Received February 23, 2020, accepted March 13, 2020, date of publication March 20, 2020, date of current version April 1, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2982227

Hybrid CPU-GPU Community Detection in Weighted Networks

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

[1]Department of Applied Informatics, University of Macedonia, 54636 Thessaloniki, Greece
[2]Department of Production and Management Engineering, Democritus University of Thrace, 671 00 Xanthi, Greece

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

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

ABSTRACT Recently, a new trend has emerged in the field of parallel and high performance computing, the hybrid implementation using CPU-GPU modules. In such implementations, the computational load is shared between the CPU and GPU, in order to improve the computational efficiency. However, the task of sharing the computational load between the two modules is a rather difficult one, with a number of limitations being imposed. This paper extends our recent work on community detection, which is based on transforming a network of nodes into a set of threaded binary trees. In this work, we share the computational load between the two units: the CPU takes specific samples of the network communities and organizes them in the form of threaded binary trees. The GPU takes over the heavy load of reading this data and transforming it into a path-matrix. Finally, this matrix is sent back to the CPU for analysis, community detection and overlaps, as well as network information upgrades. Our simulation results show significant improvement over our previous strategy and other known community detection strategies found in the literature.

INDEX TERMS Community detection, parallel algorithms, binary trees, social circles, GPU-CPU scheduling.

I. INTRODUCTION

Several synchronous applications are based on graph-structured data. A very important application of this kind is *community detection*. The community detection problem refers to finding community structures, that is, sets of "strongly" related nodes. The analysis of modern social networks becomes rather cumbersome, as their size and number keeps growing larger and larger. In this sense, some level of parallelism should necessarily be employed, to reduce the computational costs. However, it has been reported in several works (for example, see [2], [3]) that even well-structured parallelized strategies suffer from low speedups and high execution times. Moreover, due to their irregular topologies and their frequent changes, the processing of these networks becomes even more difficult and more computational efforts are required to accommodate these changes. This work aims at taking advantage of the parallel processing capabilities of the GPU [4], in order to accelerate computations, keeping in mind though, that the GPU has different performance characteristics compared to the CPU and that scheduling should be implemented in such a way that not much synchronization between the two modules should be needed, as this would cause large overheads. Another idea that can improve performance is to keep busy the maximum possible number of the GPU threads.

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

Research on community detection is highly motivated by the fact that the traditional community detection algorithms fail to scale with the increasing number of users and the number and complexity of their relationships. More computationally efficient algorithms are now necessary to analyze and even predict the user's behavior, which is of great importance for organizations and companies, or even for political parties. Generally, community detection strategies have to overcome three very important problems:

1. The *increasing size* of social networks leading to huge data volumes that need to be processed.
2. The *irregular topologies* which cause load imbalances when a set of multiple processors is used for the computations (for example, one processor may process a node

with thousands of neighbors while another processes a node with just a few).

3. The *network updates* are quite often, as new users enter a community and others leave. This causes changes to the irregular topologies mentioned just above, which pose another burden in community detection.

This work extends our previous parallel community detection strategy [1], which introduced the use of threaded binary trees for community detection. One of the issues raised in this work was whether it is possible to implement the proposed algorithm or parts of it in the GPU and if such an implementation could improve the computational efficiency. Although it is known that the GPUs provide higher gains when operating on problems with high B/F (bytes per flop), like computationally intensive matrix multiplications or Fast Fourier Transformations (FFT), there have been a lot of recent efforts to exploit the parallelism benefits they offer on irregular problems, with generally unpredictable, input-dependent computations and memory accesses. Tree traversals are such an example. (for example, see [5], [6]). This work addresses the aforementioned issues of community detection strategies and redesigns our previous work, in such a manner that a part of the overall computational load is performed on the GPU. To do so, some basic design principles need to be followed:

1. To take full advantage of parallelism using a GPU, the executing threads should avoid control divergence like branches. Ideally, all threads in a warp (group of threads) should execute similar instructions simultaneously, on different data.
2. Memory divergences should be avoided that is, consecutive threads should read consecutive data bytes, so as to satisfy the memory reads with the minimum number of memory transactions. In other words, the memory requests should be predictable and involve aligned and continuous memory positions. This would reduce the total memory costs.
3. Little or no communication between the executing threads is required, in order to explore the GPU parallelism capabilities. If this is not the case, communication overheads may occur.
4. Ideally, when large number of threads are executed on the GPU, the GPU will reach its maximum performance.

These principles will be further discussed when we present our GPU/CPU scheme. The remainder of this work is organized as follows: Section II describes the related work. Section III gives a very brief theoretical background of the community detection problem and presents the criteria posed in our strategy to detect memberships and overlaps. More details can be found in [1]. Section IV describes our community detection algorithm and the CPU-GPU scheduling strategy, taking into account the four design principles presented above. 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 [7]. 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* [8]. The nodes of a community are considered similar to each other, dissimilar to the other nodes of the network [9] 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. 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.

Although there has been some work on disjoined communities (a good example is the work of Staudt and Meyerhenke in [3]), the majority of the latest algorithms study the problem of *overlapped communities*. The main reason behind such 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* [10]. 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 [9]. 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 [11]–[13]. 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. Researchers have to deal with dynamic graphs, where nodes can be added or removed at a time. 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 approaches, which start from the graph representing the entire network and try to divide it into communities, (b) the bottom-up approaches that use the local structures and try to expand them to form communities, and (c) the data-structure-based approaches, that try to convert the entire network into a data structure, which is then processed to detect communities. In this section, we briefly mention the most representative papers from each category. A more detailed description can be found in our previous work ([1]).

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 detect communities. When the links connected to a node are found in more than one communities, this node is assumed to be overlapped. In [14], the authors 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. A similar triangle-based approach, called $k-$mutual-friend subgraph was also used in [15].

Chen *et al.* [16] 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. A similar approach is found in [17], 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*.

A newly introduced top-down strategy named picaso was introduced very recently by Qiao *et al.* [2]. 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. 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 optimization schemes 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. Optimization based examples are [18]–[20], and [21].

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 vertices are connected. Thus, connected vertices on the graph represent network communities. Interesting clique percolation techniques were introduced in [22], [23], and [24].

Label propagation is another interesting bottom-up approach. It 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. Interesting label propagation schemes were introduced in [9] and [25].

Finally, some other interesting ideas include the work of Zhi-Xiao *et al.* [26], which is based on the estimation of the nodes' mass and on spotting their location in the network and the Parallel Louvain Method with Refinement (PLMR) [3], which is based on the initial Parallel Louvain Method (PLM) introduced by Blondel *et al.* [27].

Generally, the bottom-up approaches have the advantage that, in many cases, their complexity is linear. 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 APPROACHES

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.* [28] 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) [29] 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.

Agglomeration algorithms build tree hierarchies starting from small clusters and expanding to larger ones. Clauset *et al.* [30] presented an algorithm that starts from single nodes and builds the dendrogram structure which describes the community structure, while maintaining the changes in modularity. A slightly different approach, *matrix blocking* [31] constructs an hierarchy tree by recognizing matrix column similarities between nodes. Then, partial clustering is computed in the graph. Finally, another interesting data structure approach is presented in [32]. The scheme is called graph-skeleton-based clustering (gSkeletonClu) and its idea is to project 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 conversion 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 and combines the CPU/GPU modules, to efficiently scan for communities and to reduce the computational costs.

III. COMMUNITY DETECTION PRELIMINARIES

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 $[0 \ldots 1]$. As will be described in the Experimental Results and Discussion section, this value is computed based on the real data collected for various networks.

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 $[0 \ldots 1]$. In a network representation, the nodes are labeled by boldfaced numbers, 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 in Fig.1. User **1** has three internal links, namely with users **10**, **2** and **3** and two external links, with users **5** and **6**. To compute $NCD_{\mathbf{1}}$, we only consider the internal links and we have $NCD_{\mathbf{1}} = \frac{(0.9+0.96+1)}{3} = 0.953$.

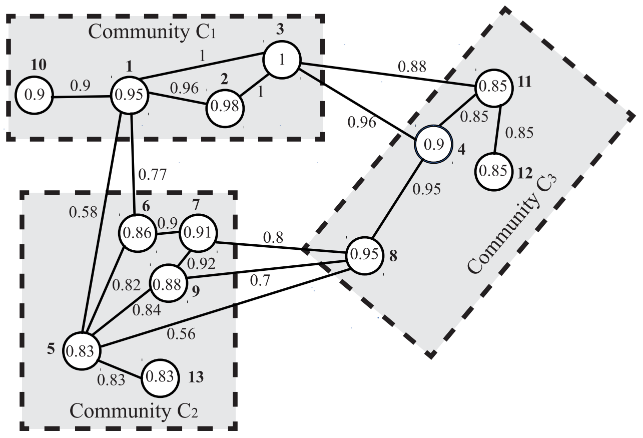


FIGURE 1. An irregular network with 3 communities.

Therefore, the *Average Community Connectivity* (ACC) for a community C (ACC_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 between i and j.
Apparently, a criterion to determine the membership of a node to a community, would be the detection of a path starting from this node, which is stronger than the ACC of this community.
2. The length of each path $\delta_{i,j}$ found to satisfy the first criterion should not exceed the *diameter d* of community C (d_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 **1, 2, 3,** and **10** divided by 4, that is, $\frac{0.9+0.95+0.98+1}{4} \approx 0.9575$ and the diameter of C_1 is 3. The path starting from node **8** to node **4** and then to node **3** has a strength of 0.955 and its length is 2. In this case, the double criterion is not satisfied, although the length of this path is smaller compared to the diameter of C_1, since its strength is smaller compared to the average connectivity of C_1. However, if the weight of the link connecting nodes **8** and **4** was 1, the double criterion would have been satisfied, and node **8** could have been considered as overlapping to community C_1.

IV. HYBRID GPU-CPU SCHEDULING

In this work, we introduce a hybrid CPU-GPU based scheme to detect communities on weighted graphs. The main idea behind our scheme is to detect possible overlaps between pairs of existing communities and keep expanding the process, using the newly formed communities. To examine if a node overlaps with a community, we seek for strong paths between this node and the members of the target community, with the restriction that the path length is at most equal to the diameter of the target community. The proposed community detection scheme operates as follows: we take samples of suitable size for GPU processing (this will be explained later on), from pairs of initially formed communities (in the Experimental Results section, we show how we can form samples of initial communities given real data) and then we transform these samples into threaded binary trees, using the CPU capabilities. These binary trees are passed to the GPU, where they are transformed into a bit-matrix form at a very high speed. This bit-matrix form is passed back to the CPU for path analysis, to determine overlaps. When new nodes enter the network, they are connected to already existing nodes, so they can be examined for possible overlaps using the paths already determined. This expansion phase is also implemented by the CPU.

As more and more nodes keep entering the network with only few connections to existing nodes, this overall process needs to be regularly repeated. Since our scheme detects overlaps and one node is not considered as a member of only one community, there may be cases where a pair of overlapped communities may have only a few nodes in common but very strong internal paths and high ACCs. In such a scenario, repeating the overall process may give us the chance to determine that, in fact, the two communities may be merged into a single one. The following paragraphs will describe and analyze the three phases of our proposed hybrid CPU-GPU strategy: a) threaded binary tree formulation, b) path formulation and analysis, and c) expansion phase.

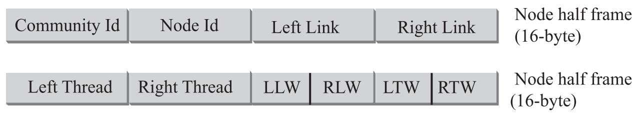


FIGURE 2. A frame describing a tree node.

A. THREADED BINARY TREE FORMULATION

In most of the cases, a tree structure is stored in the memory as an object (object-oriented programming) or as a link list. However, we need a form that will be useful for GPU processing, so we will prefer a structure of continuous array elements. Each tree node will be stored as a 32-byte frame, that is divided into the following fields (see Fig.2):

- *Community id:* A 4-byte containing information regarding the community each node belongs to.
- *Node id:* A 4-byte value, which stores the node id, typically an integer value.
- *Left Link:* A 4-byte integer value, which stores the id of a node's left link on the tree.
- *Right Link:* A 4-byte integer value, which stores the id of a node's right link on the tree.
- *Left Thread:* A 4-byte integer value, which stores the id of a node's left thread on the tree.
- *Right Thread:* A 4-byte integer value, which stores the id of a node's right thread on the tree.
- *Weight values:* There are four integer values, 2 bytes each, which store the weight values of the node's left link (LLW), right link RLW), left thread (LTW), and right thread (RTW). To avoid floating point representations, we store the weight values in an integer form, for example, a value of 0.89 is stored as 890, a value of 0.955 as 955, and so on.

To formulate the proper information, we transform pairs of communities from an irregular topology to a single binary tree. Then we add the proper threads to the binary trees, to change them into *threaded binary trees*, which are converted in a proper form for GPU processing. These tree operations are described and analyzed in the remaining of this paragraph. *Transforming Pairs of Communities Into Binary Trees:* A pair of communities can initially be represented as a forest of two trees as follows See also Fig. 3(a) for communities C_1 and C_2 of Fig. 1.

