from the description of the previous section, all the nodes can be assigned to multiple processors. Data is exchanged between processing elements as required (for example, in network updating). In this sense, the processor exchange messages (message passing) that contain data computed for a node. For our simulation environment, we used an Intel Core i7-8559U Processor system, with clock speed at 2.7GHz, equipped with four cores and eight threads/core, for a total of 32 logical processors.

To evaluate the performance of our proposed scheme, we used five real-world datasets: ego-Facebook, ego-Gplus, ego-Twitter, Pokec and Livejournal. Pokec is the most popular on-line social network in Slovakia. The popularity of this network remains high despite the advent of Facebook. Pokec connects more than 1.6 million people. LiveJournal is an online community with almost 10,000,000 quite active members, in which users maintain journals, and blogs. It allows people to declare friendship with other network members. The data is available online at [35]. Our algorithm will use this data, in search of *social circles* or *social lists*[1] These terms refer to mechanisms employed by the users to organize their networks and the data generated by them. McAuley and Lescovec [32] suggested three properties for circle formation: (1) The nodes within a circle should share some common properties, likes, opinions etc., (2) Completely different circles are formed by completely different properties, likes, opinions etc., and (3) Circles do overlap, that is "stronger" circles can be formed within weaker ones, in the sense that stronger paths can be formed within weaker ones in the algorithm presented in the previous section.

[1]The first term is used in Google+, while the second is used in Facebook and Twitter, but they practically mean the same thing.

A. DESCRIPTION OF DATA USED AND ADAPTATION TO THE PROPOSED SCHEME

Table 2 presents the basic statistic properties of the three networks being used. The triangles are basic structural properties of social networks and they represent a strong relationship between three nodes. Our algorithm handles the triangles using the left thread value of the binary tree. Dense communities generally tend to have large number of triangles. The diameter values show the number of nodes that should be traversed to travel from one vertex to another, excluding backtracks and loops. The value of average local clustering coefficient (*alcc*) is the average *lcc* for all the nodes of the network.

For each network, a combination of features describing the users was connected. The data, as presented in [35] were not in a proper form for the purpose of our algorithm, so we had to make some kind of adjustment, to adapt the data to the input demands of our scheme. Here, we describe the transformations we made for the ego-Facebook network. For the other networks, ego-Gplus and Twitter, we worked similarly. For each user examined (called *ego*), a set of combined features has been created. This set includes all the combined features possessed by *at least* two users, with whom the ego is related. This means that attributes owned only by one person related to the ego (for example, rare first names, or rare ages like 90+) tend to disappear. In [35], one can find data for a total of 26 attribute categories, including hometowns, education, birthdays, political affiliations, schools, etc and for different egos, the set includes different number of combined features. Table 3 shows 10 of these combined features, collected from the users related to with ID = 0. For these users, there are 224 different combined attributes (shown in file named 0.featnames). Also, note that the attributes are encoded as integer values, to maintain users' privacy. In Table 3, the attribute with ID = 0 corresponds to an age (apparently, one can find many more age attribute IDs, that, taken together, describe the range of ages of the users related to ego with ID = 0), the attribute with ID = 21 describes a combination of education and degree (again there are many combinations of education types and degrees with different IDs), etc.

Each ego forms circles with a set of nodes, based on the attribute values. These circles will be used as the

TABLE 3. Some combined features of users related to ego with ID=0, as shown in [35].

Attribute ID	Combination	Attribute Description
0	*birthday;anonymized feature 0*	A birthday value
21	*education;degree;id;anonymized feature 21*	Education and degree (For example, University, Informatics)
30	*education;school;id;anonymized feature 30*	Education and school user graduated from
53	*education;type;anonymized feature 53*	Education and type (for example High education and college)
72	*education;year;id;anonymized feature 72*	Education and year or period of years
77	*gender;anonymized feature 77*	Male or female (see Attribute ID=88 also)
78	*gender;anonymized feature 78*	Male or female (see Attribute ID=87 also)
79	*hometown;id;anonymized feature 79*	User's hometown
100	*languages;id;anonymized feature 100*	Languages spoken
140	*work;employer;id;anonymized feature 140*	Current work position, employer (for example programmer, IBM)

TABLE 4. Bit_vectors for the subset of attributes of Table 3.

Node ID	Bit_vector values
EGO=0	[0 0 0 1 0 0 1 0 0 0]
138	[0 0 0 0 0 0 1 0 0 0]
131	[0 0 0 0 0 0 1 0 0 0]
68	[0 0 0 0 0 1 0 1 0 0]
143	[0 0 0 1 0 1 0 0 0 0]
86	[0 0 0 0 0 1 0 0 0 0]

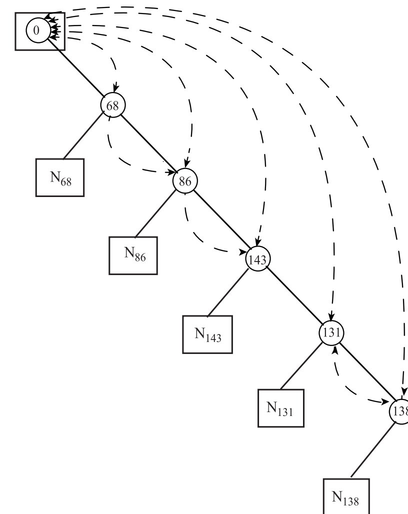

FIGURE 5. Threaded binary tree for ego node 0, based on the data of Table 4.

