Hindawi Publishing Corporation
Scientific Programming vol. 2015 (2015)
Article ID 461362

Research Article

Parallelizing SLPA for Scalable Overlapping Community Detection

Konstantin Kuzmin,[1] Mingming Chen,[1] and Boleslaw K. Szymanski[1,2]

[1]*Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180, USA*
[2]*The Faculty of Computer Science and Management, Wrocław University of Technology, 50-370 Wrocław, Poland*

Correspondence should be addressed to Konstantin Kuzmin; kuzmik@rpi.edu

Received 3 March 2014; Accepted 17 November 2014

Academic Editor: Przemyslaw Kazienko

Communities in networks are groups of nodes whose connections to the nodes in a community are stronger than with the nodes in the rest of the network. Quite often nodes participate in multiple communities; that is, communities can overlap. In this paper, we first analyze what other researchers have done to utilize high performance computing to perform efficient community detection in social, biological, and other networks. We note that detection of overlapping communities is more computationally intensive than disjoint community detection, and the former presents new challenges that algorithm designers have to face. Moreover, the efficiency of many existing algorithms grows superlinearly with the network size making them unsuitable to process large datasets. We use the Speaker-Listener Label Propagation Algorithm (SLPA) as the basis for our parallel overlapping community detection implementation. SLPA provides near linear time overlapping community detection and is well suited for parallelization. We explore the benefits of a multithreaded programming paradigm and show that it yields a significant performance gain over sequential execution while preserving the high quality of community detection. The algorithm was tested on four real-world datasets with up to 5.5 million nodes and 170 million edges. In order to assess the quality of community detection, at least 4 different metrics were used for each of the datasets.

1. Introduction

Analysis of social, biological, and other networks is a field which attracts significant attention as more and more algorithms and real-world datasets become available. In social science, a community is loosely defined as a group of individuals who share certain common characteristics [1]. Based on similarity of certain properties, social agents can be assigned to different social groups or communities. Knowledge of communities allows researchers to analyze social behaviors and relations between people from different perspectives. As social agents can exhibit traits specific to different groups and play an important role in multiple groups, communities can overlap. Usually, there is no a priori knowledge of the number of communities and their sizes. Quite often, there is no ground truth either. Knowing the community structure of a network empowers many important applications. Communities can be used to model, predict, and control information dissemination. Marketing companies, advertisers, sociologists, and political activists are able to target specific interest groups. The ability to identify key members of a community provides a potential opportunity to influence the opinion of the majority of individuals in the community. Ultimately, the community opinion can be changed when only a small fraction of the most influential nodes accepts a new opinion [2].

Biological networks such as neural, metabolic, protein, genetic, or pollination networks and food webs model interactions between components of a system that represent some biological processes [3]. Nodes in such networks often correspond to genes, proteins, individuals, or species. Common examples of such interactions are infectious contacts, regulatory interaction, and gene flow.

The majority of community detection algorithms operate on networks which might have strong data dependencies between the nodes. While there are clearly challenges in

designing an efficient parallel algorithm, the major factor which limits the performance is scalability. Most frequently, a researcher needs to have community detection performed for a dataset of interest as fast as possible subject to the limitations of available hardware platforms. In other words, for any given instance of a community detection problem, the total size of the problem is fixed while the number of processors varies to minimize the solution time. This setting is an example of a strong scaling computing. Since the problem size per processor varies with the number of processors, the amount of work per processor goes down as the number of processors is increased. At the same time, the communication and synchronization overhead does not necessarily decrease and can actually increase with the number of processors, thus limiting the scalability of the entire solution.

There is yet another facet of scaling community detection solutions. As more and more hardware computing power becomes available, it seems quite natural to try to uncover the community structure of increasingly larger datasets. Since more compute power currently tends to come rather in a form of increased processor count than in a single high performance processor (or a small number of such processors), it is crucial to provide enough data for each single processor to perform efficiently. In other words, the amount of work per processor should be large enough so that communication and synchronization overhead is small relative to the amount of computation. Moreover, a well-designed parallel solution should demonstrate performance which at least does not degrade and perhaps even improves when run on larger and larger datasets.

Accessing data that is shared between several processes in a parallel community detection algorithm can easily become a bottleneck. Several techniques have been studied, including shared-nothing, master-slave, and data replication approaches, each having its merits and drawbacks. Shared-memory architectures make it possible to build solutions that require no data replication at all since any data can be accessed by any processor. One of the key design features of our multithreaded approach is to minimize the amount of synchronization and achieve a high degree of concurrency of code running on different processors and cores. Provided that the data is properly partitioned, the parallel algorithm that we propose does not suffer performance penalties when presented with increasing amounts of data. Quite the contrary, results show that, with larger datasets, the values of speedup avoid saturation and continue to improve up to maximal processor counts.

Validating the results of community detection algorithms presents yet another challenging task. After running a community detection algorithm how do we know if the resulting community structure makes any sense? If a network is known to have some ground truth communities then the problem is conceptually clear—we need to compare the output of the algorithm with the ground truth. It might sound like an easy problem to solve but in reality there are many possible ways to compare different community structures of the same network. Unfortunately, there is no one single method that can be used in any situation. Rather a combination of metrics can tell us how far our solution is from that represented by the ground truth. As mentioned earlier, for many real-life datasets it is not feasible to come up with any kind of ground truth communities at all. Without them, comparative study of values obtained from different metrics for community structures output by different algorithms seems to be the only way of judging the quality of community detection.

The rest of the paper is organized as follows. An overview of relevant research on parallel community detection is presented in Section 2. Section 3 provides an overview of the sequential SLPA algorithm upon which we base our parallel implementation. It also discusses details of the multithreaded community detection on a shared-memory multiprocessor machine along with busy-waiting techniques and implicit synchronization used to ensure correct execution. We describe the way we partition the data and rearrange nodes within a partition to maximize performance. We also discuss the speedup and efficiency accomplished by our approach. Detailed analysis of the quality of community structures detected by our algorithm for four real-life datasets relative to ground truth communities (when available) and based on the sequential SLPA implementation is given in Section 4. Finally, in Section 5, closing remarks and conclusions are provided. We also discuss some of the limitations of the presented solution and briefly describe future work.

2. Related Work

During the last decade substantial effort has been put into studying network clustering and analysis of social and other networks. Different approaches have been considered and a number of algorithms for community detection have been proposed. As online social communities and the networks associated with them continue to grow, the parallel approaches to community detection are regarded as a way to increase efficiency of community detection and therefore receive a lot of attention.

The clique percolation technique [4] considers cliques in a graph and performs community detection by finding adjacent cliques. The k-means clustering algorithm partitions m n-dimensional real vectors into k n-dimensional clusters where every point is assigned to a cluster in such a way that the objective function is minimized [5]. The objective function is the within-cluster sum of squares of distances between each point and the cluster center. There are several ways to calculate initial cluster centers. A quick and simple way to initialize cluster centers is to take the first k points as the initial centers. Subsequently at every pass of the algorithm the cluster centers are updated to be the means of points assigned to them. The algorithm does not aim to minimize the objective function for all possible partitions but produces a local optimal solution instead, that is, a solution in which for any cluster the within-cluster sum of squares of distances between each point and the cluster center cannot be improved by moving a single point from one cluster to another. Another approach described in [6] utilizes an iterative scan technique in which density function value is gradually improved by adding or removing edges. The algorithm implements a shared-nothing architectural

approach. The approach distributes data on all the computers in a setup and uses master-slave architecture for clustering. In such an approach, the master may easily become a bottleneck when the application requires a large number of processors because of the network size. A parallel clustering algorithm is suggested in [7], which is a parallelized version of DBSCAN [8].

A community detection approach based on propinquity dynamics is described in [9]. It does not use any explicit objective function but rather performs community detection based on heuristics. It relies on calculating the values of topology-based propinquity which is defined as a measure of probability that two nodes belong to the same community. The algorithm works by consecutively increasing the network contrast in each iteration by adding and removing edges in such a way as to make the community structure more apparent. Specifically, an edge is added to the network if it is not already present and the propinquity value of the endpoints of this proposed edge is above a certain threshold, called the *emerging threshold*. Similarly, if the propinquity value of the endpoints of an existing edge is below a certain value, called the *cutting threshold*, then this edge is removed from the network. Since inserting and removing edges alters the network topology, it affects not only propinquity between individual nodes but also the overall propinquity of the entire topology. The propinquity of the new topology can then be calculated and used to guide the subsequent changes to the topology in the next iteration. Thus, the whole process called *propinquity dynamics* continues until the difference between topologies obtained in successive iterations becomes small relative to the whole network.

Since both topology and propinquity experience only relatively small changes from iteration to iteration, it is possible to perform the propinquity dynamics incrementally rather than recalculating all propinquity values in each iteration. Optimizations of performing incremental propinquity updates achieve a running time complexity of $O((|V| + |E|) \cdot |E|/|V|)$ for general networks and $O(|V|)$ for sparse networks.

It is also shown in [9] that community detection with propinquity dynamics can efficiently take advantage of parallel computation using message passing. Nodes are distributed among the processors which process them in parallel. Since it is essential that all nodes are in sync with each other, the bulk synchronous parallel (BSP) model is used to implement the parallel framework. In this model, the computation is organized as a series of *supersteps*. Each superstep consists of three major actions: receiving messages sent by other processors during the previous superstep, performing computation, and sending messages to other processors. Synchronization in BSP is explicit and takes the form of a barrier which gathers all the processors at the end of the superstep before continuing with the next superstep. Two types of messages are defined for the processors to communicate with each other. The first type is used to update propinquity maps that each processor stores locally for its nodes. Messages of the second type contain parts of the neighbor sets that a processor needs in its local computation.

A number of researchers explored a popular MapReduce parallel programming model to perform network mining operations. For example, a PeGaSus library (Peta-Scale Graph Mining System) described in [10] is built upon using Hadoop platform to perform several graph mining tasks such as PageRank calculations, spectral clustering, diameter estimation, and determining connected components. The core of PeGaSus is a GIM-V function (generalized iterated matrix-vector multiplication). GIM-V is capable of performing three operations: combining two values, combining all the values in the set, and replacing the old value with a new one. Since GIM-V is general, it is also quite universal. All other functions in the library are implemented as function calls to GIM-V with proper custom definitions of the three GIM-V operations. Fast algorithms for GIM-V utilize a number of optimizations like using data compression, dividing elements into blocks of fixed size, and clustering the edges. Finding connected components with GIM-V is essentially equivalent to community detection. The number of iterations required to find connected components is at most the diameter of the network. One iteration of GIM-V has the time complexity of $O(((|V| + |E|)/P) \log((|V| + |E|)/P))$ where P is the number of processors in the cluster. Running PeGaSus on an M45 Hadoop supercomputer cluster shows that GIM-V scales linearly with the number of machines increasing from 3 to 90. Accordingly, PeGaSus is able to reduce time execution on real-world networks containing up to hundreds of billions of edges from many hours to a few minutes.