1) From each community a node is assigned as the tree root. In this work, we choose as the root the community member with the largest number of internal links. In case of a draw, we choose the node with the largest number of external links. This is just a second criterion to handle cases of a draw, there is no specific reasoning behind this choice. Any other criterion could also be used.
2) For each node, we add the proper directed links to members inside the community. (Duplicates are allowed for example, if two nodes are linked to the same node)

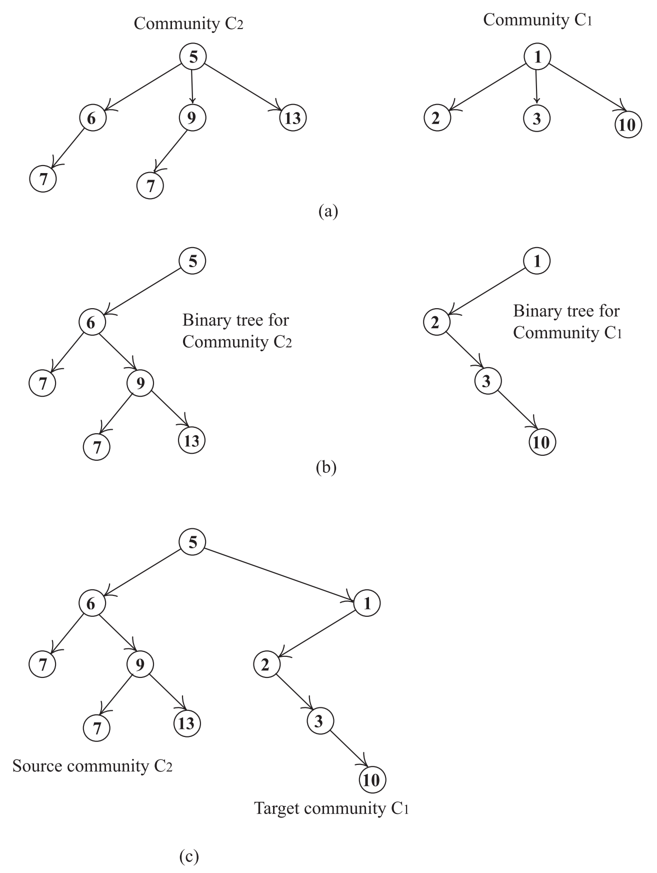


FIGURE 3. **Transforming pairs of communities into binary trees.**

provided that, the duplicate has not appeared at a higher level of the forest (so that there is no chance of being found twice over a path - node 7 is such an example).

Once the forest has been created, we can transform each tree to a binary form as follows:

1) For each node i, its left subtree is LS_i, a tree rooted at its leftmost child node, in the representation of a community as a forest.
2) For each node i, its right subtree is RS_i, a tree rooted at its right sibling, in the representation of a community as a forest. Recall that the roots have no siblings, so they have no right subtree.

Fig. 3(b) shows this simple procedure for communities C_1 and C_2 of Fig. 1.

Finally, we link together the roots of the two trees of the initial representation and add the one community as the right subtree of the other. This is seen in Fig. 3(c), where C_1 has been added as the right subtree of C_2. In this representation, the left tree will be referred to as *source community* and the right tree will be referred to as *target community*.

An important consequence of this type of representation is the avoidance of triangles and other cycles, since each node has its own siblings in its right subtree and its own children in its left subtree. The elimination of triangles and cycles is important, to reduce the time required to find paths between communities. The existence of triangles has been shown to

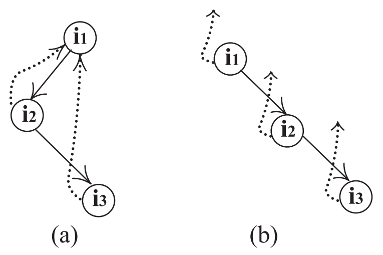
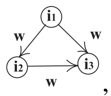


FIGURE 4. **Removing triangles.**

be important in the formation of community structures [33]. Intuitively, a strong community includes a set of highly interconnected nodes and such a strong interconnection is supported by the existence of triangles between nodes i_1, i_2, and i_3, such that $w_{i_1,i_2} \approx w_{i_1,i_3} \approx w_{i_2,i_3}$, where these weight values are relatively large. Although the idea of eliminating some links by representing the network as a forest and then a binary tree may look somehow arbitrary, it is well based on the following proposition.

Proposition 1: For a certain triangle in the form , where the tree nodes have approximately the same weight w, we can freely remove one edge, say $i_2 \rightarrow i_3$ and process only the paths $i_1 \rightarrow i_2 \rightarrow \ldots$ and $i_1 \rightarrow i_3 \rightarrow \ldots$, but not $i_1 \rightarrow i_2 \rightarrow i_3 \ldots$. This last path actually has the same similarity value compared to the other "shortest" ones. So, this triangle will be processed in either of the two following ways:

(1) As shown in Fig. 4(a), if i_2 and i_3 are siblings, with i_1 as their parent node

(2) As shown in Fig.4(b) if i_1, i_2, and i_3 are all siblings, with some other node as their parent.

The placement of threads will be explained immediately after. In Fig.4 note that, the right links are used to locate a node's next sibling and do not indicate a straight linking between two siblings on a path being formed, to avoid circles. As will be explained in the Path Formulation and Analysis section, two nodes are linked on a path via left or right threads or via left links. *Adding Threads to the Binary Trees:* Each node i has two threads, the left thread LT and the right thread RT. The left thread of each node i simply links to its parent node:

$$LT(i) = \text{ parent of } i$$

The right thread of i is the maximum-weight link of i to the target community. If i has no link to the target community, then the right thread is NULL:

$$RT(i) = \begin{cases} i', & \text{where } w_{i,i'} \text{ is maximum} \\ \text{NULL}, & \text{if no link exists} \end{cases}$$

Fig.5(a) shows the threads added to the binary tree of Fig.3(c). *The "Reflection" of a Threaded Binary Tree:* Now, we introduce the notion of "reflection" of a threaded binary tree. It is a tree similar to the one of Fig. 5(a), in which the

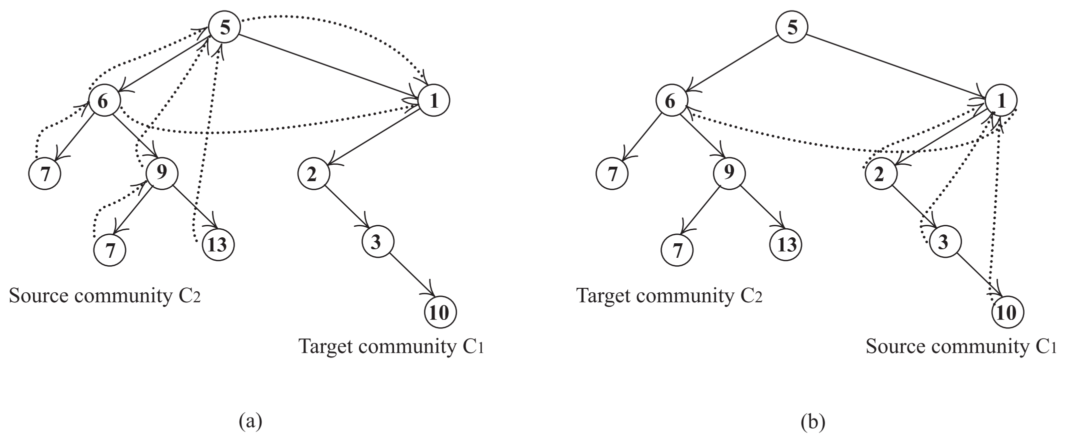
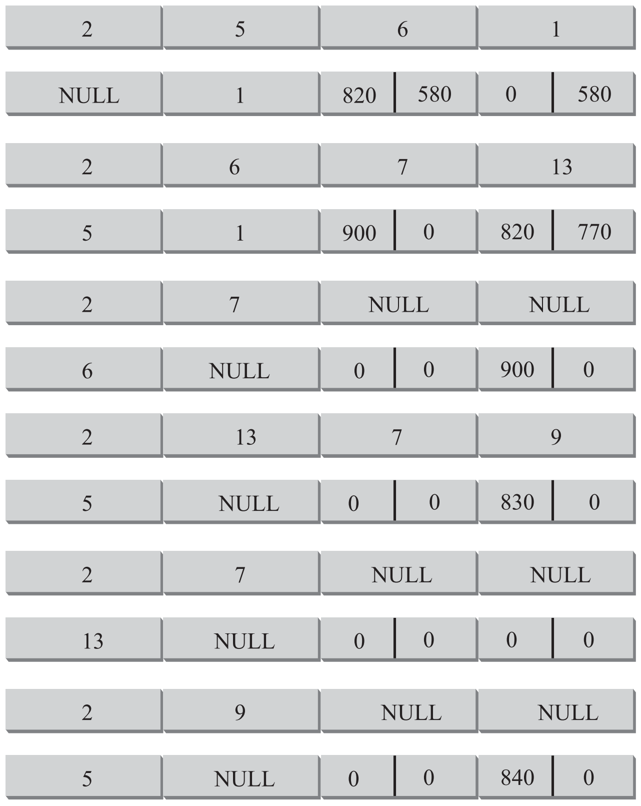


FIGURE 5. **(a) Representation of the binary tree of Fig. 3(c) as a threaded binary tree and (b) its reflection.**

source community becomes target vice versa. This tree is generated in a similar manner and its use is to find strongest paths between a pair of communities in the reverse direction (from the target to the source). The left threads point to the parent nodes and the right threads (in this case, there is only one right thread with no NULL value) point to the highest-weight link of each node to the target community (in this case it is the link from node 1 to node 6, with weight equal to 0.77). Its use will be clear when we describe the path analysis phase of our scheme. Fig. 5(b) is the reflection of Fig.5(a).

Storing the Threaded Binary Trees Into Arrays: The threaded binary tree and its reflection can be naturally represented as a set of 32-byte frames, in the form of Fig.2. Each tree node requires one such frame. The reason we do not cut down the frame size (for example, to 16 bytes per node could also be a reasonable choice) is simply to have enough space to represent networks with large number of communities and nodes. Each threaded binary tree is stored separately in an array A of size $(n/4) \times 128$ ($n/4$ rows and 128 columns), where n is the number of nodes of the tree. The $n/4$ value derives from the fact that each read/write GPU memory transaction involves a continuous area of 128 bytes. In 128 bytes, we can store 8 half frames, 16 bytes each), thus to accommodate n nodes, we need $(n/8) \times 2 = n/4$ rows. Each node is stored into two rows of the array (half 16-byte frame in one row and another half in the immediate next row). The format of the two half frames is similar to the one seen in Fig.2. An instance of such an array is shown in Fig.6, in which the 6 frames of community C_2 of Fig. 5(a) are stored. Now, if we label every node's i half frame as N_iH_1 and the other half frame as N_iH_2, each row j of A is filled as follows:

$$
\begin{aligned}
j = 0&: A[j] = |N_0H_1|N_1H_1|N_2H_1\ |\ldots|N_7H_1\ |\\
j = 1&: A[j] = |N_0H_2|N_1H_2|N_2H_2\ |\ldots|N_7H_2\ |\\
j = 2&: A[j] = |N_8H_1|N_9H_1|N_{10}H_1|\ldots|N_{15}H_1|\\
j = 3&: A[j] = |N_8H_2|N_9H_2|N_{10}H_2|\ldots|N_{15}H_2|\\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\
j = (n/4) - 2&: A[j] = |N_{[(n/4)-2]\times 4}H_1|\ldots\\
&\qquad |N_{7+[(n/4)-2]\times 4}H_1|
\end{aligned}
$$



2	5	6		1	
NULL	1	820	580	0	580
2	6	7		13	
5	1	900	0	820	770
2	7	NULL		NULL	
6	NULL	0	0	900	0
2	13	7		9	
5	NULL	0	0	830	0
2	7	NULL		NULL	
13	NULL	0	0	0	0
2	9	NULL		NULL	
5	NULL	0	0	840	0

FIGURE 6. **Storing the threaded binary trees of Fig.5.**

$$
\begin{aligned}
j = (n/4) - 1&: A[j] = |N_{[(n/4)-2]\times 4}H_2|\ldots\\
&\qquad |N_{7+[(n/4)-2]\times 4}H_2|
\end{aligned}
$$

The last two rows are used to store the elements of the last 8 nodes, from $N_{[(n/4)-2]\times 4}$ to $N_{7+[(n/4)-2]\times 4}$. The reason the CPU employs this storage policy will be made clear when we describe our storage model during our path analysis approach, in the next section. Fig. 6 shows how community C_2 (see Fig.5(a)) is stored. Note that, the first two rows of this array correspond to node 5, the next two rows correspond to node 6, etc. Each row stores exactly 16 bytes of data. The nodes shown in Fig. 6 will be stored in the first two memory rows: specifically, the

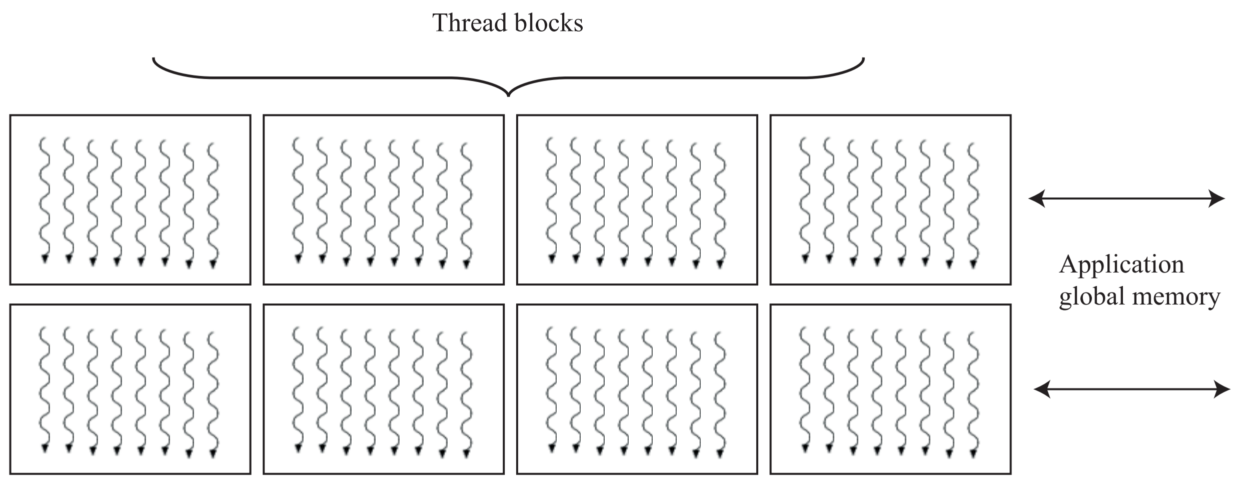


FIGURE 7. GPU memory model.

frames |2|5|6|1|, |2|6|7|13|, |2|7|NULL|NULL|, |2|13|7|9|, |2|7|NULL|NULL|, and |2|9|NULL|NULL| will be stored in the first memory row and will move together in one transaction, representing half-node data, while the remaining frames will be stored in the next memory row and be transferred together, representing the remaining half-node data.