initial communities for our algorithm. First, we give a circle example and then we will formally describe how to transform the given data to produce our initial communities in the form of weighted graphs. Based on the data given in [35], circle_13 of ego with ID = 0 includes the node ID's (or users) 138, 131, 68, 143, 86. The binary values at positions 0,21,30,53,72,77,78,79, 100, and 140 of their bit_vectors are given in Table 4. From these values, it follows that, taking into consideration the 10 attributes of Table 3, the similarities $w_{i,j}$ between each pair of nodes (i, j) can be computed by taking the bitwise Exclusive-OR (XOR) of their corresponding bit_vectors I, J into bit_vector W, counting the number of 1s resulted, and divide this number by the total number of attributes examined, in this case, 10. Mathematically:

$$w_{i,j} = \frac{\text{Num. of 1s in vector } W}{\text{Num. of Attributes}}, \quad \text{where } W = I \oplus J \qquad (8)$$

By implementing (7), we get the following similarity values between the pairs of nodes of the circle:

$$w_{0,138} = 0.9, \quad w_{0,131} = 0.9, \ w_{0,68} = 0.6$$
$$w_{0,143} = 0.8 \quad w_{0,86} = 0.7, \ w_{138,131} = 1$$
$$w_{138,68} = 0.7, \quad w_{138,143} = 0.7, \ w_{138,86} = 0.8$$
$$w_{131,68} = 0.7, \quad w_{131,143} = 0.7, \ w_{131,86} = 0.8$$
$$w_{68,143} = 0.8, \quad w_{68,86} = 0.9, \ w_{143,86} = 0.9$$

and the threaded binary tree of the ego node 0 will be formed as shown in Fig. 2 and described in Algorithm 1 (see Fig. 4). Similar threaded trees can be constructed for all the other nodes. The double arrow found in the left thread of nodes 138 and 131 indicates that 138 is the left thread of 131 and 131 is the left thread of 138, based on the similarity values computed above.

For the simulations described in the next paragraphs, we used the circles found in [35] for given egos as our initial communities and then we kept adding new nodes to form new communities, examine a given ego's membership to these communities and spot community overlaps (circles within circles or, in our terminology, paths within paths).

B. EXPERIMENTS AND PERFORMANCE EVALUATION ISSUES

In this paragraph, we present our experimental results per phase of the algorithm proposed. The experiments of phase A and B can be used to measure the strong scaling of our scheme, while phase C (as we describe) is used to measure its weak scaling. In the end, we compare the scaling of our scheme to the scaling of other state-of-the-art methods, like PLMR and the recently introduced picaso scheme.

1) PHASE A: THREADED BINARY TREE GENERATION

The first phase of our scheme uses two parallel iterators. The first iterator, SORT, performs sorting in a pipeline fashion. Since its complexity depends on the number of neighbors per node, it requires only a few milliseconds. An almost perfect speedup is achieved for small networks, like the Facebook network with only about 4,000 nodes.

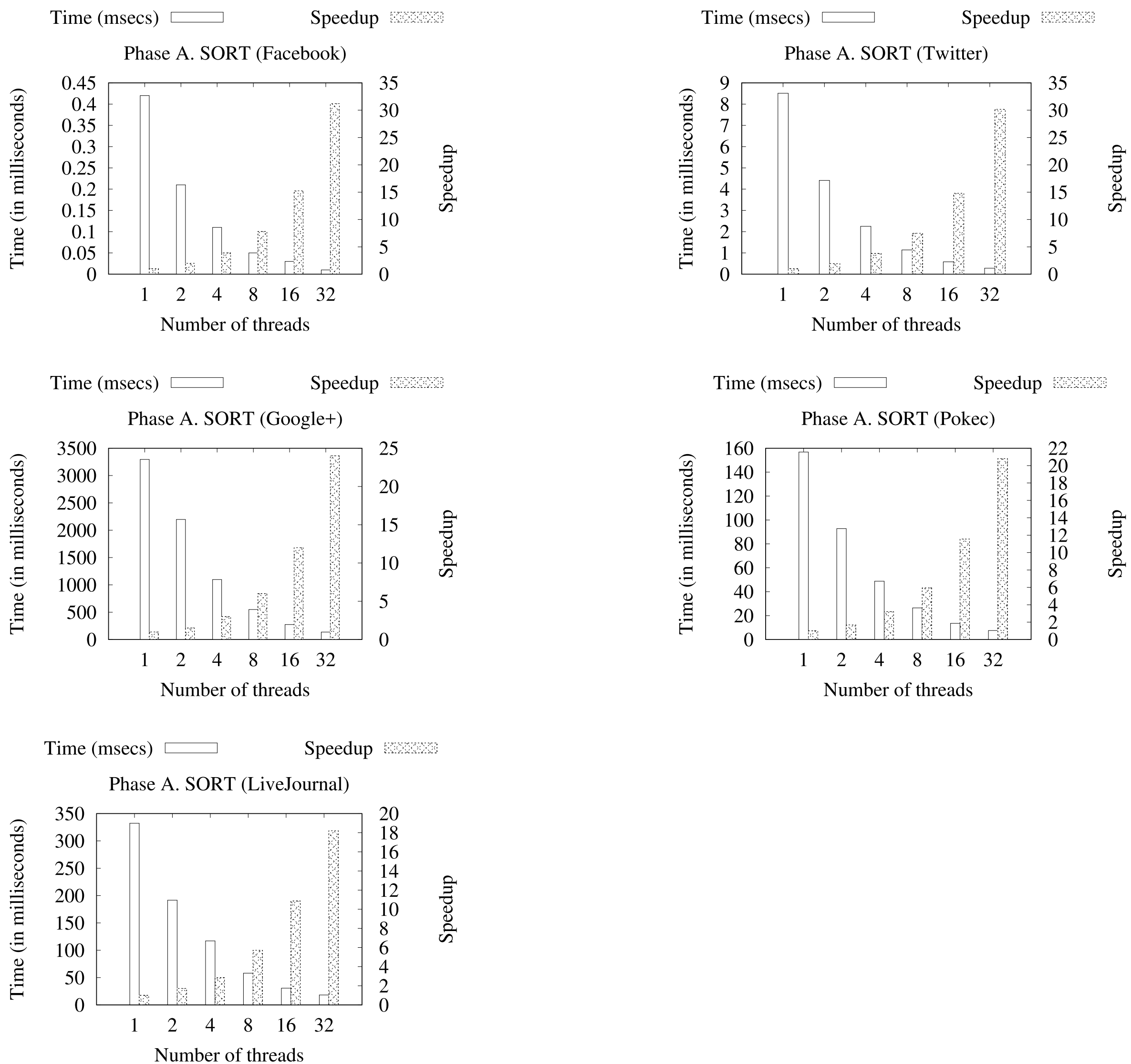

FIGURE 6. Phase A -SORT- strong scaling behavior.