A HEigen algorithm introduced in [11] is an eigensolver for large scale networks containing billions of edges. It is built upon the same MapReduce parallel programming model as PeGaSus and is capable of computing k eigenvalues for sparse symmetric matrices. Similar to PeGaSus, HEigen scales almost linearly with the number of edges and processors and performs well in up to a billion edge scale networks. Its asymptotic running time is the same as that of PeGaSus' GIM-V.

In [12] the authors consider a disjoint partitioning of a network into connected communities. They propose a massively parallel implementation of an agglomerative community detection algorithm that supports both maximizing modularity and minimizing conductance. The performance is evaluated on two different threaded hardware architectures: a multiprocessor multicore Intel-based server and massively multithreaded Cray XMT2. This approach is shown to scale well on two real-world networks with up to tens of millions of nodes and several hundred million edges.

Another method for partitioning a network into disjoint communities is scalable community detection (SCD) [13]. This two-phase algorithm works by optimizing the value of an objective function and is capable of processing undirected and unweighted graphs.

SCD uses the weighted community clustering (WCC) metric proposed in [14] as the objective function. Instead of performing simple edge counting, WCC works with more sophisticated graph structures, such as triangles. The quality of a partition is measured based on the number of triangles in a graph. Intuitively, more connections between the nodes within a community correspond to a larger number of triangles. Communities tend to have many highly connected nodes which are much more likely to close triangles with

each other rather than with nodes from other communities. Thus, for a particular node and community the value of WCC quantifies the level of cohesion of the node to the community. The metric is defined not only for individual nodes and communities but also for a community as a whole and the entire partition. One of the advantages of WCC over modularity is that it does not have a resolution limit problem. In addition, optimization of WCC is mathematically guaranteed to produce cohesive and structured communities. The measure of cohesion is defined for a partition as some real-valued function f called *degree of cohesion*. For each subset of nodes f assigns a value in the range $[0, 1]$ such that high values of f correspond to good communities and low values of f correspond to bad communities. For a given context (social, biological, etc.) an adequate metric f captures the features specific to this context. For example, for social networks the cohesion of a community depends on the number of triangles closed by the nodes inside this community. Furthermore, triangles are also used as a good indicator of community structure.

The operation of SCD consists of two phases which are executed sequentially. The first stage comprises graph cleanup and initial partitioning. Cleanup is performed by removing the edges which do not close any triangles. Then the graph is partitioned based on the values of the clustering coefficient of every node. Nodes are taken in the order of decreasing clustering coefficient and placed in communities together with their neighbors. Such partitioning yields communities with high values of WCC which is beneficial for the subsequent optimization process.

The second phase is responsible for the refinement of the initial partition. WCC optimization process consists of iterations which are repeated as long as the value of WCC for the entire partition keeps improving. In order to improve the value of WCC, the best of the following three movements is chosen for each node. These are the only movements which can potentially improve the WCC score:

(i) keep the network unchanged;

(ii) remove a node from its current community and place it in its own singleton community;

(iii) move a node from one community to another.

After movements for all the nodes have been selected, the WCC metric is calculated for the entire partition and compared to the previous value to determine if there is an overall improvement in the score. The refinement process stops when there has been no improvement (or improvement was less than a specified threshold) during the most recent iteration.

Computing the value of WCC directly at each iteration and for each node is computationally expensive and therefore should be avoided, especially for high degree nodes. In order to speed up calculations, it is possible to exploit the fact that the refinement process operates using the improvement of the score and therefore computing the absolute value of WCC is not necessary. Instead of calculating WCC directly, SCD uses certain graph statistics to build a WCC estimator. The estimator evaluates the improvement of WCC only once per iteration spending just $O(1)$ time per node.

Assuming that graphs have a quasilinear relation between the number of nodes and the number of edges, and the number of iterations of the refinement process is a constant, the overall running time complexity of SCD is $O(m \cdot \log n)$, where n is the number of nodes and m is the number of edges in the graph.

The advantage of the SCD algorithm is its amenability to parallelization. This is due to the fact that during the optimization process improvements of WCC are considered for every node individually and independently of other nodes. Therefore, the best movement can be calculated for all nodes simultaneously using whatever parallel features the underlying computing platform has to offer. Moreover, applying the moves to all nodes is also done in parallel.

SCD is implemented in C++ as a multithreaded application which uses OpenMP API for parallelization. Concurrency during the refinement process is achieved by considering improvements of WCC and then applying movements independently for each node. Benchmark datasets used in experiments include a range of networks of different sizes: Amazon, DBLP, Youtube, LiveJournal, Orkut, and Friendster. All of these graphs contain ground truth communities which are required to evaluate the quality of communities produced by SCD.

Normalized mutual information (NMI) and average F_1 score are used to evaluate the quality of community detection. SCD is compared against the following algorithms: Infomap, Louvain, Walktrap, BigClam, and Oslom. No distinction is made between methods which perform only disjoint community detection and those that are capable of detecting overlapping communities. The output of each algorithm is compared against ground truth communities without regard to possible overlaps. Although the values of NMI and average F_1 scores obtained for SCD are close to the results of other algorithms, it outperforms its competition on almost all datasets.

In terms of runtime performance, SCD is much faster than the majority of other algorithms used in the experiment. In a single threaded mode, the largest of the datasets used (Friendster) was processed in about 12 hours. SCD scales almost linearly with the number of edges in the graph. Using multiple threads can reduce the processing time even further. With 4 threads it takes a little bit over 4 hours to perform community detection on the Friendster network. Although the values of speedup are not explicitly presented, it can be inferred that the advantage of using multiple threads varies considerably depending on the dataset. The best case seems to be the Orkut graph for which speedup grows linearly as the number of threads is increased from 1 to 4. However, since the scope of parallelization in the experiment is modestly limited to just 4 threads, it is unclear how the scalability of the multithreaded SCD behaves when more than 4 cores are utilized.

A family of label propagation community detection algorithms includes label propagation algorithm (LPA) [15], community overlap propagation algorithm (COPRA) [16], and speaker-listener label propagation algorithm (SLPA) [17].

The main idea is to assign identifiers to nodes and then make them transmit their identifiers to their neighbors. With node identifiers treated as labels, a label propagation algorithm simulates the exchange of labels between connected nodes in the network. At each step of the algorithm each and every node that has at least one neighbor receives a label from one of its neighbors. Nodes keep a history of labels that they have ever received organized as a histogram which captures the frequency (and therefore the rank) of each label. The number of steps, or iterations, of the algorithm determines the number of labels each node accumulates during the label propagation phase. Being one of the parameters of the algorithm, the number of iterations eventually affects the accuracy of community detection. Clearly, the running time of the label propagation phase is linear with respect to the number of iterations. The algorithm is guaranteed to terminate after a prescribed number of iterations. When it does, communities data is extracted from nodes' histories.

Staudt and Meyerhenke [18] proposed PLP, PLM, and EPP algorithms for nonoverlapping community detection, that is, determining a partitioning of the node set.

Parallel label propagation (PLP) algorithm is a variation of the sequential LPA capable of performing detection of nonoverlapping communities in undirected weighted graphs. PLP differs from the original formulation of the label propagation algorithm [15] in that it avoids explicitly randomizing the node order and relies instead on asynchronism of concurrently executed PLP code threads. This way it saves the cost of explicit randomization. In order to optimize code execution even further, nodes are divided into active nodes and inactive nodes. Since labels of inactive nodes cannot be updated in the current iteration, the label propagation process is only performed on active nodes, thus reducing the amount of computation.

The termination criterion used by PLP is also different from the original description [15]. PLP uses a threshold value to stop processing. The value of the threshold is determined empirically and set to $n \cdot 10^{-5}$, where n is the number of nodes in the graph. Therefore, for the majority of graphs which were included in the experiment, the number of iterations is relatively small (from 2 to about a hundred). Moreover, no justification is provided for this formula which establishes a relation between the number of iterations and the number of nodes in the graph. Although it is claimed that "clustering quality is not significantly degraded by simply omitting these iterations," it is also admitted that "while the PLP algorithm is extremely fast, its quality might not always be satisfactory for some applications." No results are presented to show how the number of iterations affects the quality of community detection or how the modularity scores of PLP compare to those of the competition.

A locally greedy, agglomerative (bottom-up) multilevel community detection method called parallel Louvain method (PLM) is based on modularity optimization. Starting from some initial partition, nodes are moved from one community to another as long as it increases the objective function, that is, modularity. When modularity reaches a local optimum, a graph is coarsened and modularity optimization process is repeated.

Ensemble preprocessing (EPP) algorithm is a combination of several community detection methods. Its main goal is to form a classifier which decides if a pair of nodes should belong to the same community. EPP requires a preprocessing step which is performed by several parallel PLP instances running concurrently. The consensus of several base classifiers is used to form core communities which are coarsened to reduce the problem size.

Ensemble multilevel (EML) method is a recursive extension of the ensemble preprocessing algorithm. First, the core clustering is produced. Then the graph is contracted to a smaller graph, and the same algorithm is called recursively until a predefined termination condition is met.

All algorithms in [18] are created in C++. Parallel code is implemented using OpenMP API. Nodes are distributed between the threads and processed concurrently. Performance of the algorithms is compared to several other community detection methods: CLU_TBB, RG, CGGC, CGGCi, and the original sequential Louvain implementation. In order to compare the quality of results produced by different algorithms, Staudt and Meyerhenke use modularity [19]. Although modularity is very popular it was shown to suffer from the resolution limit and is also known to have other issues and limitations. There are other community quality metrics as well as modified versions of the original definition of modularity which overcome some of these problems [20]. However, modularity is the only measure used to compare the quality of communities produced by different algorithms in this experiment.

A shared-memory multiprocessor machine was used to test the performance and community quality of different algorithms. EML performed poorly while PLP and PLM were found to pay off with respect to either the execution time or community detection quality.

PLP was the fastest algorithm tested. It demonstrated linear strong scaling in the range 2–16 threads for uk-2002, the largest network which participated in all experiments. No data on scaling results for other datasets were provided. Since only one graph describes speedup for PLP, it is difficult to measure the values exactly, but they are approximately 0.92 for 2 threads (i.e., slower than with a single thread), 1.45 for 4 threads, 2.6 for 8 threads, and 4.6 for 16 threads. The running time drops in a slightly sublinear manner with the number of threads, although the absolute values of speedup are quite modest, and efficiency slowly goes down from 35% for 4 threads to 29% for 16 threads.