Parallelism Issues and Computational Analysis: The main advantage of the threaded binary tree formulation phase is that, it is a very straightforward procedure that involves the transformation of a community into a forest, the transformation of a forest to a binary tree, and the placement of threads. These three procedures are clearly $O(n)$, where n is the number of community nodes and we can get a theoretical speedup of n/P, where P is the number of cores of our CPU for the overall process. Since the proposed strategy involves no sorting (unlike our previous threaded formulation scheme) and there is little communication, only between processors that work on successive levels of the forest or the tree, we can achieve almost optimal speedup for the threaded binary tree formulation, as will be shown in the experimental results section.

B. PATH FORMULATION AND ANALYSIS

Once the threaded binary trees have been stored, the data in the form of Fig.6 has to be transferred to the GPU for path formulation and analysis. Since this phase is the most expensive (overall cost analysis follows in the end of this section), the data have to be carefully passed to the GPU in the most suitable form. In this paragraph, we first need to show how to map the trees generated in Phase 1 to the GPU threads, in order to avoid memory divergence. Then, we show how the GPU processes the data to formulate the paths between communities and analyze them for overlaps.

Reading the Data Into the GPU: Before describing our GPU mapping strategy, it is necessary to describe our GPU memory model, which is shown in Fig. 7. All the executing threads are organized into blocks and the blocks are executed on a single *streaming multiprocessor*. Fig. 7 shows a SM with eight blocks. The application global memory is accessible from all the blocks. Each SM can have a maximum number of blocks and each block is divided int groups called *warps*. Each warp has 32 threads that execute the same instructions on different data, in parallel. In our scheme, each warp will be processing two threaded binary trees. Specifically: half warp will be processing a threaded binary tree (the source community) and the other half will be processing its reflection (the target threaded binary tree). Each load/store memory transaction can read or write up to 128 byte of contiguous memory. With 4-byte data words, the result is that each memory transaction can read or write up to 32 values. It is important that these values fall within the same memory segment, where each segment starts from an address, which is an integer multiple of 128 bytes. If this is not the case, part of the bandwidth is unexploited (uncoalesced memory transactions).

Generally, the available threads can access data from multiple memory spaces while executing. For example, each thread has its own private local memory, while blocks of threads have their own shared memory, which is accessible to all the threads that belong to the block. This shared memory is very fast, however its size makes it unsuitable for applications with high storage demands. A typical shared memory of 48KBytes places severe limits to the amount of nodes that can be stored. If each node is stored in 32 bytes, only $\frac{48\times1024}{32} = 1536$ nodes can be stored per block. With a block of 256 threads (which is normal, since it is desirable to have as many as possible active threads), this means that each thread can only store six nodes in the shared memory. Moreover, reducing the number of threads would be inefficient: the shared memory would be enough but because of the large number of nodes, many transfers between this memory and the global memory would be necessary and this is a burden. A possible idea to be used in order to (at least at some extent) use the shared memory is to take advantage of the temporal locality of the data generated in the path matrix. However, because of the small size of this memory, the algorithm has to be changed so that: 1) Node samples should be processed instead of the overall network, in order to reduce the data being stored, and 2) The algorithm has to be changed to work in a pipeline fashion, so that the time losses due to the necessary regular transfers between the shared and the global memory overlaps with some other useful operation. This is part of our future work. For this reason, our implementation will not use the GPU shared memory, despite its higher speed compared to the global memory. Another sort of memory is the so called Unified Memory, shared between the CPU and the GPU. In this approach, we will not make use of the unified memory. To the best of our knowledge, this memory has one disadvantage: the CPU cannot access any portion of the unified memory, while the GPU is executing, that is, synchronization is required for the CPU to be allowed to access unified memory. In our scheme, we try to avoid such synchronization issues, so we have chosen to work on the GPU global memory.

The number of threads executed in parallel depends on the amount of hardware available (for example, registers) and the thread-block size. It is desirable to have a high number of active threads, in order to take advantage of the stalling time of the memory transactions. Specifically, when a warp requires reading some data, its threads are put to stall waiting for the transaction to complete. During this time, the warp scheduler can detect another warp not stalled and put its threads into execution. This type of behavior is necessary to smooth the effects of constant memory reads which take place in our scheme.

To efficiently read data from the global memory, that is, to maximize the global memory bandwidth, it is important to minimize the number of memory transactions. These memory transactions are implemented on a half-warp basis (16 threads), which means that a memory transaction issued will satisfy half warp. The total number of transactions is minimized via *coalesced memory access*, which refers to the combination of a number of memory accesses into one transaction. In other words, the memory requests for half a warp are satisfied in one memory transaction. During each transaction, each thread can read at most 16 bytes, and the maximum bus transaction size is 128 bytes. The memory segments should be aligned, so their first address should be an integer multiple of 128. The pseudo-code of the algorithm used to perform coalesced reading from the global memory is given in Algorithm 1.

The remaining half-warp can use the same algorithm to read the reflection of the threaded binary tree, which is stored in the same fashion as the threaded binary tree. The transactions for the other half-warp are issued independently. In other words, half of a warp reads and processes the paths from a source to a target community and the other half works in the "opposite" direction (by processing the reflection tree).

Now, let us illustrate how this algorithm is applied for the storage pattern we presented for the array A, which is used by the CPU to store the binary threaded trees. The reason we have chosen the size of A to be $(n/4) \times 128$ is because each thread can read at most 16 bytes in each memory transaction. Also, all the memory transactions are implemented in batches of at most 128 bytes. This means that, in every transaction, eight threads can read half a frame, so they need two transactions to read a full frame. In other words, eight nodes are completely read within two transactions which means that, we need a total of $n/4$ transactions to fully read a threaded binary tree with N nodes. Thus, A has $n/4$ rows and 128 columns. Each transaction is on a "half-warp" basis, but in our loading scheme we use half of this "half-warp", that is, eight threads, each reading 16 bytes. Apparently, as data is stored in A in the form of 128-byte rows, each transaction reads exactly one row of A.

Algorithm 1: Coalesced Global Memory Reading per Half-Warp

```
input : An array of $n$ nodes, with size $n/4 \times 128$, each
        node is stored in 32 bytes
        index: the starting address for each thread
        mem : the first byte of an 128-byte segment
        block_id: the id of a block
        thread_id: the id of a thread within a warp
output: Each half-warp reads the proper data with the
        minimum number of transactions
1 begin
2   Locate mem, as the first memory address of the 128-byte
      segment requested by the thread with the lowest id
      number
3   Spot the consecutive, 8 in total, active threads whose
      requested data is located within this segment and locate
      the starting address for the data they request as follows:
4   $index = block\_id \times \frac{n}{4} + [thread\_id \times 16] + mem$
5   Load the 128-byte data into registers for these 8 threads
      (16 bytes per thread)
6   Repeat Steps 2 and 5 to read another 16 bytes into these
      8 threads
7   Mark the threads that completed their reading as done
8   Repeat steps 2 to 7 for the remaining 8 threads of the
      half warp
9 end;
```

Let us use an example of a set of eight threaded binary trees and their reflections, which contain 64 nodes (we use a small number of nodes to make the illustrative example more clear). These trees will be handled by eight warps found in block with $id = 0$. Also, suppose that the CPU has stored the nodes in continuous positions of an array of size $8 \times 2 \times 64 \times 32 = 32KBytes$, starting from address 0 up to address 32K-1. Initially, thread 0 (thread with the lowest id) will locate $mem = 0$, the address of the byte segment that it requests. Now, the threads $t_0 - t_7$ will compute their index value as

$$index = block\_id \times \frac{n}{4} + [thread\_id \times 16] + mem$$

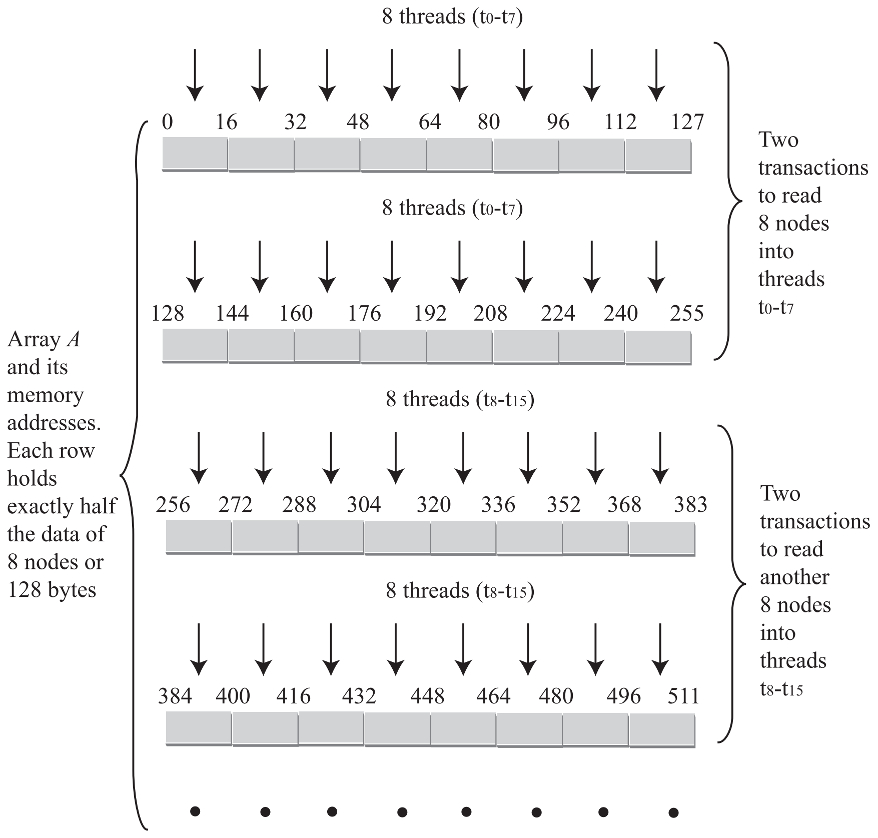

8 threads (t0-t7)
0 16 32 48 64 80 96 112 127
8 threads (t0-t7)
128 144 160 176 192 208 224 240 255
Two transactions to read 8 nodes into threads t0-t7
8 threads (t8-t15)
256 272 288 304 320 336 352 368 383
8 threads (t8-t15)
384 400 416 432 448 464 480 496 511
Two transactions to read another 8 nodes into threads t8-t15
Array A and its memory addresses. Each row holds exactly half the data of 8 nodes or 128 bytes

FIGURE 8. Applying Algorithm 1 to perform four memory transactions.

Thus, the eight threads will start from the following addresses:

$$t_0 : 0 \times 64/4 + 0 \times 16 + 0 = 0$$
$$t_1 : 0 \times 64/4 + 1 \times 16 + 0 = 16$$
$$t_2 : 0 \times 64/4 + 2 \times 16 + 0 = 32$$
$$t_3 : 0 \times 64/4 + 3 \times 16 + 0 = 48$$
$$t_4 : 0 \times 64/4 + 4 \times 16 + 0 = 64$$
$$t_5 : 0 \times 64/4 + 5 \times 16 + 0 = 80$$
$$t_6 : 0 \times 64/4 + 6 \times 16 + 0 = 96$$
$$t_7 : 0 \times 64/4 + 7 \times 16 + 0 = 112$$

This 128-byte segment is loaded to these eight threads (step 5). Then these steps are repeated to load another 16 bytes to the very same threads. The memory location *mem* is now set to 128 and the index values are similarly computed:

$$t_0 : 0 \times 64/4 + 0 \times 16 + 128 = 128$$
$$t_1 : 0 \times 64/4 + 1 \times 16 + 128 = 144$$
$$t_2 : 0 \times 64/4 + 2 \times 16 + 128 = 160$$
$$t_3 : 0 \times 64/4 + 3 \times 16 + 128 = 176$$
$$t_4 : 0 \times 64/4 + 4 \times 16 + 128 = 192$$
$$t_5 : 0 \times 64/4 + 5 \times 16 + 128 = 208$$
$$t_6 : 0 \times 64/4 + 6 \times 16 + 128 = 224$$
$$t_7 : 0 \times 64/4 + 7 \times 16 + 128 = 240$$

Then, nodes $t_8 - t_{15}$ read their nodes in a similar fashion, to complete the readings of the half warp. These 16 threads will repeat this reading mode, until all the nodes of the threaded binary tree are read. Independently, the other half warp reads the bytes of the reflection tree. Fig. 8 shows the first four memory transactions, for threads 0 to 15.