However, as the networks become larger and more communications (thus overheads) are required between processors, the speedup reduces, and for the biggest network (LiveJournal), we only achieve a speedup of 18.2 when working with 32 processing elements. Also, note that the Google+ network, although it has almost the same number of nodes as Twitter, almost 15 times fewer nodes than Pokec and 45 times fewer nodes than LiveJournal, the time required to complete its SORT is about 2.5-3 times less than the time required for Pokec (depending on the number of threads used) and about 5-10 times less than the corresponding time for LiveJournal (again, depending on the number of threads used). This is explained by the very large numbers of neighbors per node, found in Google+ (the network has a total of 13,673,453 edges). Fig. 6 presents the strong scaling behavior of the SORT procedure of Phase A.

The second iterator of Phase A, TBTGEN, generates the threaded binary tree, by adding values to the left and right links and threads. In this case, the number of comparisons performed for the left link computations are actually quadratic per neighborhood, resulting in larger execution times. As the computational load increases, the effects of hyper-threading present themselves. In TBTGEN, the threads used are more memory-bound, in the sense that more writings to memory are required to store the intermediate values. This fact decreases the overall performance, however we can still achieve a speedup of 28.80 for the very small Facebook network and 16.96 for the larger LiveJournal due to the relatively low complexity of TBTGEN (square of number of neighbors). Fig. 7 presents the strong scaling behavior of the TBTGEN procedure of Phase A.

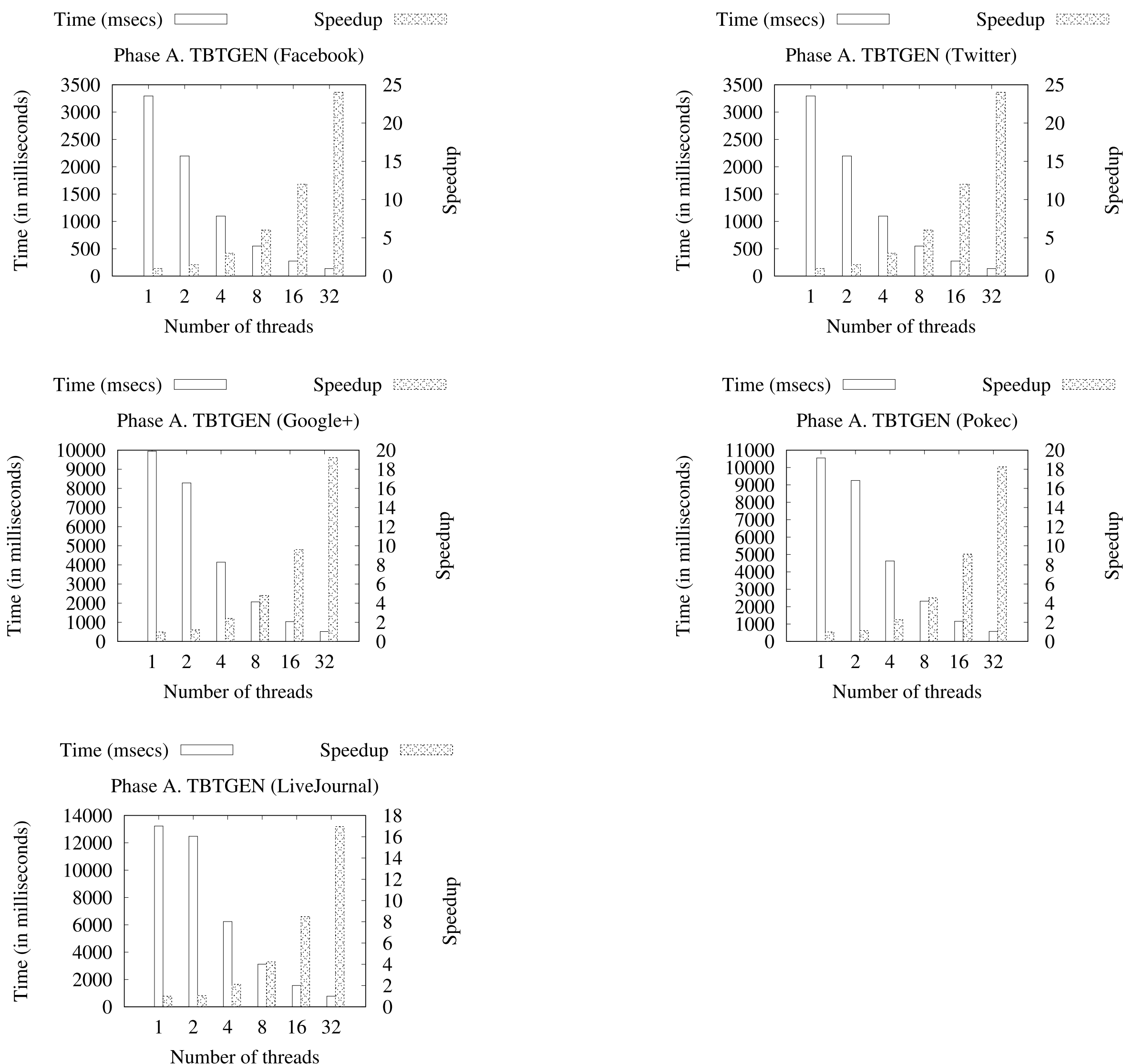

FIGURE 7. Phase A -TBTGEN- strong scaling behavior.

