PARALLEL OVERLAPPING COMMUNITY DETECTION WITH SLPA

By

Konstantin Kuzmin

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved by the
Examining Committee:

_____
Boleslaw K. Szymanski, Thesis Advisor

_____
Mark K. Goldberg, Member

_____
Sibel Adali, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2014
(For Graduation May 2014)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. 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 U.S. Government.

Portions of this thesis are based on earlier work which previously appeared as: (K. Kuzmin, S. Y. Shah, and B. K. Szymanski, "Parallel overlapping community detection with SLPA," in *Social Computing (SocialCom), 2013 International Conference on.* IEEE, 2013, pp. 204–212) and (K. Kuzmin, M. Chen, and B. K. Szymanski, "Parallelizing SLPA for Scalable Overlapping Community Detection," *IEEE Special Issue on Computational Aspects of Social Network Analysis*, 2014). However, the entire contents of this thesis is based solely on the work of its author. No part of this document includes any material written by any of the co-authors.

We would like to express our gratitude to the thesis advisor, Dr. Boleslaw K. Szymanski, whose knowledge, wisdom, and dedication made this research both intellectually challenging and enriching.

We are also happy to extend our special thanks to co-authors of the papers listed above: Syed Yousaf Shah and Mingming Chen.

Yousaf designed, developed, and tested another parallel SLPA implementation based on the Message Passing Interface (MPI). Although this work is not described here, it served as a valuable performance benchmark when developing our multithreaded version. Competition always leads to progress, and knowing speedup and efficiency of an alternative solution was important in calibrating our implementation. Numerous discussion that we had with Yousaf on different aspects of writing parallel code and implementing community detection algorithms were invaluable.

Mingming did a tremendous job validating communities detected by our multithreaded algorithm using a comprehensive set of both traditional methods and an innovative metric that they proposed. Despite the fact that presenting and

discussing these results is outside the scope of this thesis, it was crucial to verify that our parallel implementation produces meaningful results. Having the quality of communities measured numerically demonstrates that our parallel version of SLPA not only runs several times faster than a sequential implementation but it also outputs communities that reveal the actual structure of the network. It brings confidence that these communities can be successfully used in various applications, be it information networks, social science, or biology.

ABSTRACT

In a very general context, communities in networks are defined as groups of nodes that have some common properties such that connections are stronger between the nodes in a community than with the nodes in the rest of the network. It is quite common for nodes to participate in multiple communities. Therefore a community detection algorithm for such applications should be able to detect overlapping communities. However, overlapping community detection is more computationally intensive than disjoint community detection and presents new challenges that algorithm designers have to face. Besides, the big data phenomenon with exabytes of data brings up datasets that take too long to analyze using even the fastest algorithms currently available. Fortunately, the amount of computing power available to researches also increases. This computing power usually comes structured as a number of cores, processors, or machines joined together to form a high performance computer, cluster or a supercomputer. In this thesis we analyze what other researchers have done to utilize high performance computing to perform efficient community detection in social, biological, and other networks. We use the Speaker-listener Label Propagation Algorithm (SLPA) as the basis for our parallel overlapping community detection implementation. SLPA provides near linear time community detection and is well suited for parallelization. We explore the benefits of a multithreaded programming paradigm for both synthetic and real-world networks and show that it yields a significant performance gain over sequential execution in detecting overlapping communities.

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. Communities allow researches 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 entire community structure can be altered or destroyed by acting upon only a small fraction of the most influential nodes.

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 [2]. Nodes in such networks often correspond to genes, proteins, individuals, or species. Common examples of interactions are infectious contacts, regulatory interaction, or 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 commu-

Portions of this chapter previously submitted as: (K. Kuzmin, M. Chen, and B. K. Szymanski, "Parallelizing SLPA for Scalable Overlapping Community Detection," *IEEE Special Issue on Computational Aspects of Social Network Analysis*, 2014).

nity 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 decreases and can actually increase with the number of processors thus limiting the scalability of the entire solution.

Yet, there is another facet of scaling community detection solutions. As more and more hardware compute 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 in a form of increased processor count rather 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 doesn't degrade and hopefully even improves when run on larger and larger datasets.