Forming the Paths: The path formulation is performed on the GPU, in order to have the proper data available at the higher possible speed. However, the use of the GPU places a number of limitations, so the design of the path formulation phase has to obey to a a number of rules:

1. To fully exploit parallelism, any type of control divergences (like if-then-else statements) has to be avoided.
2. The writing of the results to the global memory should ideally be coalesced (like the readings) so that, we achieve a minimum number of memory transactions. This means that the threads that process the tree nodes should transfer back to the path data in a contiguous form of 128 byte segments.
3. The CPU should receive the paths in a way that, it can use them to efficiently execute the expansion phase.

To facilitate the path storage, we use we use a *path matrix*, $\mathcal{P}$, which stores the nodes that need to be linked to form the paths. This linking is performed before the expansion phase by the CPU. The path-matrix is $n \times n$-bit long array, where n is the number of nodes of the threaded binary tree. If we let n be a multiple of 32 (that is, $n = n' \times 2^5$), the size of $\mathcal{P}$ will be an integer multiple of 128 bytes, since $\frac{n' \times 2^5 \times n' \times 2^5}{2^3} = (n')^2 \times 2^7 = 128(n')^2$. Therefore, the path-matrix can reside in the global memory and be made available to all 32 threads of a warp in a series of $(n')^2$ broadcast (coalesced). Also, all the threads can read/write to $\mathcal{P}$ simultaneously. The idea behind updating the path matrix is as follows:

Each node being processed initiates a path followed by its left child and follows the path followed by its parent. Also, it initiates one path starting from its parent node followed by its right link (sibling with the same parent). All the required information has been read by the thread processing this node using Algorithm 1.

Algorithm 2 presents the path matrix update process.

Algorithm 2: Updating the Path-Matrix

```
input  : Each node's LL, RL, LT, RT and the
         corresponding weights from array $A$
output: Path-matrix $\mathcal{P}$

1  begin
2  Set all elements of $\mathcal{P}$ to 0.
3   foreach thread $t$ in a warp
4      let $n_t$ be the node processed by a thread $t$
5      $\mathcal{P}[n_t, LL] = 1$ // a path continuing at node LL
6      $\mathcal{P}[LL, n_t] = 1$ // as in line 5, but opposite
     direction
7      $\mathcal{P}[LT, RL] = 1$  // new path originating from
8                                // $n_t$'s parent, followed by
     $n_t'$s
9                                // $n_t'$s sibling
10     $\mathcal{P}[RL, LT] = 1$  // as in line 7, but opposite
11                               // direction
12     $\mathcal{P}[n_t, RT] = 1$  // path to the target community
13     $\mathcal{P}[RT, n_t] = 1$  // as in line 12, but opposite
14                               // direction
15 end foreach;
16 end;
```

In the example of Fig. 5(a), let us see how the thread that processes node number 6 will update $\mathcal{P}$. From line 5, it will set $\mathcal{P}[6, 7] = 1$, indicating that node 7 follows a path immediately after node 6 (note that, the thread processing node 5 has used the same instruction to add node 6 to a path started at itself. This suggests a very important property of our scheme related to synchronization that, will be discussed after this example). Then (line 6), the same path is defined in the opposite direction, that is, $\mathcal{P}[7, 6] = 1$. Then (line 7), $\mathcal{P}[LT, RL] = \mathcal{P}[5, 13] =$ is set to 1, to define a path starting from node's 6 parent, that is node 5, followed by node's 6 sibling, that is node 13. Line 10 is doing the same job, but in opposite direction. Lines 12 and 13 set to 1 the values that define paths to the target community, so $\mathcal{P}[n_t, RT] = \mathcal{P}[1, 6] = 1$ and $\mathcal{P}[RT, n_t] = \mathcal{P}[6, 1] = 1$. Table 1 shows the overall $\mathcal{P}$ matrix for the tree of Fig. 5(a).

TABLE 1. Matrix $\mathcal{P}$ as formed by the warp handling the tree of Fig. 5(a).

	Node id								
Node id	5	6	1	7	9	2	13	3	10
5	0	1	1	0	1	0	1	0	0
6	1	0	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0	1	0
7	0	1	0	0	1	0	0	0	0
9	1	0	0	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
13	1	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0
10	0	0	1	0	0	0	0	0	0

The path-matrix is now ready to be read and analyzed by the CPU. Before proceeding to the reading and analysis phase, it is important to mention the following properties of our path formulation scheme:

1. Our scheme does not make use of dynamic structures like queues or stacks to store the path data. This would necessarily require *synchronization* between threads, e.g., in our previous example, the thread that processes node 6 should add node 6 to its queue *before* the thread processing node 7. However, when working with threads that execute in parallel, there is no guarantee of a specific order of execution, unless some type of lock or semaphore is used. In such a scenario, some parallelism will be lost.
2. The path-matrix is a multiple of 128 bytes (provided that, the number of tree nodes is a multiple of 32). Thus, it can be made available in a few memory transactions to all the warp threads, thus coalescing can be achieved. Moreover, it is a bit matrix, thus its size is relatively small.
3. Control divergence is also avoided, so we fully exploit parallelism to produce the path-matrix.
4. The implementation described focuses on the NVIDIA GPU architecture. For example, the aforementioned architecture uses 32 warps per thread, while other architectures like AMD use 64 threads in their hardware. Each vendor may decide to change that, and probably new hardware vendors may decide to come up with other sizes. However, our work can be directed to a different architecture with slight modifications. For example, if the warp size becomes 64, then, an architecture that supports memory transactions of 256 contiguous bytes would give exactly the same scheduling as the one described.

Analyzing the Paths: Once the paths have been written in the form of a path-matrix, this bit-matrix is transferred to the CPU, for *path analysis*. The main reasons that, we prefer the CPU to analyze the paths and determine new communities and overlaps are the following:

1. Reading and analyzing involves control divergences, which should preferably be avoided by the GPU.
2. The path sizes are not known in advance, so dynamic memory allocation is required, which generally reduces the GPU performance.

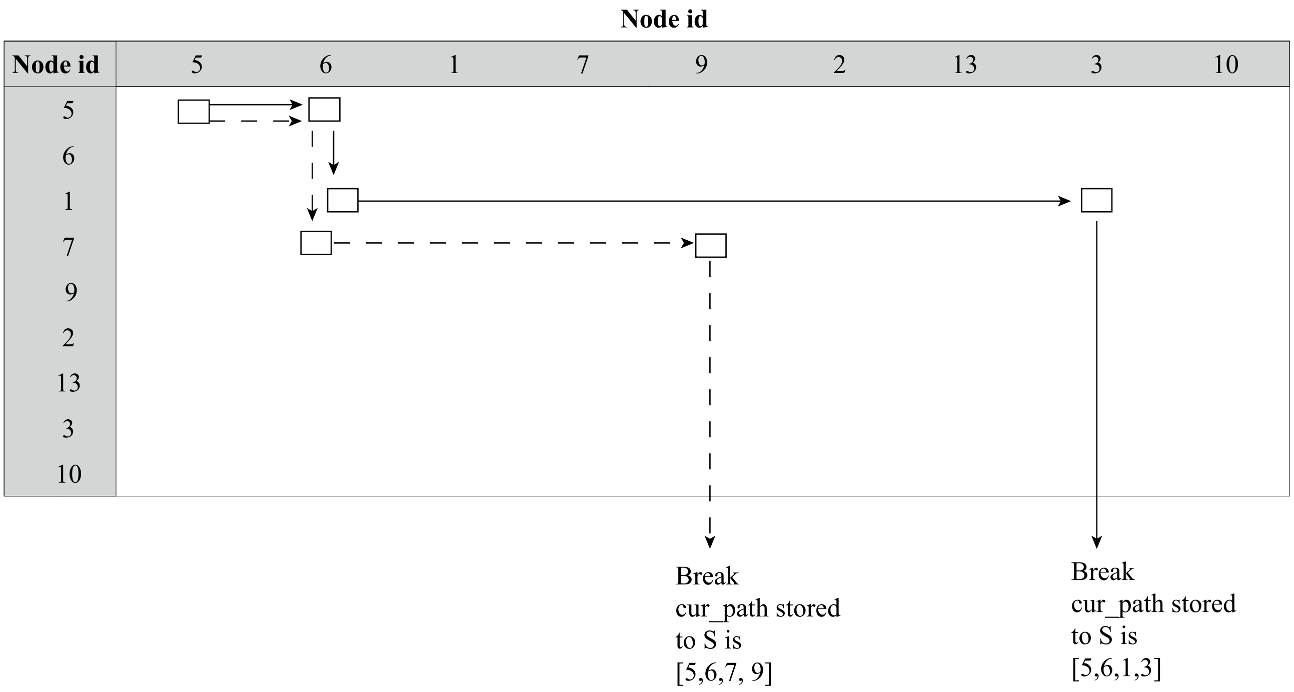


FIGURE 9. The first two zigzag moves for the example of Table 1.

3. The CPU will have to process a rather small bit-vector for each pair of communities, so the memory and processing costs are efficiently reduced. Moreover, the proposed path analysis is scheduled to be efficiently parallelized by the CPU cores.

When the CPU receives $\mathcal{P}$, it can easily examine possible overlaps between a node and a community by:

1. Reading the node's data from $\mathcal{P}$
2. Applying the double criterion of Section III, to determine the node's membership to a community.

Algorithm 3 presents the pseudo-code for our path analysis. The algorithm indicates a greedy strategy, which finds path from a selected node to other nodes of the threaded tree by employing a left-to-right and top-to-bottom zigzag move on a copy of $\mathcal{P}$, say $\mathcal{P}'$. There are two procedures, one handling the "right" move on $\mathcal{P}'$ and one handling the "down" move. Specifically, the algorithm for path analysis operates as follows:

1. Creates a copy of $\mathcal{P}$, say $\mathcal{P}$ with its rows down-rotated and its columns left-rotated so that, the node examined is located in the first row and column, to allow zigzag "right" and "down" moves.
2. Generates an active list, which includes all the node labels corresponding to the positions of the first row which are filled with 1's. These are the nodes that, the source will follow to form the paths.
3. Starting from the first row, the algorithm proceeds rightwards ("right move") to find the first non-zero element (this corresponds to the label of the first element of the active list). Then, it moves downwards ("down") until it finds the first non-zero element. The sequence of "right"-"'down" moves continues, until either a "right" move finds no other 1-element on the row being processed, or a "down" move finds no other 1-element on the column being processed, or we have reached the last row/column position, indexed $n-1$, for a matrix of n elements. All the nodes labeled in the positions this zigzag move has passed are added to the current path and when the move completes, the entire path is stored to the *path scheduling matrix*, S.
4. The last element to be found equal to 1 is eliminated (becomes 0) and the process continues, until all the paths involving the source node and the first node of the active list are exhausted. This node is removed from the active_list.
5. The next element of the active_list is selected and the original $\mathcal{P}$ matrix obtained from the GPU is copied to $\mathcal{P}'$ to restart the process, with one difference: The position corresponding to the active_list element that was removed in step 4 gets a 0 value, to avoid reprocessing paths involving this element.

An Illustrative Example: To clarify the ideas of the path analysis scheme, we use the example of Table 1 and see how the paths starting from node 5 will be processed. For node 5, there is no need for rotations. If, for example, we were to process node id 7, the rows and columns would be rotated so that they appear in the order: 7, 9, 2, 13, 3, 10, 5, 6, 1.