2) PHASE B: PATH ANALYSIS

The path analysis is the most computationally intensive and memory- demanding procedure of our scheme and the time required for completion is in order of seconds (except the very small Facebook network). As the number of nodes grows larger (for Pokec and LiveJournal) or in cases where the number of edges is too large (and each node has a very large number of neighbors) the time required to traverse the total of n trees enlarges. Moreover, the presence of eight threads per processor lowers the speedup. For LiveJournal, we only obtain a speedup of 12.8, meaning that the maximum attainable speedup of 32 is reduced by 60%. It is also remarkable, that even for small networks with small number of edges and nodes, like ego-Facebook, we note a reduction of 15% (the speedup for this network is 27.2) from the maximum speedup of 32. Figure 8 presents the strong scaling behavior of the path analysis phase, for the five networks examined.

Figure 9 compares the accumulative running times (Fig. 9a) and the accumulative speedups (Fig. 9b) for the five networks. Note that, for one or two processors, the execution of Phases A and B require longer time for the relatively small Google+ network compared to the almost eight time larger Pokec. The very large number of neighbors found in Google+ is the main reason behind this behavior, and it is remarkable that even with more processors and threads, the execution times are very close.

3) PHASE C: NETWORK UPDATING

In this set of experiments, we choose one of the networks used for our experiments, to study the weak scaling of the proposed scheme. We chose the Google+ network, because of its interesting structure, that comprises a relatively small number of nodes but disproportionate number of neighbors per node. To perform weak scaling, we had to execute our

FIGURE 8. Phase B, strong scaling behavior.

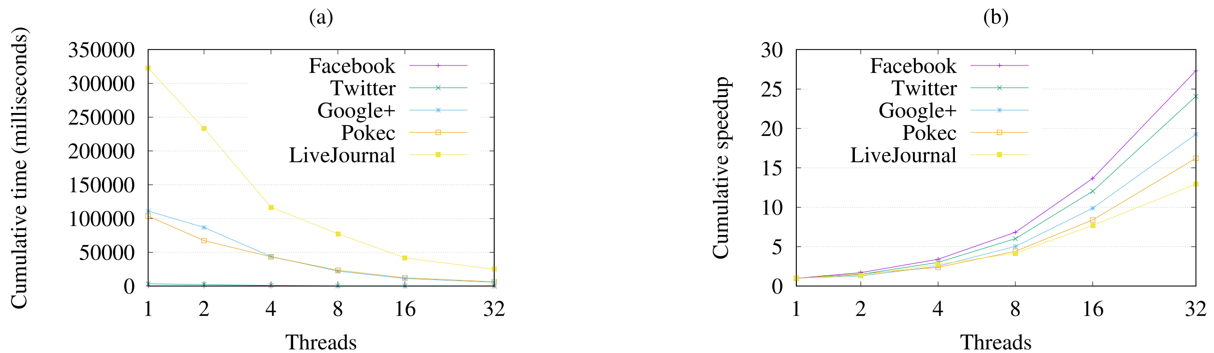

FIGURE 9. Phases A and B, cumulative run times and speedups for the 5 networks.

algorithm on a single core and then keep doubling the number of network nodes while simultaneously doubling the number of processing nodes. To add new nodes, we generated random bit_vectors like the ones in Table 4. One factor that played major role in this type of experiment was the use of the *alcc* value, which determines the ratio of the

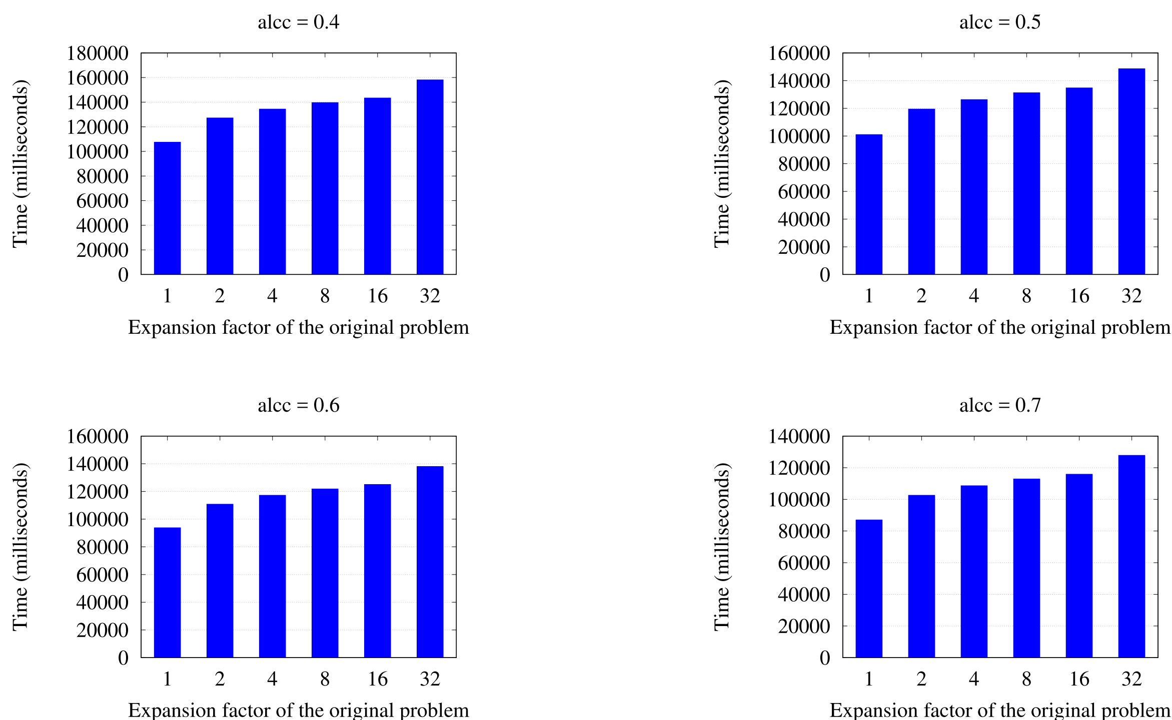

FIGURE 10. Weak scaling for the Google+ network, for four different *alcc* values.

links internal to its community to its total links(internal and external).