In order to design efficient parallel community detection code, it is necessary to ensure high degree of concurrency. It means that multiple processors or cores should be able to process data in parallel with as little interaction and synchronization between them as possible.

A network consists of nodes, and there is a decision that needs to be made on which nodes are processed by which processors or cores. Clearly, this partitioning of nodes between the cores has a dramatic effect on how effectively the parallel code can run. If a process has a lot of dependencies to pieces of data processed by other cores, it will cause substantial synchronization overhead and severely limit the degree of parallelism. If, in contrast, nodes that are assigned to a process do not depend on any nodes processed by other cores, this process can run at full speed regardless of

how other cores process their data. Unfortunately, except for very special cases of networks analyzed with a particular number of processors, this scenario is rarely the case in real life.

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 their merits and drawbacks.

An input to a community detection algorithm is usually a single network file that needs to be analyzed. Effectively reading a network file and creating its in-memory representation can be challenging in certain scenarios since multiple processes access different regions of a single file simultaneously causing congestion in the I/O system. Besides, disk I/O is usually very slow in most systems compared to the speed of computation.

Our solution is built upon a shared memory architecture, and therefore an input file is read and processed only once. Since only one copy of the dataset data is resident in memory at any time, it completely avoids data replication any makes any data accessible by any processor. One of the key design features of our multithreaded approach is to minimize the amount of synchronization and achieve high degree of concurrency of code running on different processors and cores. Provided 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, there is no speedup saturation and it continues to improve up to maximal processor counts.

1.1 Motivation

In the modern world more and more data is being collected, processed, and stored every second. Video surveillance systems, sensor device networks, ubiquitous RFID tags, global positioning devices, smartphones, and other systems that became known as the Internet of things (IOT) [3] generate an enormous amount of data. This data can become a valuable source of information, if properly mined. Extending the concept of IOT to include also digital data that cannot be directly perceived by

humans leads to the *Internet of data (IOD)* [4]. IOD is likely to spawn even bugger collections of data than everyday objects.

With dataset sizes growing rapidly, it becomes increasingly important to be able to analyze this data within reasonable time limits. Different community detection methods and algorithms that have been developed by researchers to date vary greatly in their worst case running time complexity. Comprehensive reviews of the major community detection and clustering algorithms [5],[6] cover a range of complexities from $O(m^3n)$, $O(n^4/m^2)$, and $O(n^3)$ to $O(m)$, where n is the number of nodes and m is the number of edges in a network.

Although the asymptotic running time of all practical community detection algorithms is polynomial in the size of input, the actual running time on real datasets varies substantially depending both on the degrees of polynomial variables and constants hidden by the asymptotic notation. For instance, running time of InfoMap and FCD community detection algorithms on the same Epinion network (119,130 nodes and 704,276 edges) is compared in [7]. Although both algorithms were able to complete community detection in under three hours for this relatively modest graph, it is easy to imagine how long it would take to process a billion-edge network. Besides, the completion times achieved by these two algorithms differed by as much as a factor of two for certain network sizes. A comparison of FCD, InfoMap, WalkTrap, and GN algorithms conducted for synthetic data revealed that a ratio of slowest to fastest running time was peaking 50,000. A recent study [8] proposed a new community detection algorithm called CESNA and compared it performance on a synthetic network to the state-of-the-art community detection methods: COD-ICIL, MAC, BigCLAM, and DEMON. While some sequential algorithms finished processing 300,000 nodes in under two hours, others were running 3,000 nodes for more than three hours.

While major advances in design of community detection algorithms do not happen often and the best algorithms already offer linear time complexity ($O(m)$), a natural choice for improving performance and being able to analyze larger networks is to exploit the benefits of parallel platforms. A parallel implementation of CESNA [8] using 24 threads on a single machine was an order of magnitude

faster than its sequential counterpart. A Parallel Label Propagation (PLP) scheme described in [9] runs roughly 7 times faster with 32 threads than on a single thread.

Thus, in order to boost performance of community detection methods it seems quite natural to apply efforts to scale sequential algorithms for parallel environments. Given a fast and efficient near–linear time Label Propagation Algorithm we expect the solution to be very scalable and able to process networks with hundreds of millions of edges.