Fig. 9 graphically displays the formulation of two paths originating from node 5 followed by node 6. As it can be seen from Table 1, the active list includes node labels 6, 7, 9, and 13. These labels correspond to the positions of the first row which are filled with 1's. When the scan_right procedure is

Algorithm 3: Path Analysis

```
   input  : Path-matrix $\mathcal{P}'$ (copy of $\mathcal{P}$ after proper rotation)
            of size $n \times n$
   output: Path-scheduling array $S$ for a source node
 1 begin
 2 set active_list={all $j$.labels for which $\mathcal{P}[1, j] = 1$}
 3 Add first active_list element to cur_path, Set $i, j$ to 0;
 4 while active_list not empty
 5      scan_right (i,j);
 6     procedure scan_right(i,j)
 7     begin
 8         $y = j; j = j + 1;$  Add source node to cur_path;
 9         while ($\mathcal{P}[i, j] \neq 1$)
10             if ($j == n - 1$) then // indexing starts from 0
11                 if ($i == 0$) then terminate;
12                 else
13                 {
14                     Set $\mathcal{P}'[i, y]$, and $i, j, x, y$ to 0;
15                     break to Line 4
16                 }
17             end if;
18             $j = j + 1$;
19         end while;
20         add $j.label$ to cur_path; $cur\_path.length + +$;
21         if ($i < n - 1$) then scan_down (i,j);
22         else
23             {
24             scan_down (i − 1, j);  $\mathcal{P}'[i, j] = 0$
25             }
26     end procedure;
27
28     procedure scan_down(i,j)
29     begin
30         $x = i; i = i + 1;$
31         while ($\mathcal{P}[i, j] \neq 1$)
32             if ($i == n - 1$) then  // indexing starts from 0
33                 $\mathcal{P}'[x, j] = 0$; store cur_path to $S$;
34                 Compute cur_path's strength by Eq.(3)
35                 if $x == 0$ then remove $x$.label from the
36                 active_list;
37                 Set $i, j, x, y \to 0$;  break to Line 4;
38             end if;
39             $i = i + 1$;
40         end while;
41         add $i$.label to cur_path; $cur\_path.length + +$;
42         if ($j < n - 1$) then scan_right (i,j);
43         else
44             {
45             scan_right (i, j − 1); $\mathcal{P}'[i, j] = 0$
46             }
47     end procedure;
48 end while;
49 end;
```

called with parameter values $(i, j) = (0, 0)$, the `while` loop increases the column index j until it finds the first 1 in position $j = 1$, corresponding to node number $j.label = 1.label = 6$. Note that, the `if` clauses of lines 11-18 have false values, thus, no effect. Node 6 is the first node of the active list, so it is added to the *cur_path* (Line 20). The cur_path now becomes [5,6], its length becomes $1 + 1 = 2$ and the scan_down procedure (Line 22) is called with parameters $(i, j) = (0, 1)$ for the next zigzag "down" move. The scan_down procedure stores the row value by which it was called in variable x, so $x = 0$ and keeps increasing i until it finds the first 1 in column $j = 1$. The `if` clauses have no effect as we haven't reached the end of this column ($i < n - 1$), so after two iterations, the first 1 is located, for $i = 2$. The corresponding $i.label$ value is 1, so node 1 is added to the current path, thus $cur\_path = [5, 6, 1]$ and its length becomes $2 + 1 = 3$ (Line 42). Now, scan_right is called for the next zigzag "right" move, with parameters $(i, j) = (2, 1)$.

The scan_right procedure stores the column position value by which it was called to variable y, that is $y = 1$ (Line 9) and keeps increasing the column values ("right" moves) until the first 1 is found, at column $j = 7$. Since $7.label$ is node 3, this node is added to cur_path, which becomes [5, 6, 1, 3] and its length becomes $3+1 = 4$ (Line 42). Since we haven't reached the last column element, scan_down is called with parameters $(i, j) = (2, 7)$. Again, it stores the row value by which it was called to x, so $x = 2$ and starts a "down" move. However, it reaches $i = 8$ and there is no non-zero element. Now, the condition of Line 33 holds, since $i = n-1 = 8$. The algorithm now will perform as follows: (1) it turns the value of $\mathcal{P}'[2, 7]$ to 0 (Line 34), so that no other path processed can follow the route 5,6,1,3 (the path 5,6,1 could have been followed only by node $label.8 = 10$, if this value was 1), (2) stores cur_path [5,6,1,3] to S (Line 34), (3) Computes the path strength by Eq.(3), which is $\frac{0.82+0.77+1}{3} = 0.86$. The zigzag moves to form this path are shown by the arrows of Fig.9. Note that, this algorithm is greedy: a stepwise algorithm could have placed a condition to stop the zigzag moves upon reaching an element of the target community, in this example, node 3. However, the greedy approach, although slower, has the advantage that more paths are available for the update phase. Because $x \neq 0$ (Line 36), it follows that there are more paths to be processed, which involve node 6. Thus node 6 is not removed from the avtive_list. Then, the algorithm sets all the variables i, j, x, y to 0 and breaks to Line 4, to continue processing.

The scan_right process will start with parameters $(0, 0)$, and the consecutive calls scan_down(0,1) and scan_right(2,1) will be repeated as before, generating the cur_path [5,6,1]. However, when scan_right(2,1) executes, it will store the value of $y = 1$ by which it was called, but then it will find no other non-zero element on the right until position (2,8) of $\mathcal{P}'$ [recall that (2,7) has now a zero value]. Thus, because $i = 8 = n - 1$, the `If` clause of Line 11 is true. The second `If` clause of Line 12 is false (we are not in the first line, so more paths including node 6 are to be processed). The control passes to Line 15, $\mathcal{P}'[2, 1]$ becomes 0, indicating that the path [5,6,1] will no longer be processed. Again, variables i, j, x, y will turn to 0.

Then, scan_right is called again with parameters $(i, j) = (0, 0)$, then scan_down(0,1) is called and the next 1 in column $j = 1$ corresponds to node 7 ($\mathcal{P}'[3, 1]$). Processing continues as described above, and the first path to be formed is [5,6,7,9] and it is shown using the dashed arrows in Fig.9. Then, in a similar manner, the path [5,6,7] is formed and position $\mathcal{P}'[3, 1]$ is set to 0.

Finally, scan_right is called again with parameters $(i, j) = (0, 0)$, again scan_down(0,1) is called, but there is no other 1 available in column 1. Now, x has stored the value of 0 (row index by which the procedure was called), the `if` clause of Line 33 is true, $\mathcal{P}'[0, 1]$ becomes 0, cur_path [5,6] is stored to S (Line 34), but this time the `if` clause of Line 36 is also true. Thus, the node labeled at row x, column 1, that is, node 6 has to be removed from the active_list. This means that, the new copy $\mathcal{P}'$ that will be used for further processing will have the same values as $\mathcal{P}$, with the exception of $\mathcal{P}'[0, 1]$ which will be set to 0, so to avoid processing paths including the link $5 \rightarrow 6$. In the next time scan_right is called, with parameters $(i, j) = (0, 0)$ and using the new copy of $\mathcal{P}'$, it will initially call scan_down(0,2), to start forming the paths involving nodes 5,1.

Parallelism Issues and Computational Analysis: This phase involves updating the path-matrix, which is performed on the GPU and the path analysis, which is performed on the CPU. The path-matrix updates are performed in a GPU warp and by taking threads with node numbers multiple of 32, each warp processes $n/32$ threads at full speed. Moreover, the memory operations are fully coalesced and the overhead induced by their large numbers is soothed by the fact that, multiple threads are kept busy. Each SM can work on a set of community pairs to update various path matrices. The procedure is $O(n)$ per threaded tree and we obtain almost full speedups of $O(n/T)$, where T is the number of executing threads.

For the analysis performed on the CPU, we can get maximum performance, by trying to take such samples from the communities that, the binary threaded trees formed are complete (or at least, most of the nodes other than the leaves have one left and one right link). Under this hypothesis regarding the construction (which can generally be satisfied, as each node of our threaded binary trees has one sibling on the right link and one child on the left) of the threaded binary trees, the *internal path length* of the tree is $m\log(m) + O(m)$, where m is the number of internal nodes (not leaves). Thus, for a set of m_1 nodes in the source node's active_list, we require at most $m_1[m\log(m) + O(m)]$ and this value is optimal [34].

C. EXPANSION PHASE

One of the main problems in the analysis of social networks is that their structure changes in a rapid way, from moment to moment due to nodes entering/leaving the network. When a node leaves the network, processing is quite straightforward: the new node is simply removed from all the paths it is involved and the path strengths are computed accordingly. When new samples are taken to define new paths, path strengths and communities, these nodes are not taken into account. When new nodes enter the network, the nodes to which it is connected are re-examined and their strengths are re-computed. The newly entered node is considered as part of those communities, for which its paths' satisfy Eq.(4) and (5). If this is not the case, the new-coming nodes are considered to form their own communities.

Let us analyze the updating process. Initially, all the nodes are members of a single community and their path strengths determine their membership to other communities. Because the threaded binary tree structures we develop involve pairs of communities it is clear that, the information regarding its membership to other communities will be spread all over the CPU processors. A node may have a number of connections inside a community (internal links) and a number of relations outside a community (external links). The internal links are the ones necessarily stored in the same processor. The *local clustering coefficient* (see [35]), 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 increases.

To prove this fact, assume that, for a newly inserted node i, all its links, say l in total, lie in community C, thus $lcc = 1$. In this case, we only need to examine the membership of i to C and the information required (the paths) is stored locally in one processor. Therefore, a set of κl paths will be re-computed, where κ is an integer value and the overall cost is $O(\kappa l)$. Now, if one link of i, say i', is outside C and belongs to C', then the membership of i to C' has to be examined as well. In this case, if there are l' links of i' in C', then a set of $\kappa' l'$ paths will be re-computed, where κ' is an integer value and the overall cost is $O(\kappa' l')\Delta$, where Δ represents inter-processor communication overheads. Now, if without loss of generality, we assume that each neighbor of i outside C has, on the average, M links in their neighborhoods. the overall complexity would be $lcc[(\kappa l)] + (1 - lcc)(l'M)\Delta$. Consequently, the overall computational costs decrease as $lcc \rightarrow 1$ and increases as $lcc \rightarrow 0$.

V. EXPERIMENTAL RESULTS AND DISCUSSION

The community detection algorithm was implemented using an object-oriented environment, where each node is implemented as an object-member of a threaded tree. As it can be clear from the description of the previous sections, the proper data is exchanged between processing units and elements as required (for example, between the CPU and the GPU during the second phase or between the processing elements of the CPU in the third). For our simulation environment, we used an Intel Core i7-8559U Processor system, with clock speed at 2.7GHz, equipped with four cores and two threads/core, for a total of 8 logical processors. The GPU used for our simulations were implemented on a TESLA K10 GPU (an NVIDIA GPU), with 3072 cores, 192 cores per SM, 16 SMs, warp size equal to 32 threads and global memory of 8GB. With 3072 cores, we can theoretically have 3072 independent trees being processed, but because of the intense memory transactions required to transfer tree structures from the CPU to the GPU, we avoid to process simultaneously so many trees.

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 on-line 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 [36]. 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 [37] 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.

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 used. The data, as presented in [36] 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 [36], 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 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 [36], 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:

$$w_{i,j} = \frac{\text{Num. of 1s in vector } W}{\text{Num. of Attributes}}, \quad \text{where } W = I \oplus J \quad (6)$$

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

$$\begin{array}{lll} w_{0,138} = 0.9, & w_{0,131} = 0.9, & w_{0,68} = 0.6 \\ w_{0,143} = 0.8, & w_{0,86} = 0.7, & w_{138,131} = 1 \\ w_{138,68} = 0.7, & w_{138,143} = 0.7, & w_{138,86} = 0.8 \\ w_{131,68} = 0.7, & w_{131,143} = 0.7, & w_{131,86} = 0.8 \\ w_{68,143} = 0.8, & w_{68,86} = 0.9, & w_{143,86} = 0.9 \end{array}$$

and these values will be used for the link weights when forming the threaded binary trees.

For the simulations described in the next paragraphs, we took samples of circles found in [36] for given egos as our initial communities, keeping in mind though that: (1) the size of these initial communities should be a multiple of 32 to allow coalesced data reads in the GPU, and

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

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

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

TABLE 3. Some combined features of users related to ego with ID=0, as shown in [36].

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]

(2) different samples from one community could be taken in different simulations. 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, "stronger" paths within "weaker").

B. EXPERIMENTS AND PERFORMANCE EVALUATION

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 is used to measure its weak scaling. In the end, we compare the scaling of our scheme to the scaling and runtime of our scheme to other state-of-the-art methods, like PLMR [3] and the recently introduced picaso scheme [2].

1) PHASE A: THREADED BINARY TREE GENERATION

In our previous work, the first phase was divided into two procedures: The SORT procedure, which sorted every node's list of neighbors, performs sorting in a pipeline fashion and the TBTGEN procedure that takes an ordered list and adds the proper threads. In our CPU-GPU scheme, we use two procedures, the one that translates the network to a forest (from now on, it will be referred to, as NETOFOR) and the one that changes each forest into a threaded binary tree (from now on, it will be referred to, as FORTOTHR). Thus, to compare the first phase of the two algorithms, we need to compare the SORT to the NETOFOR procedure and the TBTGEN to the FORTOTHR procedure.

NETOFOR vs SORT: In the following, we compare the speedups obtained by running the NETOFOR and the SORT procedure for the 5 real network datasets, using up to 8 threads. Apparently, the speedups obtained are related to the execution times of the two procedures, but for the strong scaling behavior, we decided to present only the speedups here, for clarity reasons.

The cost of the SORT procedure depends on the number of neighbors per node to be sorted. For very small networks like Facebook with only about 4,000 nodes, it achieves an almost perfect speedup. However, as the networks become larger and the pipeline-based sorting requires more inter-processor communications, overheads increase, as sorted sub-lists are communicated and thus the speedup is reduced. The speedup achieved by the NETOFOR procedure overcomes the speedup of the SORT procedure when working on very large networks like LiveJournal. The NETOFOR procedure does not make use of any sorting procedure and requires fewer inter-processor communications, which involve processing elements working on adjacent levels of the initial forest. For smaller networks, the speedups achieved are almost identical, but for larger networks we have managed speedup increases up to $\approx 27\%$ (See Fig. 10). For example, referring to the LiveJournal network, the speedup achieved by SORT was 5.5, while NETOFOR reached a speedup of almost 7.