To perform updating, we add groups of double the size of the original network. Our scheme tries to place each new node in community or communities by locating stronger paths originating from the new node to each community examined. The requirement for a node's membership to a community was that 50%, of the paths examined were found to satisfy (4) and (5). For our experiments, the *alcc* value ranged between 0.4 (its value for the Google+ dataset is $\approx$0.5) and 0.7. Thus, each new-coming node has *alcc* of its nodes inside a community and another $1-alcc$ spread to other communities, $alcc \in [0 \ldots 1]$. Figure 10 shows the weak scaling for the Google+ dataset. The expansion factors are shown in the horizontal axis. The original problem (expansion factor = 1) had about 100K (107,614) nodes. The larger problem (expansion factor = 32) had about 3M nodes and an average of 127 neighbors per node. Note that, the total execution time increases every time the problem size and the processing elements are doubled. This is normal; since, each time the processing elements are doubled, overheads incur due to hyperthreading.

In the last set of experiments, we show the effect of *alcc* to the number of communities detected. The experiments concern the Google+ network expansion, from 100K to 3M nodes. For low *alcc* values (average up to 50%), the algorithm detects more newly formed communities than overlaps. This is an expected behavior, since we have chosen a strict approach for overlapping and every node has more of its links connected to nodes outside its community. As *alcc* enlarges, more overlaps than new communities are detected and the total execution time decreases, as explained in Section IV-C. For example, when the expansion factor was 32 and $alcc = 0.7$, about 490K communities were detected, where 70% (about 340K) were overlaps while only 30% (about 150K) were newly formed communities. Figure 11 shows the effects of increasing the *alcc* to the number and nature of communities detected.

4) COMPARISON RESULTS

In this paragraph we compare the scaling of our algorithm to two other well-known strategies: The PLMR method [6], and the picaso method [15]. We conducted two sets of experiments. In the first set, we compare the aggregated speedup for each phase of our scheme to the aggregated speedup for each of the phases of the PLMR method. In the second set, we perform speedup comparisons between our strategy, the PLMR and the picaso method, on the Livejournal dataset.

Similarly to the proposed scheme, the Parallel Louvain Method with Refinement (PLMR) also includes three phases and, based to the functionality, we can say that there is some correspondence between the phases of the two schemes. In its first phase, the LPMR algorithm moves the nodes to neighboring communities until the modularity is maximized. This phase generates stable communities, just like our scheme generates neighbors for each node, in the form of a threaded binary tree. In its coarsening phase, the PLMR method forms communities of communities by recursively