1.2 Thesis outline

The thesis is organized as follows. An overview of relevant research on parallel community detection is presented in Chapter 2. Chapter 3 introduces 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 as well as 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. Detailed performance analysis of our algorithm for one synthetic network and four real-life datasets is presented in Chapter 4. We also discuss the speedup and efficiency accomplished by our approach. Finally, in Chapter 5 we provide some closing remarks and discuss limitations of our implementation as well as the ways to overcome them.

2. LITERATURE REVIEW

Substantial effort has been put during the last decade into studying network clustering and analysis of social and other networks. Different approaches have been considered and a number of algorithms for community detection has been proposed. As online social communities continue to grow, and so do networks associated with them, 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 [10] 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 such that the objective function is minimized [11]. 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 doesn't aim to minimize the objective function for all possible partitions but produces a local optima solution instead, i.e. 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 [12] 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 as the number of processors and the network size increases. A parallel clustering algorithm is suggested in [13], which is a parallelized version of DBSCAN [14].

Portions of this chapter previously submitted as: (K. Kuzmin, M. Chen, and B. K. Szymanski, "Parallelizing SLPA for Scalable Overlapping Community Detection," *IEEE Special Issue on Computational Aspects of Social Network Analysis*, 2014).

A community detection approach based on propinquity dynamics is described in [15]. It doesn't use any explicit objective function but rather performs community detection based on heuristics rather than using an explicit objective function. 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 *emerging threshold*. Similarly, if the propinquity value of the endpoints of an existing edge is below a certain value, called *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 [15] 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 processors 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 [16], 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 $O(\frac{|V|+|E|}{P} log \frac{|V|+|E|}{P})$ where P is the number of processors in the cluster. Running PeGaSus on an M45 Hadoop supercomputer cluster shows that GIM-V scales up linearly as the number of machines is increased 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 [17] 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. Similarly to PeGaSus, HEigen scales up almost linearly

with the number of edges and processors and performs well up to billion edge scale networks. Its asymptotic running time is the same as that of PeGaSus' GIM-V.

In [18] 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 XMT/XMT2. Because of different hardware platforms two multithreaded programming environments had to be used: Cray XMT and OpenMP. These environments provide significantly different ways of managing the parallelism. While Cray XMT offers implicit, automatic concurrency, OpenMP requires a developer to explicitly manage parallelism.

For both architectures and programming environments 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. Additionally, the application was tested on a large uk-2007-05 graph with over a hundred million nodes and more than 3 billion edges. Unfortunately, no speedup or efficiency data for this dataset is presented in [18]. Therefore, it is not clear what scalability behavior this solution is capable of delivering for such networks. Based on the running time values given in the article, the processing rate can be calculated to be between approximately 5.58 million and 6.66 million edges/second on an Intel-based platform, and slightly above 1.3 million edges/second on a Cray XMT2 which is close to the values of processing rates seen for the other tested networks.

An improved version of the agglomerative parallel community detection solution is proposed in [19]. Three datasets are tested, uk-2007-05 being the largest, across five different hardware platforms (Cray XMT, Cray XMT2 and three different Intel-based systems). The algorithm demonstrates good scalability on both Cray and Intel platforms. The running time performance advantage relative to the initial implementation is especially significant for Cray XMT2.

3. PARALLEL LINEAR TIME COMMUNITY DETECTION

A family of label propagation community detection algorithms includes COPRA [20], LPA [21], and SLPA [22]. 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.

In this thesis we design 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 doesn't affect the computational complexity of the method. Since each edge is treated as undirected, an extra edge is added to the network for every edge of the network being read. Essentially, a network is made symmetrical, i.e. if there is an edge from some node i to some node j then there is also an edge from node j to

Portions of this chapter previously appeared as: (K. Kuzmin, S. Y. Shah, and B. K. Szymanski, "Parallel overlapping community detection with SLPA," in *Social Computing (SocialCom), 2013 International Conference on.* IEEE, 2013, pp. 204–212).

Portions of this chapter previously submitted as: (K. Kuzmin, M. Chen, and B. K. Szymanski, "Parallelizing SLPA for Scalable Overlapping Community Detection," *IEEE Special Issue on Computational Aspects of Social Network Analysis*, 2014).

node i. Every undirected edge is represented with two directed edges connecting two nodes in opposite directions. Although if the input network is initially undirected this can lead to doubling the number of edges that are represented internally in code, such approach is more general, and can be used for networks with directed edges as well. 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.