FORTOTHR vs TBTGEN: In the following, we compare the speedups obtained by running the FOTTOTHR and the TBTGEN procedure for the 5 real network datasets, using up to 8 threads. Again, we present only the speedups here, for clarity reasons.

The TBTGEN procedure suffers a total of m^2 value comparisons per neighborhood in order to determine the left thread values, when generating the threaded binary tree.

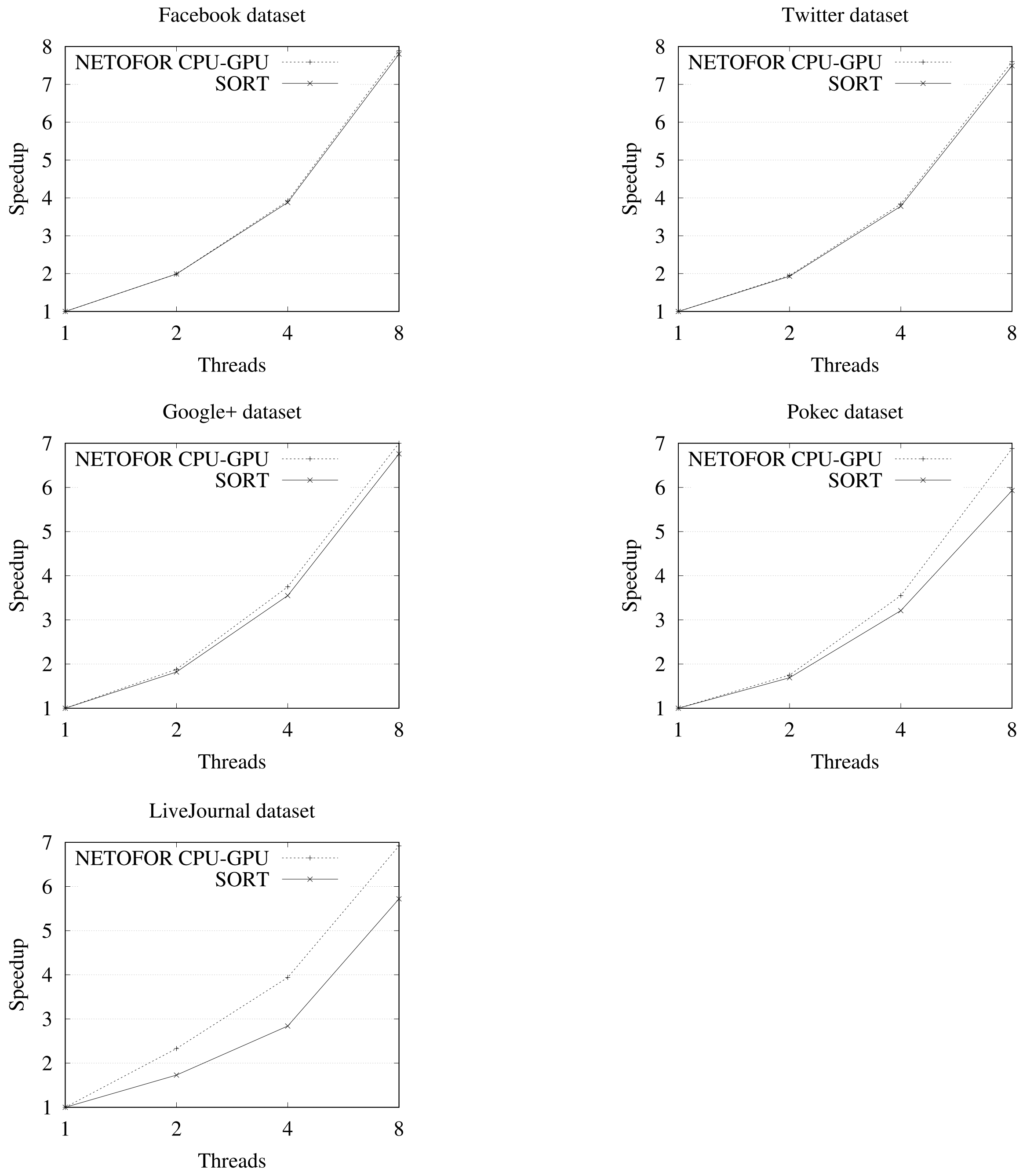


FIGURE 10. **Phase A -NETOFOR vs SORT- strong scaling behavior.**

To the contrary, the FORTOTHR procedure compares m values to find the maximum-weight link to add as a right thread value. The experimental results of Fig. 11 (strong scaling) show that the FORTOTHR speedup is about 20% larger (5 against 4.1) than the TBTGEN speedup, referring to largest network, LiveJournal. No better improvements can be achieved due to the extra storing procedure, which is necessary to transform the data into a format suitable for GPU processing.

2) PHASE B: PATH FORMULATION AND ANALYSIS

In the experiments for Phase B, we have to consider the effects of the GPU involvement. Instead of comparing a CPU only version of our new scheme to the CPU-GPU approach, we preferred to compare our new scheme to our previous strategy, which is purely CPU-implemented. There are two reasons behind this choice: (1) The path formulation procedures in both strategies are, to a large extent, comparable, based on reading data from threaded binary trees. In this regard, we can highlight the advantage of the GPU involvement, and (2) Running our new scheme on the CPU only, would require eliminating all the modifications suggested in the Section "Reading Data into the GPU", so this would actually be a different algorithm.

The path analysis is the most computationally intensive and memory- demanding procedure of our scheme and the

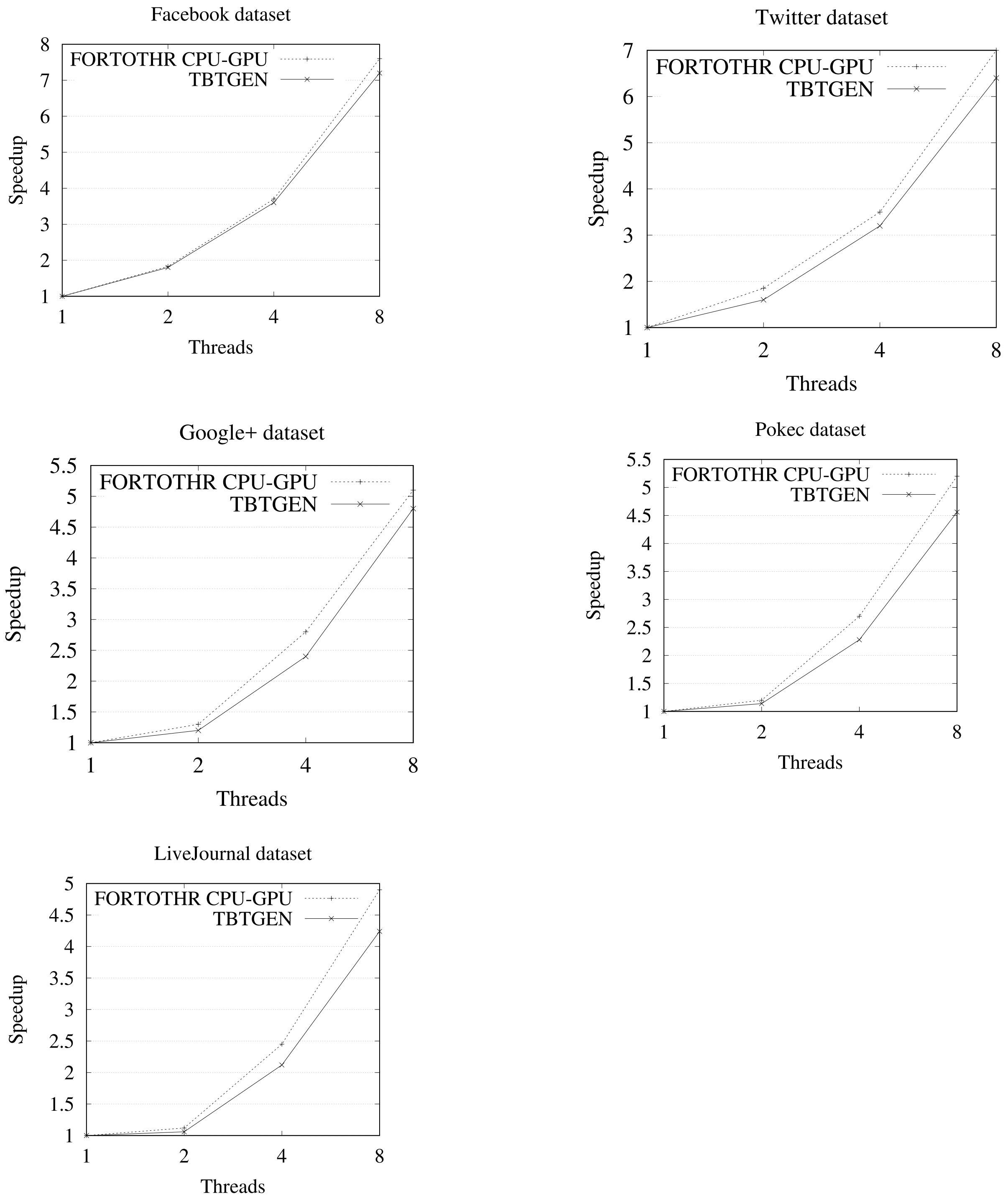


FIGURE 11. Phase A -FORTOTHR vs TBTGEN- strong scaling behavior.

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 (like Google+), the time required to traverse the trees enlarges.

For LiveJournal, our CPU-GPU strategy achieves a speedup of 5 (for 8 threads) and the speedup increase is $\approx$ 20% compared to the speedup of about 4.1 of our previous strategy. For small networks with small number of edges and nodes, like Facebook, the CPU-GPU strategy manages speedups of about 7. Figure 12 presents the strong scaling

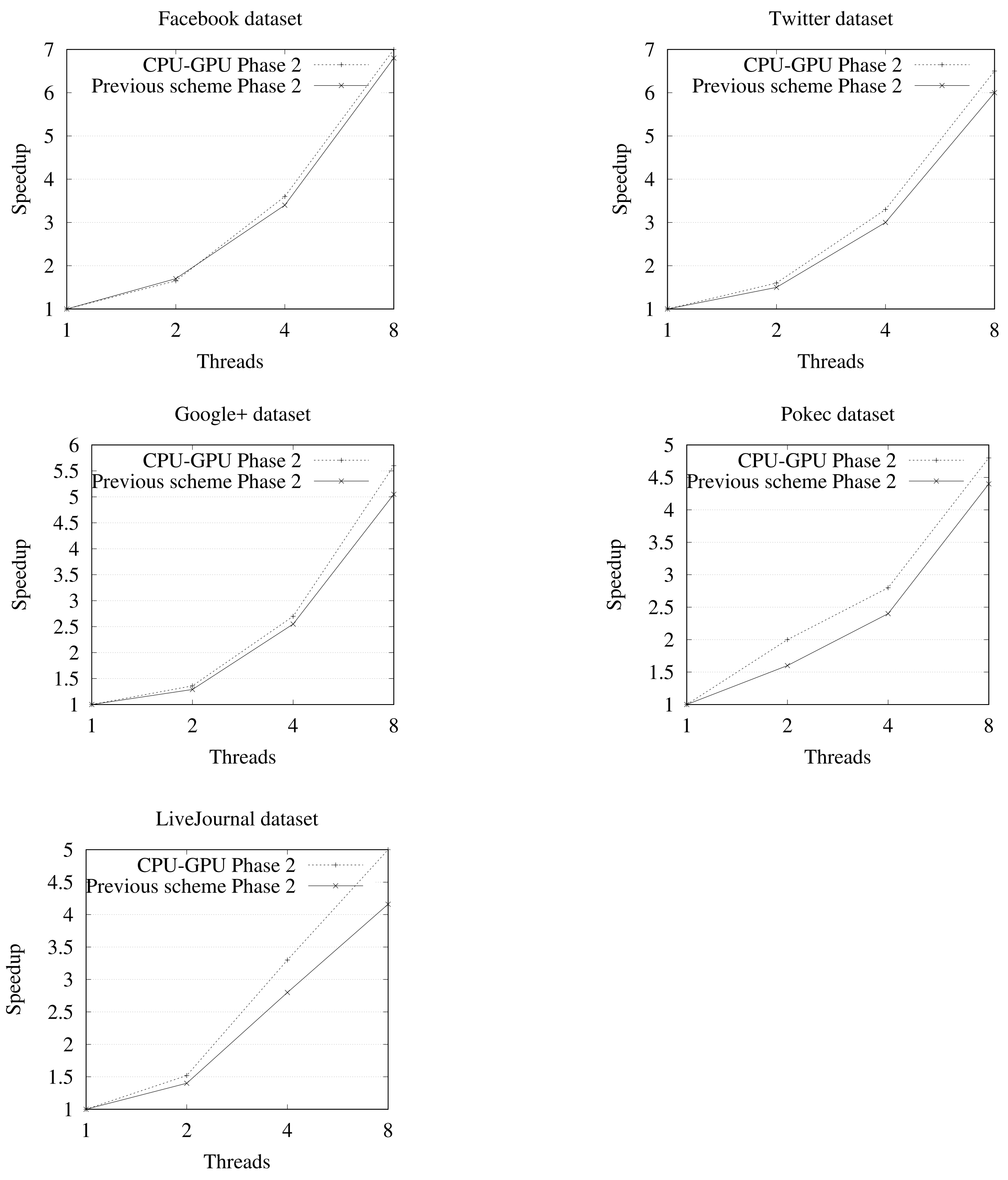


FIGURE 12. Phase B, strong scaling behavior comparisons.

behavior of the path analysis phase, for the five networks examined.