In almost all the cases, EPP was able to improve the values of modularity achieved by PLM. However, this advantage comes at the cost of running on average 10 times slower. At the same time, scalability of EPP remains unclear since no data is provided on the running time performance for different ensemble sizes.

For uk-2007-05 which was the largest graph used in the experiments, only the processing time of 120 seconds using the PLP algorithm and a parallel configuration with 32 threads is reported. No information is provided about scalability tests with this graph for other numbers of threads. In addition, due to memory constraints a different hardware platform with larger memory and a different CPU had to be

used to process this network. Therefore, the results are not directly comparable to those of other datasets. Although it is also mentioned that "a modularity of 0.99598 is reached for the uk-2007-05 graph in 660 seconds," it is not clear under which conditions this result was achieved. There is no mention of any other results concerning uk-2007-05, including any comparisons with other algorithms. Despite mentioning that uk-2007-05 requires "more than 250 GB of memory in the course of an EPP run," no EPP results for this graph are reported either.

In [21] we designed a multithreaded parallel community detection algorithm based on the sequential version of SLPA. Although only unweighted and undirected networks have been used to study the performance of our parallel SLPA implementation, an extension for the case of weighted and directed edges is straightforward and does not affect the computational complexity of the method. To facilitate such generalization, each undirected edge is represented with two directed edges connecting two nodes in opposite directions. In effect, the number of edges that are represented internally in code, is doubled, but the code is capable of running on directed graphs. A distinctive feature of our parallel solution is that, unlike other approaches described above, it is capable of performing overlapping community detection and has a parameter enabling balancing the running time and community detection quality.

In this paper, we further explore the multithreaded parallel programming paradigm that was used in [21] and test its performance on several real-world networks that range in size from several hundred thousand nodes and a few million edges to almost 5.5 million nodes and close to 170 million edges. We also provide a detailed analysis of the quality of communities detected with the parallel algorithm.

3. Parallel Linear Time Community Detection

The SLPA [17] is a sequential linear time algorithm for detecting overlapping communities. SLPA iterates over the list of nodes in the network. Each node i randomly picks one of its neighbors n_i and the neighbor then selects randomly a label l from its list of labels and sends it to the requesting node. Node i then updates its local list of labels with l. This process is repeated for all the nodes in the network. Once it is completed, the list of nodes is shuffled and the same processing repeats again for all nodes. After t iterations of shuffling and processing label propagation, every node in the network has a label list of length t, as every node receives one label in each iteration. After all iterations are completed, postprocessing is carried out on the list of labels and communities are extracted. We refer interested readers to the full paper [17] for more details on SLPA.

It is obvious that the sequence of iterations executed in SLPA algorithm makes the algorithm sequential and it is important for the list of labels updated in one iteration to be reflected in the subsequent iterations. Therefore, the nodes cannot be processed completely independently of each other. Each node is a neighbor of some other nodes; therefore, if the lists of labels of its neighbors are updated, it will receive a label randomly picked from the updated list of labels.

3.1. Multithreaded SLPA with Busy-Waiting and Implicit Synchronization. Our multithreaded implementation closely follows the algorithm described in [21] with minor improvements and bug fixes. In the multithreaded SLPA, we adopt a busy-waiting synchronization approach. Each thread performs label propagation on a subset of nodes assigned to this particular thread. This requires that the original network to be partitioned into subnetworks with one subnetwork to be assigned to each thread. Although partitioning can be done in several different ways depending on our objective, in this case the best partitioning will be that which makes every thread spend the same amount of time processing each node. Label propagation for any node consists of forming a list of labels by selecting a label from every neighbor of this node and then selecting a single label from this list to become a new label for this node. In other words, the ideal partitioning would guarantee that at every step of the label propagation phase each thread deals with a node that has exactly the same number of neighbors as nodes that are being processed by other threads. Thus, the ideal partitioning would divide the network in such a way that a sequence of nodes for every thread consists of nodes with the same number of neighbors across all the threads. Such partitioning is illustrated in Figure 1. $T_1, T_2, \ldots, T_p$ are p threads that execute SLPA concurrently. As indicated by the arrows, time flows from top to bottom. Each thread has its subset of nodes $n_{i1}, n_{i2}, \ldots, n_{ik}$ of size k where i is the thread number, and node neighbors are $m_1, m_2, \ldots, m_k$. A box corresponds to one iteration. There are t iterations in total. Dashed lines denote points of synchronization between the threads.

In practice, this ideal partitioning will lose its perfection due to variations in thread start-up times as well as due to uncertainty associated with thread scheduling. In other words, in order for this ideal scheme to work perfectly, hard synchronization of threads after processing every node is necessary. Such synchronization would be both detrimental to the performance and unnecessary in real-life applications.

Instead of trying to achieve an ideal partitioning we can employ a much simpler approach by giving all the threads the same number of neighbors that are examined in one iteration of the label propagation phase. It requires providing each thread with such a subset of nodes that the sum of all indegrees is equal to the sum of all indegrees of nodes assigned to every other thread. In this case, for every iteration of the label propagation phase every thread will examine the same overall number of neighbors for all nodes that are assigned to this particular thread. Therefore, every thread will be performing, roughly, the same amount of work per iteration. Moreover, synchronization then is only necessary after each iteration to make sure that no thread is ahead of any other thread by more than one iteration. Figure 2 illustrates such partitioning. As before, $T_1, T_2, \ldots, T_p$ are p threads that execute SLPA concurrently. As shown by the arrows, time

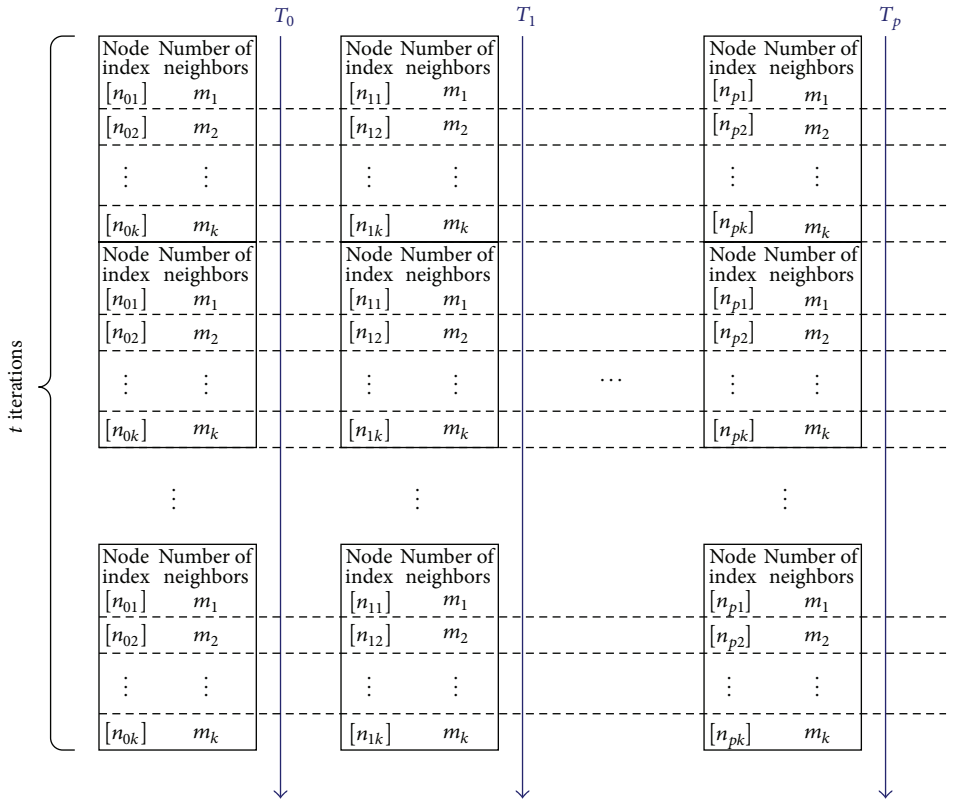

Figure 1: Ideal partitioning of the network for multithreaded SLPA.

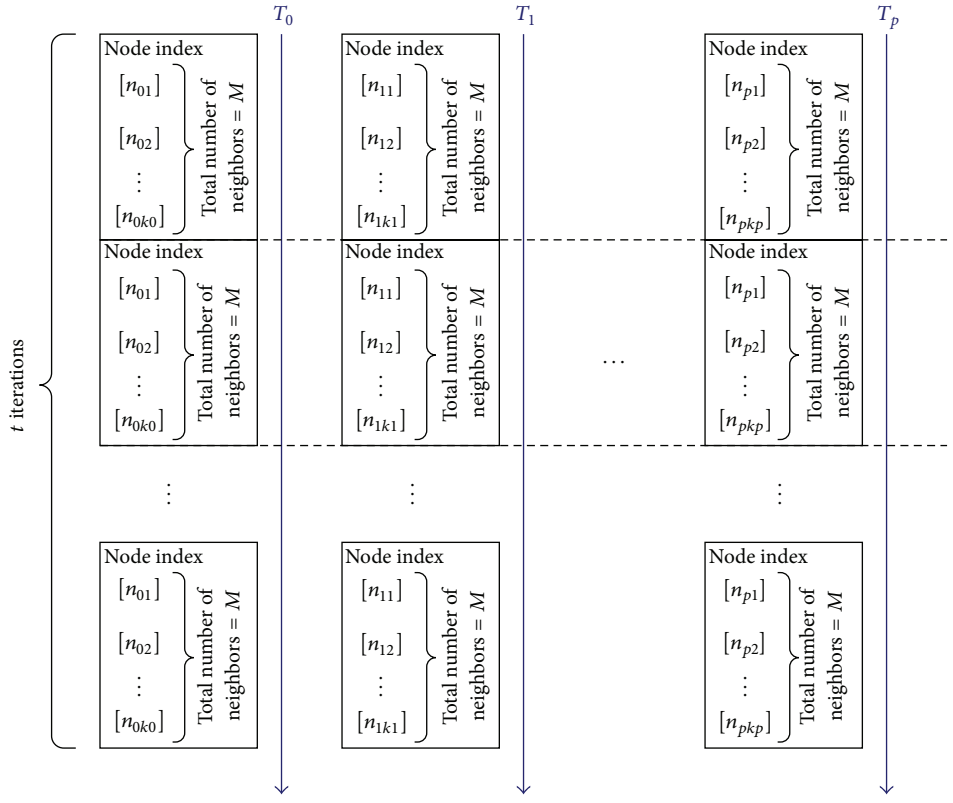

Figure 2: A better practical partitioning of the network for multithreaded SLPA.

flows from top to bottom. However, each thread now has its subset of nodes $n_{i1}, n_{i2}, \ldots, n_{ik_i}$ of size k_i where i is the thread number. In other words, threads are allowed to have different number of nodes that each of them processes, as long as the total number of node neighbors $M = \sum_{i=1}^{k_i} m_i$ is the same across all the threads. A box still corresponds to one iteration. There are t iterations in total. Dashed lines denote points of synchronization between the threads.