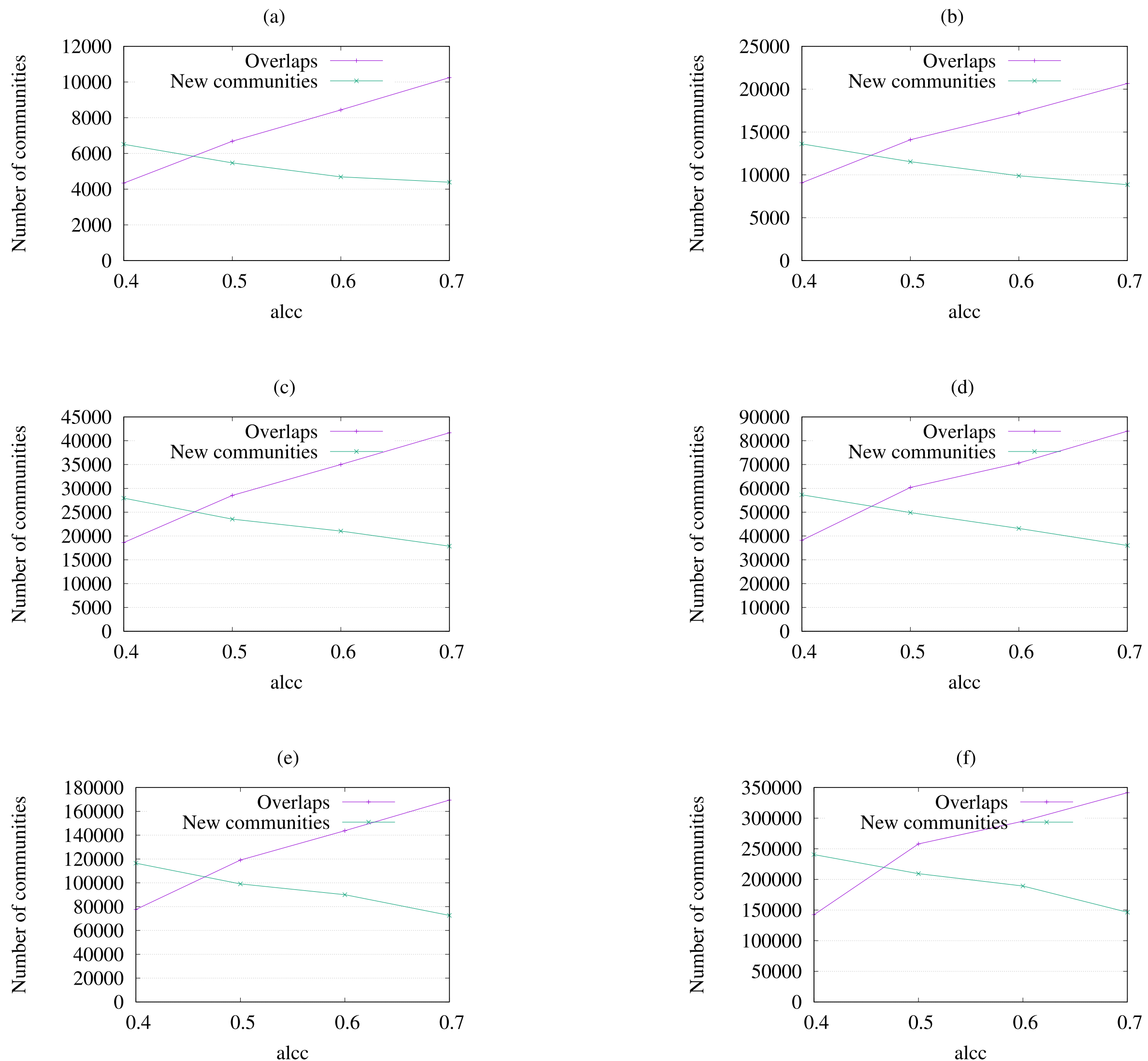

FIGURE 11. Number of new communities and overlaps detected for four different *alcc* values.

contracting each community to a certain supernode, just like our scheme detects communities and overlaps by using the path analysis. In its last phase, the PLMR method reevaluates the communities based on the network changes made during the last coarsening phase. This corresponds to our update method.

From the aggregated results derived, the move and the update (refinement) phase of the PLMR scheme present the largest aggregated speedups, while the second phase (coarsening) is the most computationally intensive. As shown in the experiments presented in the previous paragraphs, our scheme also has the same behavior: the first and third phases scale better compared to the phase that detects the communities and their overlaps. Fig. 12 compares the first phase of the two strategies. The pipeline based organization of our scheme offers good scaling performance to our SORT iterator and the TBT_GEN strategy is only based on the number of neighbors per node (not on the total number of nodes), so our first phase naturally scales better compared to the corresponding phase of PLMR. The second phase our scheme does not scale as well as the other two phases but the experiments conducted the PLMR method indicate that this scheme dos not have very large gains due to parallelism. This explains the large speedup difference (see Fig. 13) between the these corresponding phases of the two strategies. The third phase of our strategy scales better for networks with larger *alcc*, like Facebook, Twitter or Google plus. Its performance degrades for networks like LiveJournal or Pokec. Because the number of computations required for our update phase is reduced (computations from the second phase are used to a big or small extent, depending on the *alcc*), this phase outperforms the third phase of the PLMR strategy, which is based on the changes occurring the coarsening phase (see Fig. 14). Apparently, there are cases when the changes in

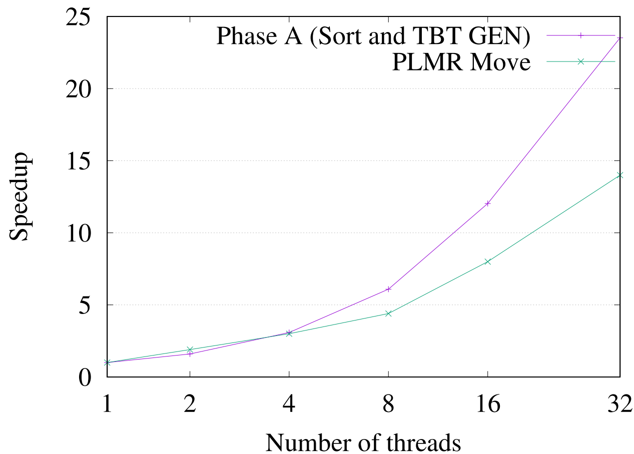

FIGURE 12. Speedup comparisons between the first phase of our scheme and the first phase of the PLMR scheme.

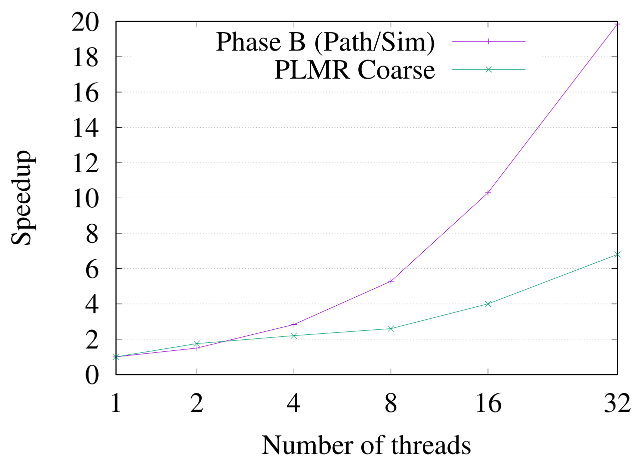

FIGURE 13. Speedup comparisons between the second phase of our scheme and the second phase of the PLMR scheme.

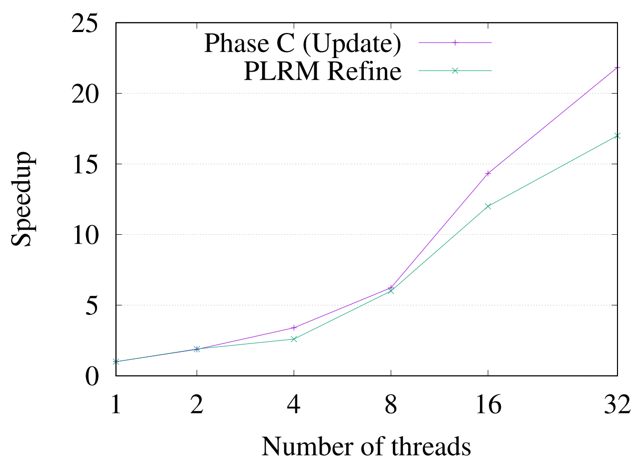

FIGURE 14. Speedup comparisons between the third phase of our scheme and the third phase of the PLMR scheme.

the coarsening phase are such that the gains from parallelism are reduced.