Figure 13 compares the accumulative speedups of the new CPU-GPU strategy and our previous scheme, for two networks, namely Pokec and LiveJournal. For Pocek, the speedup increases by $\approx 12\%$, while for the largest network examined, LiveJournal, this increase is about 15%. These results are the average values obtained from different sets of experiments.

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 and compare it to the weak scaling of our previous 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 algorithm on a single core and then keep doubling the number

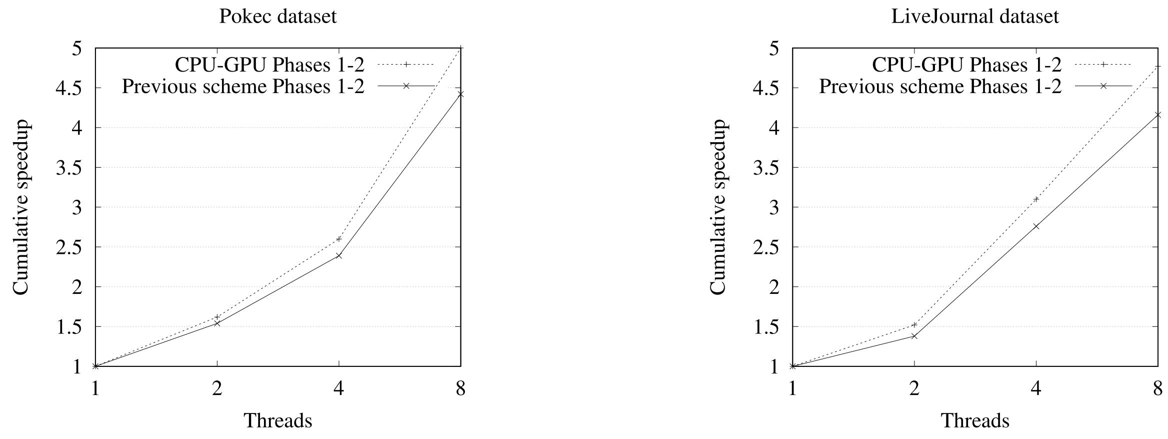


FIGURE 13. Phases 1 and 2, cumulative speedups for Pokec and LiveJournal networks.

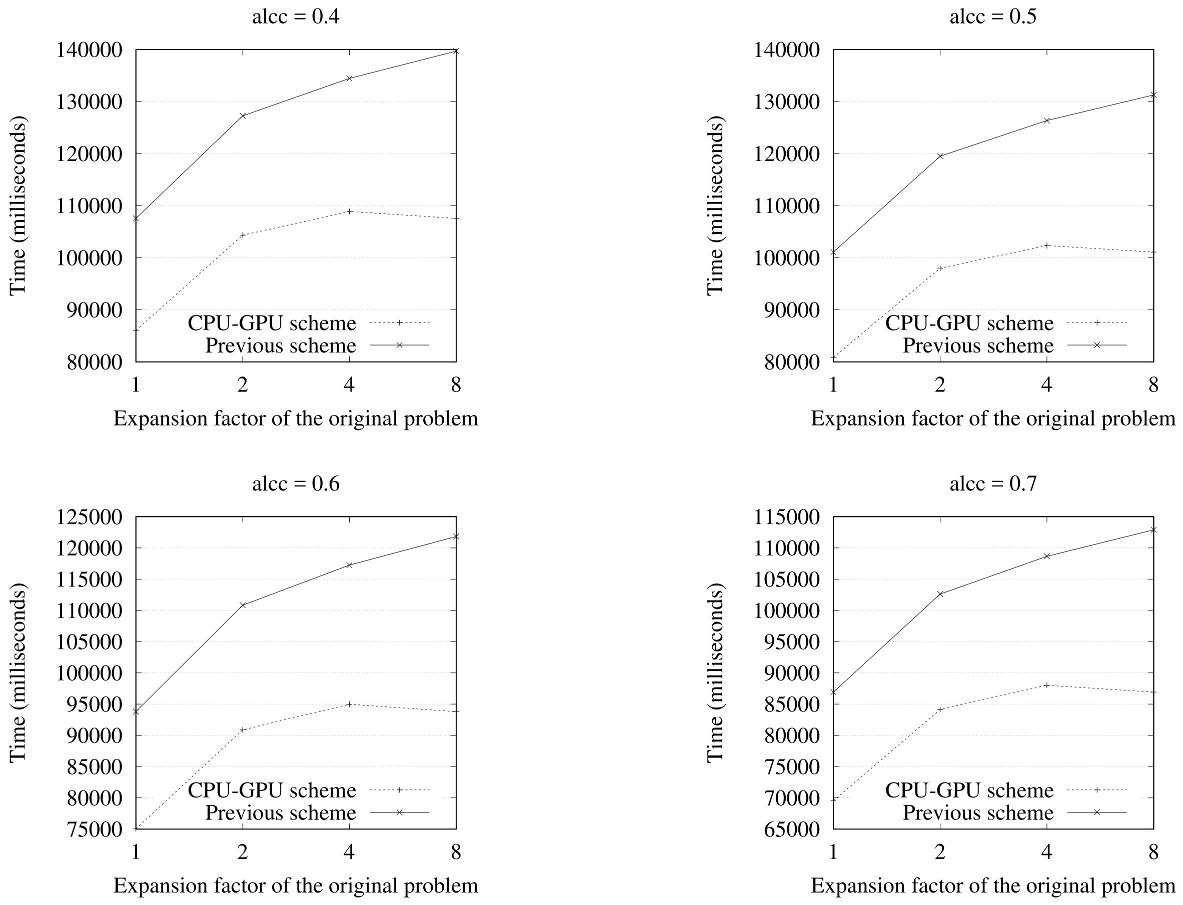


FIGURE 14. Comparison of the weak scaling behavior on the Google+ network, for four different *alcc* values.

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

To perform updating, we added samples of double the size of the original network. Our scheme tries to place each new node in a community or in 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 Equations (4) and (5). For our experiments, the *alcc* value ranged between 0.4 (its value for the Google+ dataset is ≈ 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 14 shows the weak scaling for the Google+ dataset. The expansion factors are

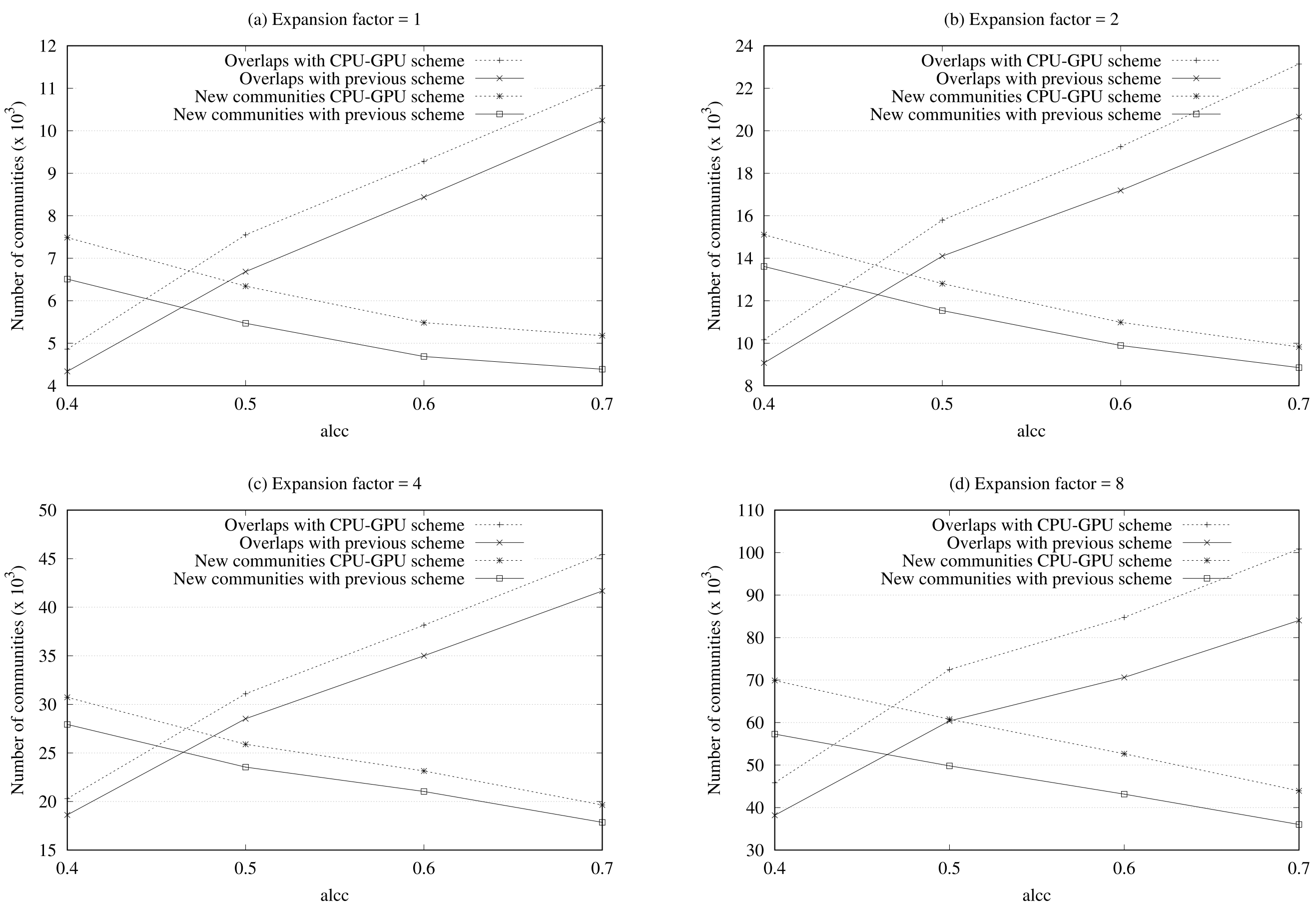


FIGURE 15. **Comparisons of the CPU-GPU strategy and our previous scheme on the number of new communities and overlaps detected for four different *alcc* values.**

shown in the horizontal axis. The original problem (expansion factor=1) had about 100K (107,614) nodes. The larger problem (expansion factor=8) had about 1M nodes and an average of 85 neighbors per node. Note that, the both schemes perform better as the *lcc* values increase, as expected (see the proof in the Expansion Phase paragraph). Moreover, from all the experiments, we find that, on the average, the CPU-GPU scheme outperforms our previous scheme by $\approx 28\%$.

One way to analyze the GPU's contribution to the entire application would be to quantify the cost of CPU-GPU transferring cost as part of the overall cost. However, such a quantification is not easy as there is a phase which involves CPU-GPU communications phases relying only on the CPU. The experimental results offer us an intuitive conclusion regarding the GPU's contribution: In our new CPU-GPU scheme, the total execution time does not decrease analogously to the speedup increase. The accumulative speedup values (referring merely to the CPUs, see Figure 13) would suggest an average of 15% improvement on the execution time. However, the results have shown a total execution time reduction of about 27%. This reduction is mainly due to the introduction of the GPU processing in Phase 2: the use of the GPU plays an important role in this case, as the data required for path analysis is delivered to the CPU at high speeds, even when the problem size increases. Although the memory transactions also increase, our strategy keeps using large thread numbers of the GPU as these transactions incur, thus the memory effect is soothed.

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 1M nodes (In Fig. 15(a) to (d), we keep doubling the expansion factor for every set). The two schemes have similar behavior, but the CPU-GPU scheme detects $\approx 12\%$ more communities. This increase is explained by the fact that our new scheme, due to the presence of the GPU, does not exclude any path from processing while our previous scheme considers ever-increasing paths only. However, both strategies use a strict double criterion to determine community membership, so, although the number of paths processed is larger, the increase on the number of communities and overlaps is only 12%.

For both schemes, when we have low *alcc* values (average up to 50%), more newly formed communities than overlaps are detected. 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* increases, more overlaps than new communities are detected, as explained in Section IV.C. For example, when the expansion factor was 8 (about 1M nodes) and *alcc* = 0.7, about 140K communities were detected, where 32% (about 45K) were overlaps while 68% (about 95K) were newly formed communities. Figure 15 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 CPU-GPU scheme to our previous scheme and to two other well-known strategies: The PLMR method [3], and the picaso method [2]. We conducted two sets of experiments. In the first set, we compare the aggregated speedup for each phase of the CPU-GPU strategy to the aggregated speedup for each of the phases of our previous work and of the PLMR method. In the second set, we perform speedup comparisons between the CPU-GPU scheme, the PLMR, the picaso method and our previous scheme, 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 PLMR 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 contracting each community to a certain super-node, 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, the CPU-GPU scheme behaves in a similar manner, as was the case with our previous work: the first and third phases scale better compared to the phase that detects the communities and their overlaps.