We can employ yet an even simpler approach of just splitting nodes equally between the threads in such a way that every thread gets the same (or nearly the same) number of nodes. It is important to understand that this approach is based on the premise that the network has small variation of local average of node degrees across all possible subsets of nodes of equal size. If this condition is met, then, as in the previous case, every thread performs approximately the same amount of work per iteration. Our experiments show that for many real-world networks this condition holds, and we accepted this simple partitioning scheme for our multithreaded SLPA implementation.

Given the choice of the partitioning methods described above, each of the threads running concurrently is processing all the nodes in its subset of nodes at every iteration of the algorithm. Before each iteration, the whole subset of nodes processed by a particular thread needs to be shuffled in order to make sure that the label propagation process is not biased by any particular order of processing nodes. Additionally, to guarantee the correctness of the algorithm, it is necessary to ensure that no thread is more than one iteration ahead of any other thread. The latter condition places certain restrictions on the way threads are synchronized. More specifically, if a particular thread is running faster than the others (for whatever reasons), it has to eventually pause to allow other threads to catch up (i.e., to arrive at a synchronization point no later than one iteration behind this thread). This synchronization constraint limits the degree of concurrency of this multithreaded solution.

It is important to understand the importance of partitioning the network nodes into subsets to be processed by the threads in respect to the distribution of edges across different network segments. In our implementation we use a very simple method of forming subsets of nodes for individual threads. First, a subset for the first thread is formed. Nodes are read sequentially from an input file. As soon as a new node is encountered, it is added to the subset of nodes processed by the first thread. After the subset of nodes for the first thread has been filled, a subset of nodes for the second thread is formed, and so on. Although simple and natural, this approach works well on networks with high locality of edges. For such networks, if the input file is sorted in the order of node numbers, nodes are more likely to have edges to other nodes that are assigned to the same thread. This leads to partitioning where only a small fraction (few percent) of nodes processed by each thread have neighbors processed by other threads.

Algorithm 1 shows the label propagation phase of our multithreaded SLPA algorithm which is executed by each thread. First, each thread receives a subset of nodes that it processes called *ThreadNodesPartition*. An array of dependencies *Used* is first initialized and then filled in such a way that it contains 1 for all threads that process at least one neighbor of the node from *ThreadNodesPartition* and 0 otherwise. This array of dependencies *Used* is then transformed to a more compact representation in the form of a dependency array D. Elements of array D contain thread numbers of all the threads which process any neighbor of a node that this thread processes. *Dsize* is the size of array D. If no node that belongs to the subset processed by this thread has neighbors processed by other threads, then array D is empty and $Dsize = 0$. If, for example, nodes that belong to the subset processed by this thread have neighbors processed by threads 1, 4, and 7, then array D has three elements with values of 1, 4, and 7, and $Dsize = 3$. After the dependency array has been filled, the execution flow enters the main label propagation loop which is controlled by counter t and has max T iterations. At the beginning of every iteration, we ensure that this thread is not ahead of the threads on which it depends by more than one iteration. If it turns out that it is ahead, this thread has to wait for the other threads to catch up. Then the thread performs a label propagation step for each of the nodes it processes which results in a new label being added to the list of labels for each of the nodes. Finally, the

```
ThreadPartition ← CreatePartition(InputFile)
p ← number of threads
for j = 1 to j < p do
  Used[j] ← 0
end for
for all v such that v is in ThreadNodesPartition do
  for all w such that w has an edge to v
    k ← getProcessorForNode(w)
    Used[k] ← 1
  end for
end for
Dsize ← 0
for j = 1 to j < p do
  if Used[j] > 0 then
    D[Dsize] ← j
    Dsize ← Dsize + 1
  end if
end for
while t < maxT do
  for j = 0 to j < Dsize − 1 do
    while t − t of thread D[j] > 1 do
      Do nothing
    end while
  end for
  for all v such that v is in myPartition do
    l ← selectLabel(v)
    Add label l to labels of v
  end for
  t ← t + 1
end while
```

Algorithm 1: Multithreaded SLPA.

iteration counter is incremented, and the next iteration of the loop is considered.

In order to even further alleviate the synchronization burden between the threads and minimize the sequentiality of the threads as much as possible, another optimization technique can be used. We note that some nodes which belong to a set processed by a particular thread have connection only to nodes that are processed by the same thread (we call them internal nodes), while other nodes have external dependencies. We say that a node has an external dependency when at least one of its neighbors belongs to a subset of nodes processed by some other thread. Since there are nodes with external dependencies, synchronization rules described above must be strictly followed in order to ensure correctness of the algorithm and meaningfulness of the communities it outputs. However, nodes with no external dependencies can be processed within a certain iteration independently from the nodes with external dependencies. It should be noted that a node with no external dependencies is not completely independent from the rest of the network since it may well have neighbors of neighbors that are processed by other threads.

It follows that processing of nodes with no external dependencies has to be done within the same iteration framework as for nodes with external dependencies but with less restrictive relations in respect to the nodes processed by other threads. In order to utilize the full potential of the technique described above, it is necessary to split the subset of nodes processed by a thread into two subsets, one of which contains only nodes with no external dependencies and the other one contains all the remaining nodes. Then, during the label propagation phase of the SLPA, nodes that have external dependencies are processed first in each iteration. Since we know that by the time such nodes are processed the remaining nodes (those with no external dependencies) cannot influence the labels propagated to nodes processed by other threads (due to the symmetry of the network), it is safe to increment the iteration counter for this thread, thus allowing other threads to continue their iterations if they have been waiting for this thread in order to be able to continue. Meanwhile, this thread can finish processing nodes with no external dependencies and complete the current iteration.

This approach effectively allows a thread to report completion of the iteration to the other threads sooner than it has been completed by relying on the fact that the work which remains to be completed cannot influence nodes processed by other threads. This approach, though seemingly simple and intuitive, leads to noticeable improvement of the efficiency of parallel execution (as described in Section 3.2) mainly due to decreasing the sequentiality of execution of multiple threads by signaling other threads earlier than in the absence of such splitting.

An important peculiarity arises when the number of nodes with external dependencies is only a small fraction (few percent) of all the nodes processed by the thread. In this case it would be beneficial to add some nodes without external dependencies to the nodes with external dependencies and process them together before incrementing the iteration counter. The motivation here is that nodes must be shuffled in each partition separately from each other to preserve the order of execution between partitions. Increasing partition size above the number of external nodes improves shuffling in the smaller of the two partitions.

```
Internal ← CreateInternalPartition(InputFile)
External ← CreateExternalPartition(InputFile)
p ← number of threads
/* Unchanged code from Algorithm 1 omitted */
while t < maxT do
  for j = 0 to j < Dsize − 1 do
    while t − t of thread D[j] > 1 do
      Do nothing
    end while
  end for
  for all v such that v is in External do
    l ← selectLabel(v)
    Add label l to labels of v
  end for
  t ← t + 1
  for all v such that v is in Internal do
    l ← selectLabel(v)
    Add label l to labels of v
  end for
end while
```

Algorithm 2: Multithreaded SLPA with splitting of nodes.

The remaining nodes without external dependencies can be processed after incrementing the iteration counter, as before. In order to reflect this optimization factor we introduce an additional parameter called the splitting ratio. A value of this parameter indicates the percentage of nodes processed by the thread before incrementing the iteration counter. For instance, if we say that splitting of 0.2 is used it means that at least 20% of nodes are processed before incrementing the iteration counter. If after initial splitting of nodes into two subsets of nodes with external dependencies and without external dependencies it turns out that there are too few nodes with external dependencies to satisfy the splitting ratio, some nodes that have no external dependencies are added to the group of nodes with external dependencies just to bring the splitting ratio to the desired value.

Algorithm 2 shows our multithreaded SLPA algorithm that implements splitting of nodes processed by a thread into a subset of nodes with external dependencies and a subset with no external dependencies. The major difference from Algorithm 1 is that, instead of processing all the nodes before incrementing the iteration counter, we first process a subset of nodes that includes nodes that have neighbors processed by other threads, then we increment the iteration counter, and then we process the rest of the nodes.

Since in [21] we studied the impact of selecting different values of the splitting ratio, it was not our main focus here. We simply accepted a splitting ratio of 0.2 and kept it fixed for all the test runs. Our major objective was to ensure that all parallel and sequential runs are performed with exactly the same code base and provide identical runtime

conditions and parameters, so that results of our performance evaluation and community detection quality metrics are directly comparable.

3.2. Performance Evaluation of the Multithreaded Solution. We performed runs on a hyper-threaded Linux system operating on top of a Silicon Mechanics Rackform nServ A422.v3 machine. Processing power was provided by 64 cores organized as four AMD Opteron 6272 central processing units (2.1 GHz, 16-core, G34, 16 MB L3 Cache) operating over a shared 512 GB bank of random access memory (RAM) (32 × 16 GB DDR3-1600 ECC Registered 2R DIMMs) running at 1600 MT/s Max. The source code was written in C++03 and compiled using g++ 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5).

Four datasets have been used to test the performance of the multithreaded solution and the quality of community detection. Three of these datasets (com-Amazon, com-DBLP, and com-LiveJournal) have been acquired from Stanford Large Network Dataset Collection (http://snap.stanford.edu/data/) which contains a selection of publicly available real-world networks (SNAP networks).

Undirected Amazon product copurchasing network (referred to as com-Amazon) was gathered, described, and analyzed in [22]. From the dataset information [23], it follows that it was collected by crawling Amazon website. A *Customers Who Bought This Item Also Bought* feature of the Amazon website was used to build the network. If it is known that some product i is frequently bought together with product j, then the network contains an undirected edge from i to j. For each product category defined by Amazon, there is a corresponding ground truth community. Each connected component in a product category is treated as a separate ground truth community.