We acknowledge that, to have an even more clear view of how the two schemes (and their corresponding phases) compare, we need to perform more comparisons on even

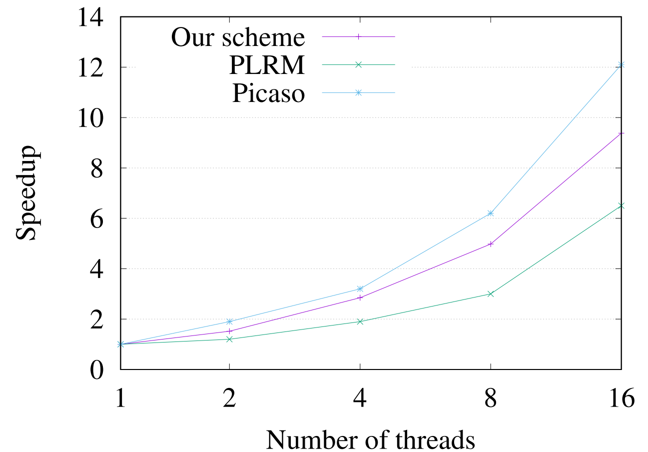

FIGURE 15. Speedup comparisons between our strategy, the PLMR and the picaso method.

larger networks. For example, the Uk-2007-05 network contains about 100 million nodes, or the uk-2002 contains about 18 million nodes. Experiments on such graphs are expected to reduce the speedups of the first two phases of our strategy, but increase the speedup of the update phase, as they have very large *alccs* (0.74 and 0.69, respectively).

In the last set of experiments, we compare the speedup of our work, to the speedups of picaso [15] and PLMR on the LiveJournal network (see Fig. 15). The results indicate that, while our scheme outperforms the PLMR, the picaso strategy seems to scale better. Partially, this result can be explained by the low *alcc* of the LiveJournal network, which is 0.27, meaning that our update phase is not so well-scaled as in other networks, reducing the overall parallelism gains. Again, more comparisons on different networks are required, using equally configured platforms to fairly compare the accepted schemes (for example, our scheme and PLMR use logical processors and thus, they also suffer some costs due to hyperthreading, unlike the picaso method, which is implemented on a set of 16 physical processors). However, these first comparisons give a good indication of good performance of the proposed scheme. Also, more experiments can aid in spotting and correcting possible weaknesses of the threaded binary tree based strategy.

VI. CONCLUSIONS-FUTURE WORK

This paper presented a threaded binary tree approach for community detection. To determine a user's membership to a community, the method tries to locate "stronger" paths (with higher similarity) between the user's node and this community. We used the real network data, which are freely available from Stanford university, to test the performance of our scheme. Strong scaling results were obtained for the first two phases and showed that our scheme benefits from parallelism, however the speedup decreases as computations/communications become intensive. Weak scaling results indicate that the total execution time increases by enlarging the problem size and processing elements analogously. Also, as *alcc* $\rightarrow$ 1 the number of detected overlaps

increases and the algorithm scales better. Finally, more new communities are detected as *alcc* → 0, at the expense of lower scaling behavior.

Having tested the basic parallelism issues of our work, we compared our scheme with two parallel community detection schemes, PLMR and picaso. The comparison results indicate good scaling performance of our parallel scheme, but they also show that more experiments are required. In our future work, we plan to perform extensive experiments on more and even larger networks, using similar system configurations. For this purpose, we need more comparable results, especially from parallel schemes that also detect overlaps, and a tool that implements parallel graph algorithms, like *NetworKit* ([6]). Also, we need to conduct some distributed experiments to test the possibility of scaling up to big data social networks. This may require a number of design considerations. Moreover, there is a wide variety of other aspects to be further examined: First, we plan to expand the tree structure, to include multiple threads, from each node to other, non-neighboring parts of the network. In this way, we will highly reduce the time required for path detection. Then, we are planning to work on the use of other data structures such as ternary or higher degree trees, to represent the entire network of nodes. The case of updating the network by *removing* some nodes needs also to be considered. Finally, we have to try and utilize GPU acceleration to further improve our scheme's performance.

REFERENCES

[1] (2017). *Statista*. Accessed: Jun. 4th, 2017. [Online]. Available: http://www.statista.com/statistics/272014/global-social-networks-rankedby-number-of-users

[2] R. Sharma and S. Oliveira, "Community detection algorithm for big social networks using hybrid architecture," *Big Data Res.*, vol. 10, pp. 44–52, Dec. 2017.

[3] Z. Bu, C. Zhang, Z. Xia, and J. Wang, "A fast parallel modularity optimization algorithm (FPMQA) for community detection in online social network," *Knowl.-Based Syst.*, vol. 50, pp. 246–259, Sep. 2013.

[4] S. Fortunato, "Community detection in graphs," *Phys. Rep.*, vol. 486, nos. 3–5, pp. 75–174, 2010.

[5] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 76, no. 3, p. 036106, 2007.

[6] C. L. Staudt and H. Meyerhenke, "Engineering parallel algorithms for community detection in massive networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 171–184, Jan. 2016.

[7] M. E. J. Newman, "Detecting community structure in networks," *Eur. Phys. J. B*, vol. 38, no. 2, pp. 321–330, Mar. 2004.

[8] M. Wang, C. Wang, J. X. Yu, and J. Zhang, "Community detection in social networks: An in-depth benchmarking study with a procedure-oriented framework," *VLDB Endowment*, vol. 8, no. 10, pp. 998–1009, 2015.

[9] S. I. Souravlas and A. Sifaleras, "On the detection of overlapped network communities via weight redistributions," in *Proc. Adv. Experim. Med. Biol.* Dordrecht, The Netherlands: Springer, 2017, pp. 205–214.

[10] S. Souravlas, A. Sifaleras, and S. Katsavounis, "A novel, interdisciplinary, approach for community detection based on remote file requests," *IEEE Access*, vol. 6, no. 1, pp. 68415–68428, 2018.