Fig. 16 compares the first phase of the three strategies. The two procedures of the CPU-GPU strategy, that is, the NETOFOR and the FORTOTHR have good scaling performance, due to the lack of large numbers of inter-processor communications and the lack of large numbers of comparisons. The PLMR first phase is based on the total number of nodes so, the CPU-GPU scheme (based on each node's neighbors) naturally scales better. The SORT and TBT_GEN operators require more communications between processing elements and more comparisons, so the procedures our CPU-GPU strategy overcomes them in terms of scaling. The second phase the CPU-GPU scheme does not scale as well as the other two phases, but the experiments conducted the

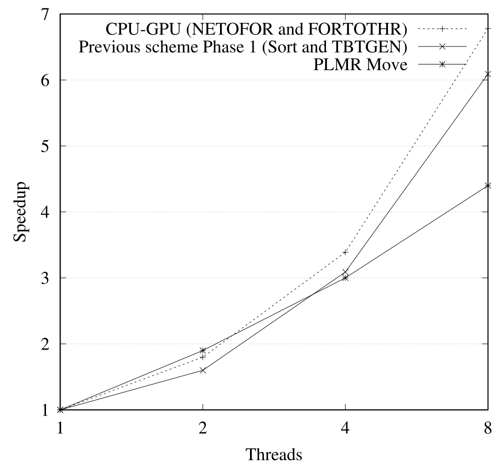


FIGURE 16. Speedup comparisons between the first phase of our scheme and the first phase of the PLMR scheme.

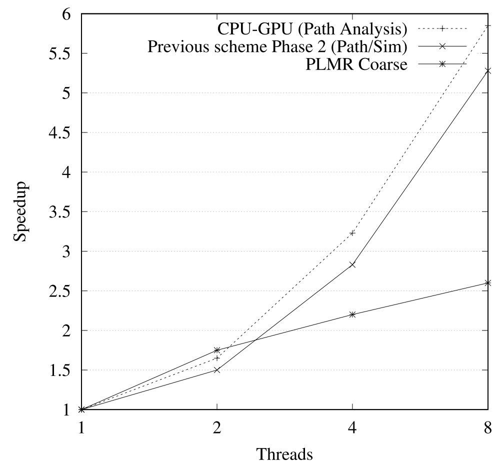


FIGURE 17. Speedup comparisons between the second phase of our scheme and the second phase of the PLMR scheme.

PLMR method indicate that this scheme dos not have very large gains due to parallelism. This explains the large speedup difference (see Fig. 17) between the corresponding phases of the two strategies. The third phase of our CPU-GPU strategy scales better for networks with larger *alcc*, like Facebook, Twitter or Google plus. Its performance degrades for networks like LiveJournal or Pokec. Since our CPU-GPU strategy requires only simple computations on the paths already defined in the previous phase (our previous strategy may need to make some extra comparisons during this phase), this phase outperforms the third phase of our previous strategy and the third phase of the PLMR strategy, which is based on the changes occurring the coarsening phase (see Fig. 18). Apparently, there are cases when the changes in the coarsening phase are such that the gains from parallelism are reduced.

Then, we compare the speedup of our work, to the speedups of our previous work, picaso [2] and PLMR on the LiveJournal network (see Fig. 19). The results indicate that, while our scheme outperforms all the other strategies while the picaso strategy comes second, as far as scaling is concerned.

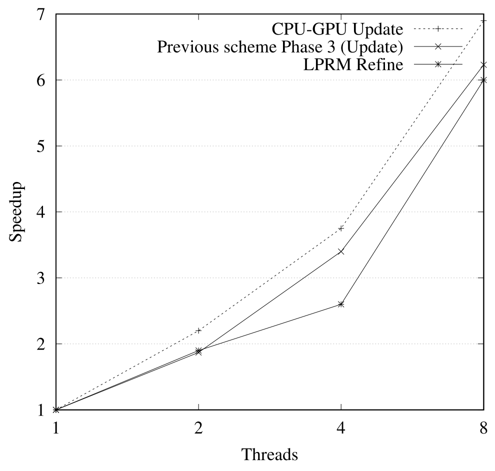


FIGURE 18. Speedup comparisons between the third phase of our scheme and the third phase of the PLMR scheme.

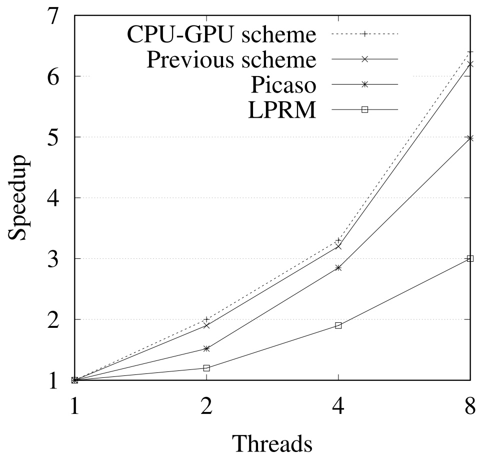


FIGURE 19. Speedup comparisons between our strategy, the PLMR and the Picaso method.

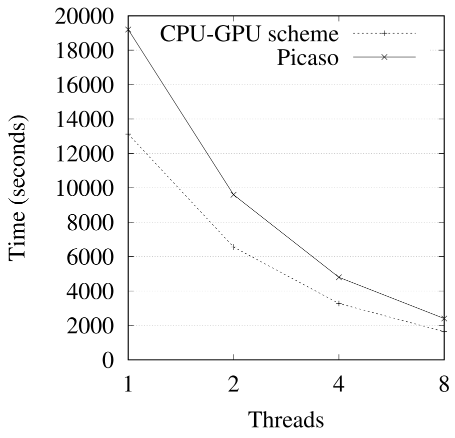


FIGURE 20. Runtime comparison of the CPU-GPU and the Picaso schemes.

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. However, the main properties of our scheme, like the GPU usage and the reduced number of inter-processor communications and comparisons explains the speedup results. It should be noted also, that while our experiments were conducted on logical processor, thus the results suffer from hyper-threading, the picaso method uses 16 physical processors and does not suffer that much from this effect. In the last set of experiments, we compare the running time of our scheme to the picaso method. In this set, we run a series of experiments and took average values. Our initial neighborhood samples fed to the GPU varied from 8K to 256 K nodes for which new communities and overlaps were detected. This community detection was followed by expansion phases up to a value of 2M. The average running times obtained are used in this set of experiments. On the average, our scheme offers an improvement of about 30% in the detection of communities, when the number of nodes approached 2M. Fig. 20 shows the comparison results for a number of threads, from 1 to 8.

VI. CONCLUSIONS-FUTURE WORK

This paper presented an extension of our previous threaded binary tree approach for community detection, which is schedule to exploit the CPU and the GPU processing capabilities. Due to the certain limitation regarding divergences, we used the GPU on the second phase of our work, where the GPU generates a bit-vector used later by the CPU to analyze the paths. The CPU-GPU strategy requires fewer inter-processor communications and comparisons than our previous scheme, thus the speedup is increased. 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.

Our experimental results have shown that, for the procedures of the first phase, we have managed average speedup increases of about 15-20% compared to our previous work. These increases were mainly of the reduced number of inter-processor communications required by the CPU-GPU strategy. For the second phase, the use of the GPU guarantees a reduced total processing time for the second phase compared to our previous work, which has reached an average of 15% for big networks, like LiveJournal, (for 8 threads). For the third phase, the CPU-GPU scheme outperforms our previous scheme by about 28%, on the average, as the problem kept increasing (weak scaling). The use of different *lcc* values verified the fact that more overlaps than new communities were detected for larger *lcc* values, Also, the overall performance was increased.

Finally, we compared our work to other known schemes like the Parallel Louvain Method with Refinement (PLMR) and the picaso strategy. The ''per-phase'' comparisons have indicated that our strategy outperformed the PLMR in terms of speedup values. Also, it outperformed the picaso strategy. Finally, we compared the runtime of the CPU-GPU scheme to the runtime of the picaso scheme, using 8 processing elements. The results have indicated an average of 30% improvement, when the number of nodes approached 2M.

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* ([3]). 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. Finally, extensive work needs to be done, in order to use the GPU in a more efficient way. For example, efforts are made to exploit the unused shared memory by using it as L1 cache. Our initial approach is to use *data prefetching*, where data blocks are prefetched from the global memory to the shared memory. A advantage in this approach is that the accuracy can reach almost 100%, since the memory reads in our work have been scheduled in a regular way. On the other hand, if each thread generates a prefetching request, then there are large numbers of additional memory requests that require additional time to be serviced. In this regard, we need to propose an efficient software-hardware mechanism tailored to our scheme and the hardware being used. Another idea is to take advantage of the temporal locality of the path matrix and work with smaller node samples (in this scenario, statistics is necessary to group node sets into a single one based on the observed behavior, so as to decrease the number of processed nodes) instead of processing large samples or the overall network. In this regard, the shared memory could be used in an effective manner.

Moreover, we seek for a way of assigning to the GPU more procedures related to our work. Because of divergences, this in not an easy task, however, the GPU technology keeps adding new features, that probably can be used for such purposes.

Finally, a careful pipeline-based implementation of the three phases of our scheme needs to be elaborated. Such a scheme could include 3 stages operating in parallel: in the first stage, a dedicated portion of the CPUs generate the threaded trees, in the second stage, the GPU generates the paths, and in the third, a portion of the CPUs performs the path analysis. In such a scenario, the overall processing time would be overlapped, but careful scheduling is required (perhaps with delay introduction that would slow down the system), as the time required by the three phases is completely different.

REFERENCES

[1] S. Souravlas, A. Sifaleras, and S. Katsavounis, "A parallel algorithm for community detection in social networks, based on path analysis and threaded binary trees," *IEEE Access*, vol. 7, pp. 20499–20519, 2019.

[2] S. Qiao, N. Han, Y. Gao, R.-H. Li, J. Huang, J. Guo, L. A. Gutierrez, and X. Wu, "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.

[3] 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.

[4] N. Antoniadis and A. Sifaleras, "A hybrid CPU-GPU parallelization scheme of variable neighborhood search for inventory optimization problems," *Electron. Notes Discrete Math.*, vol. 58, pp. 47–54, Apr. 2017.

[5] F. Zhang, P. Di, H. Zhou, X. Liao, and J. Xue, "RegTT: Accelerating tree traversals on GPUs by exploiting regularities," in *Proc. 45th Int. Conf. Parallel Process. (ICPP)*, Aug. 2016, pp. 562–571.

[6] J. Liu, N. Hegde, and M. Kulkarni, "Hybrid cpu-gpu scheduling and execution of tree traversals," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, Art. no. 40.

[7] 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.

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

[9] 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, Sep. 2007, Art. no. 036106.

[10] M. E. J. Newman, "Detecting community structure in networks," *Eur. Phys. J. B*, vol. 38, no. 2, pp. 321–330, 2014.

[11] 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," *Proc. VLDB Endowment*, vol. 8, no. 10, pp. 998–1009, Jun. 2015.

[12] S. I. Souravlas and A. Sifaleras, "On the detection of overlapped network communities via weight redistributions," in *GeNeDis* (Advances in Experimental Medicine and Biology), P. Vlamos, Ed. Dordrecht, The Netherlands: Springer, 2017, pp. 205–214.

[13] S. Souravlas, A. Sifaleras, and S. Katsavounis, "A novel, interdisciplinary, approach for community detection based on remote file requests," *IEEE Access*, vol. 6, pp. 68415–68428, 2018.

[14] A. Prat-Pérez, D. Dominguez-Sal, J. M. Brunat, and J.-L. Larriba-Pey, "Shaping communities out of triangles," in *Proc. 21st ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2012, pp. 1677–1681.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] M. E. J. Newman, "Modularity and community structure in networks," *Proc. Nat. Acad. Sci. USA*, vol. 103, no. 23, pp. 8577–8582, Jun. 2006.

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

[23] J. M. Kumpula, M. Kivelä, K. Kaski, and J. Saramäki, "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.

[24] 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. 4, pp. 218–230, Dec. 2017.

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

[26] 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.

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

[28] Y.-Y. Ahn, J. P. Bagrow, and S. Lehmann, "Link communities reveal multiscale complexity in networks," *Nature*, vol. 466, no. 7307, pp. 761–764, Aug. 2010.

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

[30] A. Clauset, M. E. J. 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.

[31] 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.

[32] 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.

[33] C. Klymko, T. G. Kolda, and D. Gleich, "Using triangles to improve community detection in directed networks," 2014, *arXiv:1404.5874*. [Online]. Available: https://arxiv.org/abs/1404.5874

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

[35] 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 Autom.*, Jun. 2014, pp. 3240–3245.

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

[37] J. Leskovec and J. J. Mcauley, "Learning to discover social circles in ego networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Red Hook, NY, USA: Curran Associates, 2012, pp. 539–547.

STAVROS SOURAVLAS (Member, IEEE) is currently an Assistant Professor of computer architecture and digital logic design with the Department of Applied Informatics, School of Information Sciences, University of Macedonia, where he joined in 2014. His research interests include computer architecture and performance evaluation, parallel and distributed systems, big data stream scheduling, cloud computing, systems modeling, and simulation.

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