Since small ground truth communities having less than 3 nodes had been removed, it was necessary to modify the original com-Amazon network to ensure that only nodes that belong to ground truth communities can appear in communities detected by the multithreaded parallel algorithm. Otherwise, comparison of communities produced by the community detection algorithm and the ground truth communities would not be feasible. The modified com-Amazon network was obtained from the original one by removing nodes which are not found in any ground truth community and all the edges connected to those nodes. While the original Amazon network consists of 334,863 nodes and 925,872 undirected edges, the modified dataset has 319,948 nodes and 1,760,430 directed edges. As outlined in Section 2, each undirected edge is internally converted to a pair of edges. Therefore, 925,872 undirected edges from the original network correspond to 1,851,744 directed edges in the internal representation of the code, and since some of the edges were incident to removed nodes, the resulting number of directed edges left in the network was 1,760,430.

The DBLP computer science bibliography network (referred to as com-DBLP) was also introduced and studied in [22]. According to the dataset information [24], it provides a comprehensive list of research papers in computer science. If two authors publish at least one paper together, then the nodes corresponding to these authors will be connected with an edge in a coauthorship network. Ground truth communities are based on authors who published in journals or conferences. All authors who have at least one publication in a particular journal or conference form a community. Similarly to the com-Amazon network, each connected component in a group is treated as a separate ground truth community. Small ground truth communities (less than 3 nodes) have also been removed.

The DBLP dataset was also modified to facilitate comparison with ground truth communities as described above for the com-Amazon network. Since DBLP is also undirected, the same considerations about the number of edges that were provided above for the com-Amazon network also apply to com-DBLP. The original DBLP network contains 317,080 nodes and 1,049,866 undirected edges, while the modified version has 260,998 nodes and 1,900,118 directed edges.

Another network from [22] that we are using to evaluate the performance of the multithreaded parallel implementation of SLPA and the quality of communities it produces is a LiveJournal dataset (referred to as com-LiveJournal). The dataset information page [25] describes LiveJournal as a free online blogging community where users declare friendship with each other. LiveJournal users can form groups and allow other members to join them. For the purposes of evaluating the quality of communities we are treating the com-LiveJournal network as having no ground truth communities. The LiveJournal network is undirected and contains 3,997,962 nodes and 34,681,189 pairs of directed edges. Since we are not comparing the communities found by the community detection algorithm with the ground truth communities, no modification of the original network is necessary.

The fourth dataset is a snapshot of the Foursquare network as of October 11, 2013. This dataset contains 5,499,157 nodes and 169,687,676 edges. No information about ground truth communities is available.

We calculated speedup using the formula shown in (1) and efficiency according to (2):

$$\text{Speedup} = \frac{T_1}{T_p}, \tag{1}$$

where Speedup is the actual speedup calculated according to (1) and p is the number of processors or computing cores:

$$\text{Efficiency} = \frac{\text{Speedup}}{p}. \tag{2}$$

All the experiments were run with 1,000 iterations (the value of $\max T$ was set to 1000) for all networks. On one hand, a value of 1,000 for the number of iterations provides a sufficient amount of work for the parallel portion of the algorithm, so that the overhead associated with creating and launching multiple threads does not dominate the label propagation running time. On the other hand, 1,000 iterations are empirically enough to produce meaningful communities since the number of labels in the history of every label is statistically significant. At the same time,

although running the algorithm for 1,000 iterations on certain datasets (especially larger ones) was in some cases (mainly for smaller core counts) taking a few days, it was still feasible to complete all runs on all four networks in under two weeks.

We conducted one set of measurements by considering only time for the label propagation phase since it is at this stage that our multithreaded implementation differs from the original sequential version. Time necessary to read an input file and construct in-memory representation of the nodes and edges as well as any auxiliary data structures was not included in this timing. All postprocessing steps and writing output files have also been excluded.

However, for an end user it is not the label propagation time (or any other single phase of the algorithm) that is important but rather the total running time. Users care about the time it took for the code to run: from the moment a command was issued until the resulting communities files have been written to a disk. Therefore, we conducted a second set of measurements to gather data on total execution time of our multithreaded parallel SLPA implementation. Since the total execution time includes not only a highly parallel label propagation stage but also file I/O, threads creation and cleanup, and other operations which are inherently sequential, it is to be expected that the values of both speedup and efficiency are going to be worse than in the case when only label propagation phase is considered.

Since the hardware platform we used provides 64 cores, every thread in our tests executes on its dedicated core. Therefore, threads do not compete for central processing unit (CPU) cores (unless there is interference from the operating system or other user processes running concurrently). They are executed in parallel, and we can completely ignore thread scheduling issues in our considerations. Because of this, we use terms "thread" and "core" interchangeably when we describe results of running the multithreaded SLPA. The number of cores in our runs varies from 1 to 64. However, we observed a performance degradation when the number of threads is greater than 32. This performance penalty is most likely caused by the memory banks organization of our machine. Speedup and efficiency are calculated using (1) and (2) defined earlier. No third-party libraries or frameworks have been used to set up and manage threads. Our implementation relies on Pthreads application programming interface (POSIX threads) which has implementations across a wide range of platforms and operating systems.

We noticed that the compiler version and compilation flags can each play a crucial role not only in terms of how efficiently the code runs but also in terms of the sole ability of code to execute in the multithreaded mode. Unfortunately, little, if anything is clearly and unambiguously stated in compiler documentation regarding implications of using various compiler flags to generate code for execution on multithreaded architectures. For the most part, developers have to rely on their own experience or common sense and experiment with different flags to determine the proper set of options which would make the compiler generate effective code capable of flawlessly executing multiple threads.

For instance, when the compiler runs with either -O2 or -O3 optimization flag to compile the multithreaded SLPA

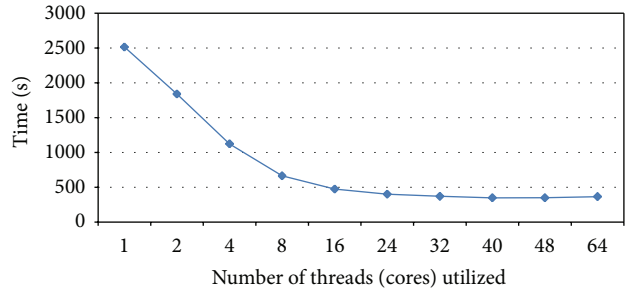

Figure 3: Label propagation time for com-Amazon network at different number of cores.

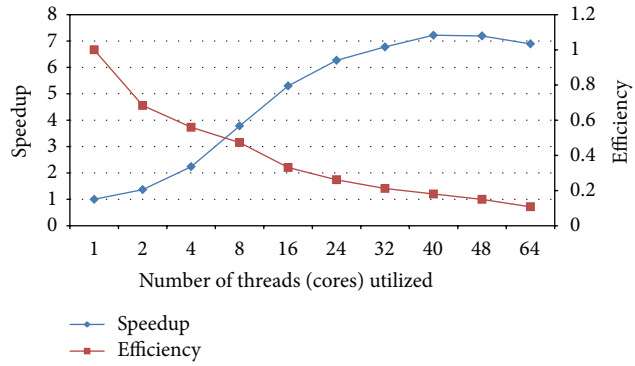

Figure 4: Speedup and efficiency for com-Amazon network (considering only label propagation time) at different number of cores.

the resulting binary code simply deadlocks at execution. The reason for deadlock is exactly the optimization that compiler performs ignoring the fact that the code is multithreaded. This optimization leads to threads being unable to see updates to the shared data structures performed by other threads. In our case such shared data structure is an array of iteration counters for all the threads. Evidently, not being able to see the updated values of other threads' counters quickly leads threads to a deadlock.

Another word of caution should be offered regarding some of the debugging and profiling compiler flags. More specifically, compiling code with -pg flag which generates extra code for a profiling tool *gprof* leads to substantial overhead when the code is executed in a multithreaded manner. The code seems to be executing fine but with a speedup of less than 1. In other words, the more threads are being used the longer it takes for the code to run regardless of the fact that each thread is executed on its own core and therefore does not compete with other threads for CPU. It is also counterintuitive since using more threads should result in a smaller subset of nodes that each thread processes.

The results of performance runs of our multithreaded parallel implementation are presented in Figures 3–19. (Data export was performed using Daniel's XL Toolbox add-in for Excel, version 6.51, developed by Daniel Kraus, Würzburg, Germany.)

Figures 3, 5, 7, and 9 show the time it took to complete the label propagation phase of the multithreaded parallel

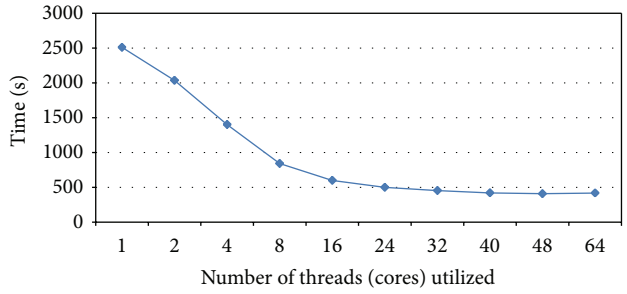

Figure 5: Label propagation time for com-DBLP network at different number of cores.

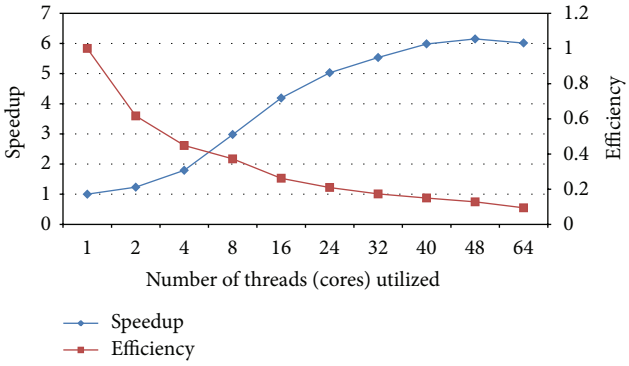

Figure 6: Speedup and efficiency for com-DBLP network (considering only label propagation time) at different number of cores.

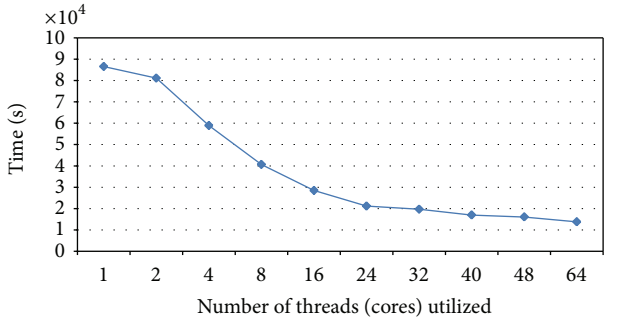

Figure 7: Label propagation time for com-LiveJournal network at different number of cores.