We test the performance of our solution on a synthetic graph and 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.

The SLPA [22] is a sequential linear time algorithm for detecting overlapping communities. SLPA iterates over 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 label list of length t, as every node receives one label in each iteration. After all iterations are completed, post processing is carried out on the list of labels and communities are extracted. We refer interested readers to full paper [22] 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 can be potentially a neighbor of some other nodes, therefore, if lists of labels of its neighbors are updated, it should receive a label randomly picked from the updated list of labels.

3.1 Multithreaded SLPA with Busy-waiting and Implicit Synchronization

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 be partitioned into subnetworks with one subnetwork to be assigned to each thread. Although partitioning can be done in several different ways depending on the objective that we are trying to reach, in this case the best partitioning will be the one that 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 partition 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 3.1. $T_1, T_2, ..., T_p$ 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 $n_{i1}, n_{i2}, ..., n_{ik}$ of size k where i is the thread number, and node neighbors are $m_1, m_2, ..., m_k$. 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 loose 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

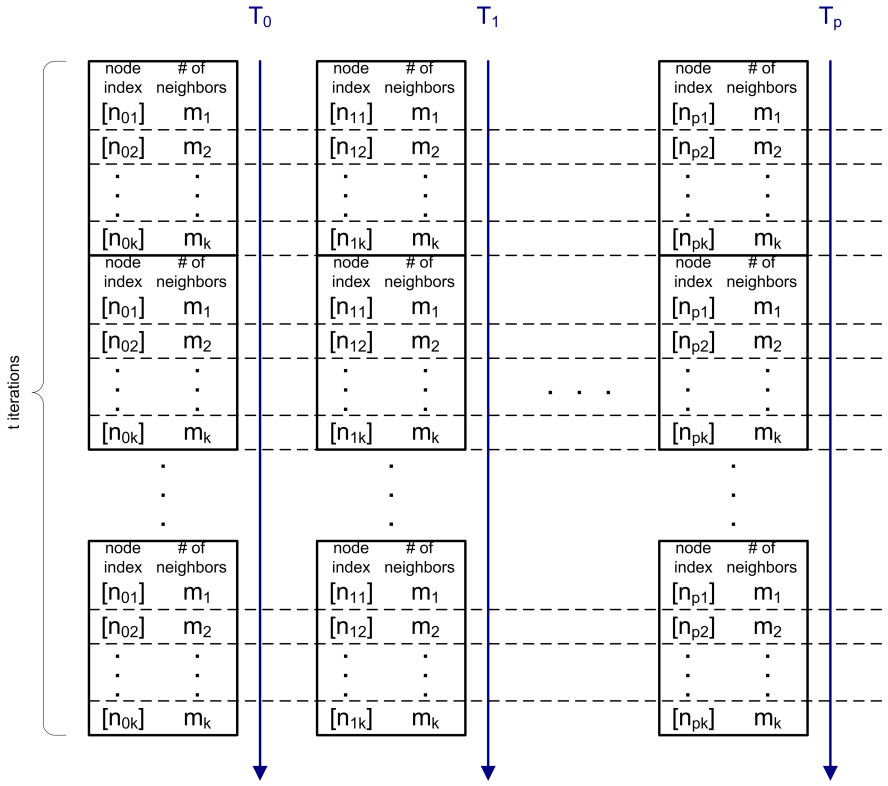

Figure 3.1: Ideal partitioning of the network for multithreaded SLPA.

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 3.2 illustrates such partitioning. As before, $T_1, T_2, ..., T_p$ are p threads that execute SLPA concurrently. As shown by the arrows, time flows from top to bottom. However each thread now has its subset of nodes $n_{i1}, n_{i2}, ..., n_{ik_i}$ 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.