[11] A. Prat-Pérez, D. Dominguez-Sal, J. M. Brunat, and J.-L. Larriba-Pey, "Shaping communities out of triangles," in *Proc. CIKM*, Oct. 2012, pp. 1677–1681.

[12] F. Zhao and A. K. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *VLDB Endowment*, vol. 6, no. 2, pp. 85–96, Dec. 2012.

[13] D. Chen, M. Shang, Z. Lv, and Y. Fu, "Detecting overlapping communities of weighted networks via a local algorithm," *Phys. A, Stat. Mech. Appl.*, vol. 389, no. 19, pp. 4177–4187, Oct. 2010.

[14] Z. Lu, X. Sun, Y. Wen, G. Cao, and T. La Porta, "Algorithms and applications for community detection in weighted networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 2916–2926, Nov. 2015.

[15] S. Qiao *et al.*, "A fast parallel community discovery model on complex networks through approximate optimization," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1638–1651, Sep. 2018.

[16] M. C. V. Nascimento and L. Pitsoulis, "Community detection by modularity maximization using GRASP with path relinking," *Comput. Oper. Res.*, vol. 40, no. 12, pp. 3121–3131, Dec. 2013.

[17] D. Džamić, D. Aloise, and N. Mladenović, "Ascent–descent variable neighborhood decomposition search for community detection by modularity maximization," *Ann. Oper. Res.*, vol. 272, nos. 1–2, pp. 273–287, Jan. 2019.

[18] R. Santiago and L. C. Lamb, "Efficient modularity density heuristics for large graphs," *Eur. J. Oper. Res.*, vol. 258, no. 3, pp. 844–865, May 2017.

[19] M. E. J. Newman, "Modularity and community structure in networks," *Nat. Acad. Sci. United States Amer.*, vol. 103, no. 23, pp. 8577–8582, 2006.

[20] I. Farkas and D. Ábel, G. Palla, and T. Vicsek, "Weighted network modules," *New J. Phys.*, vol. 9, no. 6, p. 180, Jun. 2007.

[21] J. M. Kumpula and M. Kivelä, K. Kaski, and J. Saramáki, "Sequential algorithm for fast clique percolation," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 78, no. 2, Aug. 2008, Art. no. 026109.

[22] X. Zhang, C. Wang, Y. Su, L. Pan, and H.-F. Zhang, "A fast overlapping community detection algorithm based on weak cliques for large-scale networks," *IEEE Trans. Comput. Social Syst.*, vol. 4, no. 5, pp. 218–230, Dec. 2017.

[23] W. Zhi-Xiao, L. Ze-chao, D. Xiao-fang, and T. Jin-hui, "Overlapping community detection based on node location analysis," *Knowl.-Based Syst.*, vol. 105, pp. 225–235, Aug. 2016.

[24] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large network," *J. Stat. Mech. Theory Exp.*, vol. 2008, no. 10, Oct. 2008, Art. no. P10008.

[25] S. Gregory, "Finding overlapping communities in networks by label propagation," *New J. Phys.*, vol. 12, no. 10, Oct. 2010, Art. no. 103018.

[26] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann, "Link communities reveal multiscale complexity in networks," *Nature*, vol. 466, no. 7307, pp. 761–764, 2010. [Online]. Available: http://dx.doi.org/10.1038/nature09182

[27] A. Padrol-Sureda, G. Perarnau-Llobet, J. Pfeifle, and V. Muntës-Mulero, "Overlapping community search for social networks," in *Proc. 26th IEEE Int. Conf. Data Eng. (ICDE)*, Mar. 2010, pp. 992–995.

[28] A. Clauset, M. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 70, no. 6, pp. 66–111, Dec. 2004.

[29] J. Chen and Y. Saad, "Dense subgraph extraction with application to community detection," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 7, pp. 1216–1230, Jul. 2012.

[30] J. Huang, H. Sun, Q. Song, H. Deng, and J. Han, "Revealing density-based clustering structure from the core-connected tree of a network," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 8, pp. 1876–1889, Aug. 2013.

[31] S. Souravlas and A. Sifaleras, "Efficient community-based data distribution over multicast trees," *IEEE Trans. Comput. Social Syst.*, vol. 5, no. 1, pp. 229–243, Mar. 2018.

[32] J. Leskovec and J. J. Mcauley, "Learning to discover social circles in ego networks," in *Proc. Adv. Neural Inf. Process. Syst.*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Red Hook, NY, USA, Curran Associates, Inc., 2012, pp. 539–547.

[33] D. Knuth, *The Art Computer Programming*, vol. 1. London, U.K.: Pearson, 1997.

[34] R. Zhang, L. Li, C. Bao, L. Zhou, and B. Kong, "The community detection algorithm based on the node clustering coefficient and the edge clustering coefficient," in *Proc. 11th World Congr. Intell. Control Automat.*, Jul. 2014, pp. 3240–3245.

[35] J. Leskovec and A. Krevl, "SNAP datasets: Stanford largenetwork dataset collection," Univ. Stanford, Stanford, CA, USA, Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

STAVROS SOURAVLAS has joined the Department of Applied Informatics, School of Information Sciences, University of Macedonia, in 2014, where he is currently an Assistant Professor of computer architecture and digital logic design. His research interests include computer architecture and performance evaluation, parallel and distributed systems, grid computing, cloud computing, and systems modeling and simulation. He is a member of the IEEE.

ANGELO SIFALERAS is currently an Associate Professor with the Department of Applied Informatics, School of Information Sciences, University of Macedonia, Thessaloniki, Greece. His research focuses on mathematical programming and network optimization. He is also a Senior Member of the ACM.

STEFANOS KATSAVOUNIS is currently an Associate Professor with the Department of Production Engineering and Management, Democritus University of Thrace, Greece. His scientific interests revolve around scheduling, RCMPSP, project management, graph theory and modeling, and heuristics for NP-hard problems in transportation and supply chain management, grey analysis, and data processing in material science.