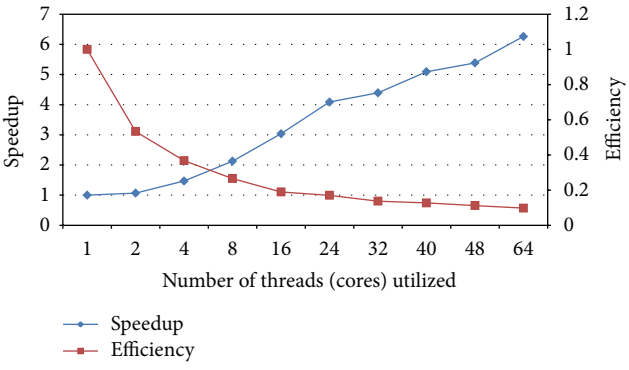

Figure 8: Speedup and efficiency for com-LiveJournal network (considering only label propagation time) at different number of cores.

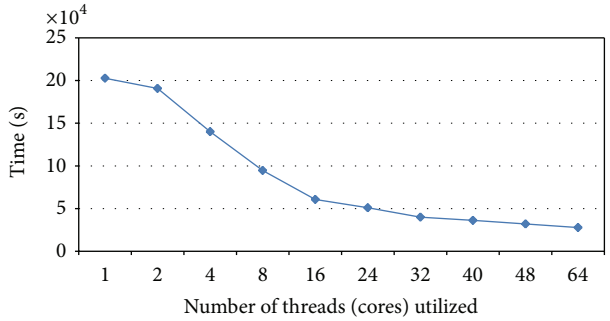

Figure 9: Label propagation time for Foursquare network at different number of cores.

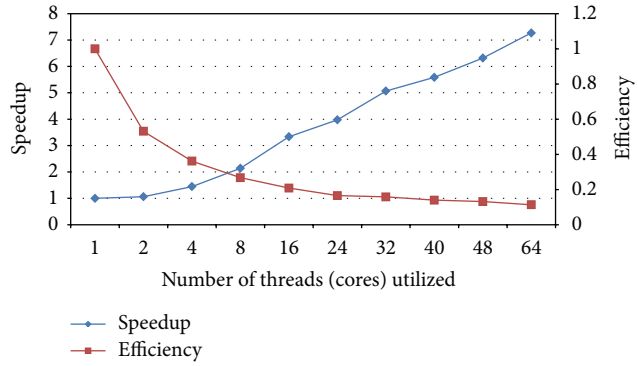

Figure 10: Speedup and efficiency for Foursquare network (considering only label propagation time) at different number of cores.

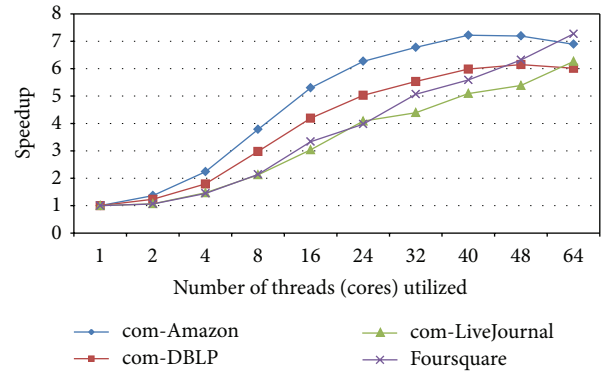

Figure 11: Speedup for all datasets (considering only label propagation time) at different number of cores.

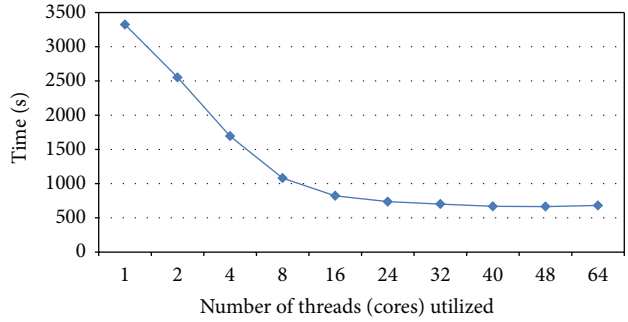

Figure 12: Total execution time for com-Amazon network at different number of cores.

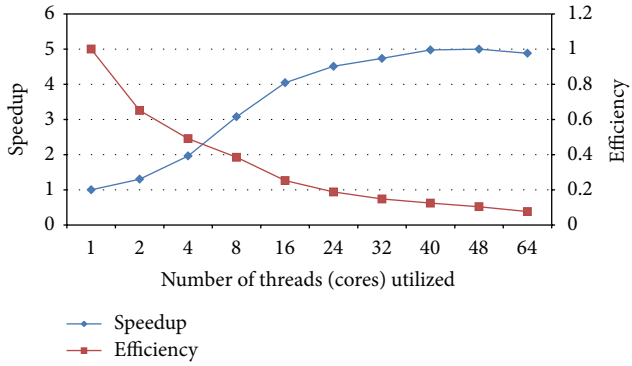

Figure 13: Speedup and efficiency for com-Amazon network (considering total execution time) at different number of cores.

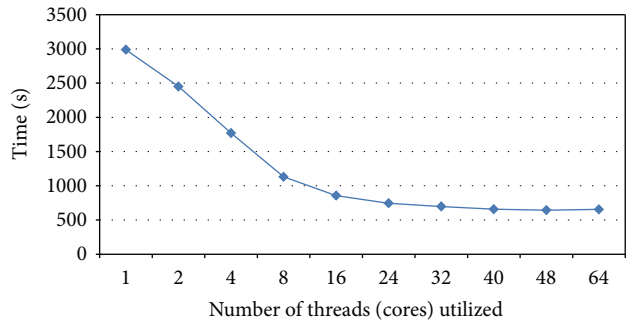

Figure 14: Total execution time for com-DBLP network at different number of cores.

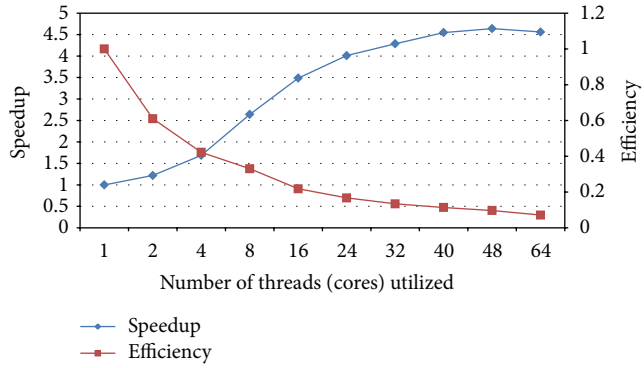

Figure 15: Speedup and efficiency for com-DBLP network (considering total execution time) at different number of cores.

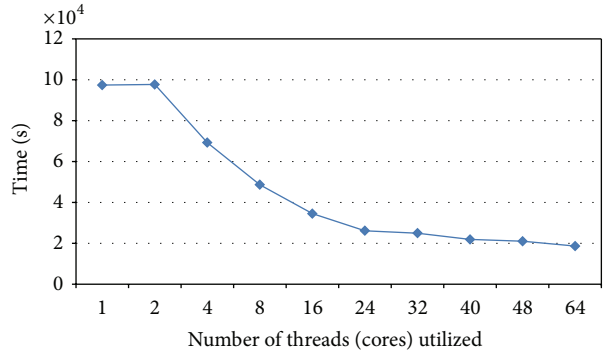

Figure 16: Total execution time for com-LiveJournal network at different number of cores.

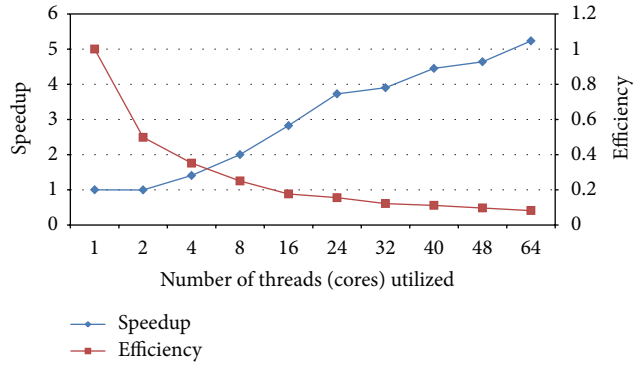

Figure 17: Speedup and efficiency for com-LiveJournal network (considering total execution time) at different number of cores.

SLPA on four datasets (com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, resp.) for the number of cores varying from 1 to 64. It can be seen that for smaller core counts the time decreases nearly linearly with the number of threads. For larger number of cores the label propagation time continues to improve but at a much slower rate. In fact, for 32 or more cores, there is almost no improvement of the label propagation time on smaller datasets (com-Amazon and com-DBLP). At the same time, larger datasets (com-LiveJournal and Foursquare) improve label propagation times all the way through 64 cores. As outlined in Section 1, this is clearly something to be expected since in a strong scaling setting enough workload should be supplied to parallel processes to compensate for the overhead of creating multiple threads and maintaining communication between them.

This trend is even more evident in Figures 4, 6, 8, and 10 which plot the values of speedup and efficiency for the four datasets (com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, resp.) and the number of cores from 1 to 64. As the number of cores increases, the speedup also grows but not as fast as the number of utilized cores, so efficiency drops. The saturation of speedup is quite evident for smaller networks (com-Amazon and com-DBLP) and corresponds to regions with no improvement of the label propagation time that we noticed earlier. Similarly, the values of speedup continue to improve (although at decreasing rates) for larger datasets (com-LiveJournal and Foursquare) even at 64 cores. Nonetheless, the efficiency degrades since speedup gains are small relative to an increase in core count. Such behavior can be attributed to several factors. First of all, as the number of cores grows while the network (and hence the number of nodes and edges) stays the same, each thread gets fewer nodes and edges to process. Approaching the limit of the thread size can cause the overhead of creating and running threads to outweigh the benefits of parallel execution for a sufficiently small thread size. Furthermore, as the number of cores grows, the number of neighbors of nodes with external dependencies

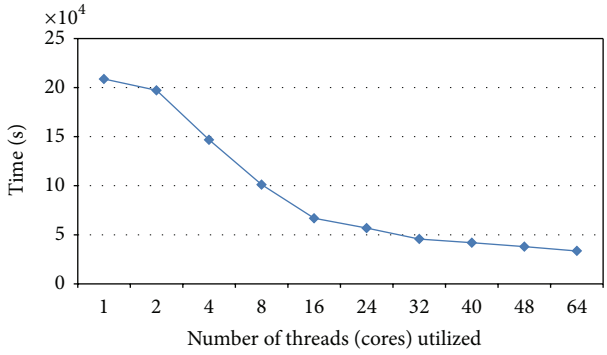

Figure 18: Total execution time for Foursquare network at different number of cores.

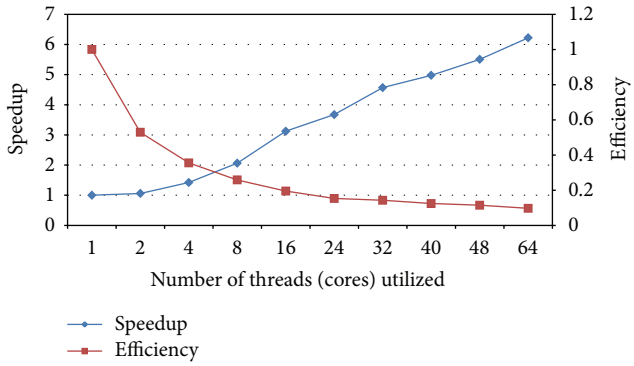

Figure 19: Speedup and efficiency for Foursquare network (considering total execution time) at different number of cores.

increases (both because each thread gets fewer nodes and there are more threads to execute them). More nodes with external dependencies, in turn, means that threads are more dependent on other threads.

However, for the sake of fair data interpretation, we need to remember that the definition of efficiency which we are using here is based on (2). It only takes into account the parallel execution speedup observed on a certain number of cores. The cost of cores is completely ignored in this formula. More realistically, the cost should be considered as well. The price paid for a modern computer system is not linear with the number of processors and cores. Within a certain range of the number of cores per system as the architecture moves towards higher processor and core counts, each additional core costs less. That is why the pure parallel efficiency defined by (2) should be effectively multiplied by the cost factor for making decisions regarding the choice of hardware to run community detection algorithms on real-life networks. After such multiplication, the efficiency including cost is going to be much more favorable to higher core counts than the efficiency given by (2).

Figure 11 combines plots of speedup values based on the label propagation time for all four datasets. Overall, the values of speedup do not vary considerably between the networks used in the experiments. However, it is quite evident that the shape of the curves is slightly different. On one hand, there is com-Amazon and com-DBLP for which the values of speedup reach local maximum at fewer than maximal number of cores. On the other hand, speedup values for com-LiveJournal and Foursquare are strictly increasing as the number of cores ranges between 1 and 64.

This observation is just additional evidence of the behavior discussed earlier. Smaller networks are too small to effectively use large core counts which leads to the saturation of speedup. The performance of multithreaded parallel SLPA on larger datasets continues to improve at almost a constant rate in a wide range of core counts between 4 and 64. It is also worth noting that, as long as a network is large enough to justify the overhead of multithreaded execution, different datasets yield almost identical speedup values. Although more testing would be required to firmly assert that speedup is independent of the size of the dataset, such behavior would be easy to explain. Indeed, speedup performance of the algorithm depends primarily on the properties of the graph (e.g., the number of edges crossing the boundary between the node sets processed by different cores) rather than on the size of the network. Such a feature is quite desirable in community detection since it enables the application to provide a user with an estimate of the overall execution time once the network is loaded and partitioned between the cores.

Figures 12, 14, 16, and 18 present the total community detection time of the multithreaded parallel SLPA on four datasets (com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, resp.) for the number of cores varying from 1 to 64. Although clearly the total running time exceeds the label propagation phase, the difference in many cases is not that significant. This is especially true for larger datasets (com-LiveJournal and Foursquare) which, as we discussed above, is something to be expected. The fact that the label propagation phase is a dominating component of the total running time justifies our efforts to increase performance by replacing sequential label propagation with a parallel implementation.

The values of speedup and efficiency calculated based on the total execution time rather than label propagation time are plotted in Figures 13, 15, 17, and 19 for the four datasets (com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, resp.) and the number of cores between 1 and 64. Although these values are worse than those calculated based only on the label propagation time, they provide a more realistic view of the end-to-end performance of our multithreaded SLPA implementation. In real life the speedup values of around 5 to 6 still constitute a substantial improvement over the sequential implementation, meaning, for example, that you would only have to spend 8 hours waiting for your community detection results instead of 2 days.

Figure 20 shows combined plots of speedup values for all four datasets considering the total execution time. Just like in Figure 11, the values of speedup for different networks are quite similar. The same two types of curves are observed which correspond to a group of relatively small (com-Amazon and com-DBLP) and large (com-LiveJournal and Foursquare) networks.

However, there are some differences. First, the absolute values of speedup are lower when we consider the total

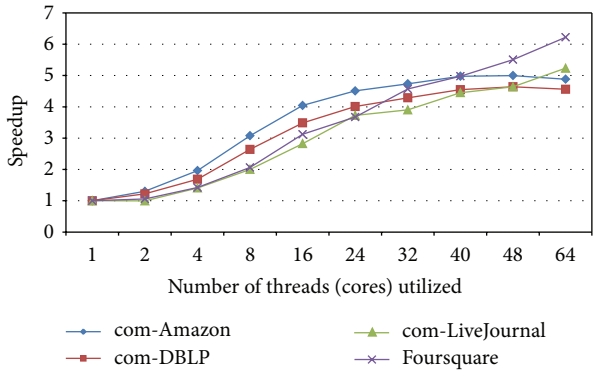

Figure 20: Speedup for all datasets (considering total execution time) at different number of cores.

execution time instead of just the label propagation phase. This is clearly something to be expected since the total execution time includes many operations (e.g., reading the input graph and writing output communities, partitioning the network between the cores, etc.) which cannot be made efficiently parallel. Second, the difference in speedup for different datasets even within the same group (e.g., large datasets) is greater than it was in Figure 11. The reason for that is also the effect of the limiting factor of sequential operations. Since we are considering the total execution time here, the size of the dataset affects speedup more significantly than in the case when only label propagation time was taken into account.

4. The Quality of Community Detection

In this section, we will evaluate the quality of the community structure detected with multithreaded version of SLPA [17] on the four datasets, Amazon, DBLP, Foursquare, and LiveJournal, introduced in Section 3.2. Amazon and DBLP have ground truth communities, while Foursquare and LiveJournal do not. Our only concern here is whether the community structure discovered by multithreaded SLPA has the quality similar to that detected by sequential SLPA [17] since we have already shown the effectiveness of sequential SLPA, compared with other community detection algorithms, in [17, 26]. Each metric value in Tables 1 and 2 is the average of results from ten runs of the community detection algorithm. The tested values of threshold r of SLPA are $r = 0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4$, and 0.45.

We calculate *variation of information* (*VI*), *normalized mutual information* (*NMI*), *F-measure*, and *normalized Van Dongen metric* (*NVD*) [20] of the community structures detected by sequential SLPA and multithreaded SLPA on Amazon and DBLP, presented in Tables 1 and 2. Notice that *VI*, *NMI*, *F-measure*, and *NVD* are intended to measure the quality of disjoint communities. However, we could still use them here to evaluate the quality of overlapping communities, although the values of *NMI*, *F-measure*, and *NVD* may not be in the range of [0, 1]. There are mainly two reasons why we adopt their disjoint versions. On one hand,

Table 1: Metric values of the community structures detected by sequential SLPA and multithreaded SLPA on Amazon (bold font denotes the best value for each metric).

Algorithm	*VI*	*NMI*	*F*-measure	*NVD*
Sequential SLPA	**65.2664**	1.6113	**1.5318**	**−0.5647**
Multithreaded SLPA	65.4445	**1.6132**	1.5034	−0.5552

Table 2: Metric values of the community structures detected by sequential SLPA and multithreaded SLPA on DBLP (bold font denotes the best value for each metric).

Algorithm	*VI*	*NMI*	*F*-measure	*NVD*
Sequential SLPA	**30.4591**	0.963	**0.4112**	**0.4306**
Multithreaded SLPA	30.8962	**0.9675**	0.4029	0.4521

we are only concerned whether multithreaded SLPA has almost the same performance with sequential SLPA, in other words, whether communities detected by sequential and parallel runs have values of *VI*, *NMI*, *F-measure*, and *NVD* close to each other. On the other hand, *VI*, *F-measure*, and *NVD* do not have definitions for overlapping communities yet. *NMI* has its overlapping version [27], but it takes a very long time to calculate its value on large networks, like Amazon and DBLP. It can be seen from Tables 1 and 2 that the metric values of the community structures detected by sequential SLPA and multithreaded SLPA on Amazon and DBLP are very close to each other, which indicates that multithreaded SLPA has almost the same performance with sequential SLPA on Amazon and DBLP.

We then compute modularity (Q) [19], *intradensity*, *contraction*, *expansion*, and *conductance* [22, 28] of the community structures found by sequential SLPA and multithreaded SLPA on Foursquare and LiveJournal, presented in Tables 3 and 4. Notice that the modularity we adopt here is also applicable to disjoint communities, so its value may not be in the range of $[0, 1]$. The reasons for using the disjoint version of modularity echo those given in the case of *VI*, *NMI*, *F-measure*, and *NVD* metrics. In addition, there are several overlapping versions for modularity [29–34], and it is not clear which one is the best. Tables 3 and 4 show that the metric values of the community structures detected by sequential SLPA and multithreaded SLPA on Foursquare and LiveJournal are also very close to each other, which implies that multithreaded SLPA has almost the same performance with sequential SLPA on Foursquare and LiveJournal.

Comparisons between different community detection algorithms are not always easy to make due to substantially different implementations which might even require mutually exclusive architectural features or software components (shared-memory versus distributed memory machines, programming languages compiled to native code versus development systems based on virtual machines or interpretation, and so on).

Table 3: Metric values of the community structures detected by sequential SLPA and multithreaded SLPA on Foursquare (bold font denotes the best value for each metric).

Algorithm	Q	Intradensity	Contraction	Expansion	Conductance
Sequential SLPA	0.7608	**0.3651**	**3.6683**	**2.5137**	**0.3849**
Multithreaded SLPA	**0.7682**	0.3535	3.5766	2.6358	0.4055

Table 4: Metric values of the community structures detected by sequential SLPA and multithreaded SLPA on LiveJournal (bold font denotes the best value for each metric).

Algorithm	Q	Intradensity	Contraction	Expansion	Conductance
Sequential SLPA	0.6834	**0.3174**	**4.4735**	**2.4332**	**0.3777**
Multithreaded SLPA	**0.6929**	0.2969	4.0367	2.8901	0.4333

It is also important to consider the type of output communities that an algorithm can produce. As mentioned earlier, overlapping community detection is more computationally intensive than disjoint. While the majority of other parallel solutions perform only disjoint community detection, our multithreaded SLPA can produce either disjoint or overlapping communities, depending on the value of threshold r.

Even though execution time is certainly one of the most important performance measures for an end user, it is often not suitable for direct comparisons between different implementations of community detection methods. Unlike execution time which depends on specific hardware, operating systems, code execution environments, compiler optimizations, and other factors, speedup evens out architectural and algorithmic differences. It is therefore a much better way to compare runtime performance of community detection algorithms.

Another important factor that makes it hard to compare the results produced by competing methods is the use of different datasets. Although several datasets seem to appear more often than the others (e.g., Amazon, DBLP, and LFR) there is no established set of datasets which are publicly available and widely accepted as a benchmark for high performance community detection. If such a benchmark existed, it should have contained a balanced blend of both real-world and synthetic datasets of varying size (from hundreds of thousands of nodes and edges to billion scale networks) carefully selected so that it does not give a priori advantage to any of the possible approaches to community detection.

There are datasets which are supplied with so-called ground truth communities, although in some cases it is very questionable whether these communities in fact represent the ground truth. For other networks, it is not feasible to establish the ground truth at all. Again, there is no established consensus on whether datasets with or without ground truth communities (or a combination of both types) should be evaluated. Different researchers approach this problem differently, mainly depending on the datasets to which they have access. There is also a problem of using proprietary datasets which might not be available to other researchers to test their community detection implementations.

Besides using different datasets, researchers also use different metrics to evaluate the quality of community detection. A decade or so ago, modularity was the dominating player on the community quality field. However, after it was discovered that the original formulation of modularity suffers from several drawbacks, a number of new or extended metrics have been proposed and a number of old, almost forgotten methods have been rediscovered. A detailed review of different existing and emerging metrics can be found in [20]. Still, there is no agreement on which metric (or combination of metrics) should be chosen as an authoritative measure of the quality of community detection performed by a certain algorithm.

From all of the above, it follows that performing fair comparisons of different community detection implementations is difficult. To take just one example, let us consider PLP/EPP, SCD, and multithreaded SLPA.

(i) Both PLP/EPP and SCD methods (see Section 2) are only able to detect disjoint communities while multithreaded SLPA performs overlapping community detection.

(ii) Experiments with SCD were only conducted with the number of threads ranging from 1 to 4. In contrast, in our approach described in Section 3, we evaluate the method and show its scalability for all datasets being tested, including large graphs, and the number of cores ranging from 1 to 64. PLP was tested for a slightly wider range of parallel configurations (1 to 16 threads) but only for one dataset, uk-2002. For the Foursquare network which is similar in size to uk-2002, the values of speedup demonstrated by multithreaded SLPA (see Figure 19) are comparable to the results of SCD described in Section 2.

(iii) Modularity is the only measure of community quality considered by PLP/EPP. SCD uses *NMI* and average F_1 score. Multithreaded SLPA uses several different metrics, including *NMI* and *F-measure*. However, for the reasons explained above the values of *NMI* and *F-measure* may not be in the conventional range of $[0, 1]$. Therefore, it is not feasible to directly compare the values of community quality metrics obtained in our experiments with the SCD results.

In conclusion, the community structure found by multithreaded SLPA has almost the same quality as that discovered by sequential SLPA. Moreover, we have demonstrated in [17, 26] that sequential SLPA is very competitive compared to other community detection algorithms, which implies the effectiveness of multithreaded SLPA on community detection.

5. Conclusion and Future Work

In this paper, we evaluated the performance of a multithreaded parallel implementation of SLPA and showed that using modern multiprocessor and multicore architectures can significantly reduce the time required to analyze the structure of different networks and output communities. We found that despite the fact that the rate of speedup slows down as the number of processors is increased, it still pays off to utilize as many cores as the underlying hardware has available. Our multithreaded SLPA implementation was proven to be scalable in terms of both increasing the number of cores and analyzing increasingly larger networks. Furthermore, the properties of the detected communities closely match those produced by the base sequential algorithm, as verified using several metrics. Given a sufficient number of processors, the parallel SLPA can reliably process networks with millions of nodes and accurately detect meaningful communities in minutes and hours.

In our future work, we plan to explore other parallel programming paradigms and compare their performance with our multithreaded approach.

Disclaimer

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the US Government.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The research was sponsored in part by the Army Research Laboratory under Cooperative Agreement no. W911NF-09-2-0053, by the EU's 7FP Grant Agreement no. 316097, and by the Polish National Science Centre, the Decision no. DEC-2013/09/B/ST6/02317.

References

[1] R. E. Park, *Human Communities: The City and Human Ecology*, vol. 2, Free Press, 1952.

[2] J. Xie, S. Sreenivasan, G. Korniss, W. Zhang, C. Lim, and B. K. Szymanski, "Social consensus through the influence of committed minorities," *Physical Review E*, vol. 84, no. 1, Article ID 011130, 8 pages, 2011.

[3] P. Sah, L. O. Singh, A. Clauset, and S. Bansal, "Exploring community structure in biological networks with random graphs," BioRxiv, 2013.

[4] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, pp. 814–818, 2005.

[5] J. A. Hartigan and M. A. Wong, "Algorithm as 136: a k-means clustering algorithm," *Journal of the Royal Statistical Society Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[6] J. Baumes, M. Goldberg, and M. Magdon-Ismail, "Efficient identification of overlapping communities," in *Intelligence and Security Informatics*, pp. 27–36, Springer, Berlin, Germany, 2005.

[7] X. Xu, J. Jäger, and H.-P. Kriegel, "A fast parallel clustering algorithm for large spatial databases," *Data Mining and Knowledge Discovery*, vol. 3, no. 3, pp. 263–290, 1999.

[8] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, vol. 1996, pp. 226–231, AAAI Press, 1996.

[9] Y. Zhang, J. Wang, Y. Wang, and L. Zhou, "Parallel community detection on large networks with propinquity dynamics," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 997–1006, ACM, July 2009.

[10] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: a petascale graph mining system implementation and observations," in *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09)*, pp. 229–238, IEEE, 2009.

[11] U. Kang, B. Meeder, and C. Faloutsos, "Spectral analysis for billion-scale graphs: discoveries and implementation," in *Advances in Knowledge Discovery and Data Mining: 15th Pacific-Asia Conference, PAKDD 2011, Shenzhen, China, May 24-27, 2011, Proceedings, Part II*, vol. 6635 of *Lecture Notes in Computer Science*, pp. 13–25, Springer, Berlin, Germany, 2011.

[12] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Parallel Processing and Applied Mathematics*, vol. 7203, pp. 286–296, Springer, Berlin, Germany, 2012.

[13] A. Prat-Pérez, D. Dominguez-Sal, and J.-L. Larriba-Pey, "High quality, scalable and parallel community detection for large real graphs," in *Proceedings of the 23rd International Conference on World Wide Web (WWW '14)*, pp. 225–236, World Wide Web Conferences Steering Committee, Seoul, Republic of Korea, April 2014.

[14] A. Prat-Pérez, D. Dominguez-Sal, J. M. Brunat, and J.-L. Larriba-Pey, "Shaping communities out of triangles," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM '12)*, pp. 1677–1681, ACM, November 2012.

[15] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E: Statistical, Nonlinear, and Soft Matter Physics*, vol. 76, no. 3, Article ID 036106, 2007.

[16] S. Gregory, "Finding overlapping communities in networks by label propagation," *New Journal of Physics*, vol. 12, no. 10, Article ID 103018, 2010.

[17] J. Xie and B. K. Szymanski, "Towards linear time overlapping community detection in social networks," in *Advances in Knowledge Discovery and Data Mining*, pp. 25–36, Springer, Berlin, Germany, 2012.

[18] C. L. Staudt and H. Meyerhenke, "Engineering high-Performance community detection heuristics for massive graphs," in *Proceedings of the 42nd Annual International Conference on Parallel Processing*, pp. 180–189, Lyon, France, October 2013.

[19] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, Article ID 026113, 2004.

[20] M. Chen, K. Kuzmin, and B. K. Szymanski, "Community detection via maximization of modularity and its variants," *IEEE Transactions on Computational Social Systems*, vol. 1, no. 1, pp. 46–65, 2014.

[21] K. Kuzmin, S. Y. Shah, and B. K. Szymanski, "Parallel overlapping community detection with SLPA," in *Proceedings of the International Conference on Social Computing (SocialCom '13)*, pp. 204–212, IEEE, September 2013.

[22] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, Beijing, China, 2012.

[23] J. Leskovec, *Amazon Product Co-Purchasing Network and Ground-Truth Communities*, 2014, http://snap.stanford.edu/data/com-Amazon.html.

[24] "DBLP collaboration network and groundtruth communities," 2014, http://snap.stanford.edu/data/com-DBLP.html.

[25] LiveJournal social network and groundtruth communities, 2014, http://snap.stanford.edu/data/com-LiveJournal.html.

[26] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: the state-of-the-art and comparative study," *ACM Computing Surveys*, vol. 45, no. 4, Article ID 2501657, 2013.

[27] A. Lancichinetti, S. Fortunato, and J. Kertész, "Detecting the overlapping and hierarchical community structure in complex networks," *New Journal of Physics*, vol. 11, no. 3, Article ID 033015, 2009.

[28] M. Chen, T. Nguyen, and B. K. Szymanski, "A new metric for quality of network community structure," *ASE Human Journal*, vol. 2, no. 4, pp. 226–240, 2013.

[29] S. Zhang, R.-S. Wang, and X.-S. Zhang, "Identification of overlapping community structure in complex networks using fuzzy c-means clustering," *Physica A: Statistical Mechanics and Its Applications*, vol. 374, no. 1, pp. 483–490, 2007.

[30] T. Nepusz, A. Petróczi, L. Négyessy, and F. Bazsó, "Fuzzy communities and the concept of bridgeness in complex networks," *Physical Review E: Statistical, Nonlinear, and Soft Matter Physics*, vol. 77, no. 1, Article ID 016107, 2008.

[31] H. Shen, X. Cheng, K. Cai, and M.-B. Hu, "Detect overlapping and hierarchical community structure in networks," *Physica A: Statistical Mechanics and its Applications*, vol. 388, no. 8, pp. 1706–1712, 2009.

[32] V. Nicosia, G. Mangioni, V. Carchiolo, and M. Malgeri, "Extending the definition of modularity to directed graphs with overlapping communities," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2009, no. 3, Article ID P03024, 2009.

[33] H.-W. Shen, X.-Q. Cheng, and J.-F. Guo, "Quantifying and identifying the overlapping community structure in networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2009, no. 7, Article ID P07042, 2009.

[34] D. Chen, M. Shang, Z. Lv, and Y. Fu, "Detecting overlapping communities of weighted networks via a local algorithm," *Physica A: Statistical Mechanics and its Applications*, vol. 389, no. 19, pp. 4177–4187, 2010.