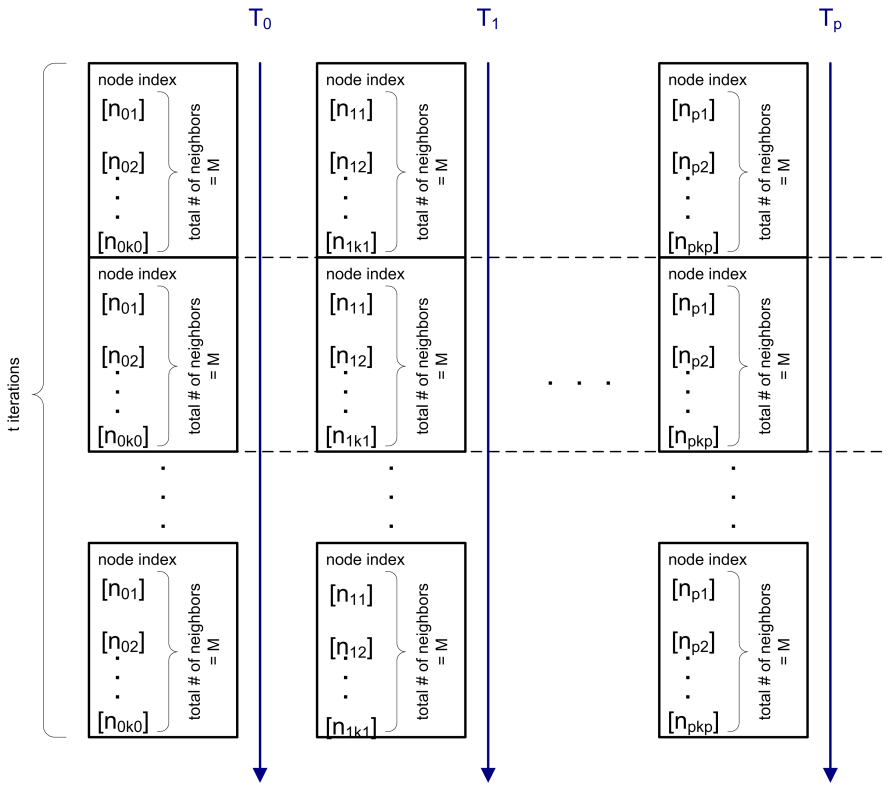

Figure 3.2: A better practical partitioning of the network for multithreaded SLPA.

We can employ yet an even simpler approach of just dividing 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. Moreover, 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 restriction on the way threads are synchronized. More specifically, if a particular thread is running faster than the others (whatever the reasons for this might be) 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. An element of array D contains thread number of the thread that processes some 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 $maxT$ 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 iteration counter is incremented, and the next iteration of the loop is considered.

3.2 Partition splitting

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. Because of 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

Algorithm 1 : Multithreaded SLPA

```
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 do
        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
```

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 (ones 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 earlier than it has in fact been completed by relying on the fact that the work which remains to be completed can not 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 Chapter 4 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 of all the nodes processed by the thread (few percent). 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.

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

Algorithm 2 : Multithreaded SLPA with splitting of nodes

```
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
```

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.

4. PERFORMANCE EVALUATION

4.1 Hardware platform

We performed runs on a hyper threaded Linux system operating on top of a Silicon Mechanics Rackform nServ A422.v3 machine (GANXIS.nest.rpi.edu). Processing power was provided by 64 cores organized as four AMD Opteron^{TM} 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 x 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).

4.2 Test datasets

One synthetic network and four real-world datasets have been used to test the performance of the multithreaded solution. The main properties of these datasets are given in Table 4.1. Three of these networks (com-Amazon, com-DBLP, and com-LiveJournal) have been acquired from Stanford Large Network Dataset Collection [23] which contains a selection of publicly available real-world networks (SNAP networks).

Undirected Amazon product co-purchasing network (also referred to as com-Amazon) was gathered, described, and analyzed in [24]. From the dataset information [25], 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

Portions of this chapter previously appeared as: (K. Kuzmin, S. Y. Shah, and B. K. Szymanski, "Parallel overlapping community detection with SLPA," in *Social Computing (SocialCom), 2013 International Conference on.* IEEE, 2013, pp. 204–212).

Portions of this chapter previously submitted as: (K. Kuzmin, M. Chen, and B. K. Szymanski, "Parallelizing SLPA for Scalable Overlapping Community Detection," *IEEE Special Issue on Computational Aspects of Social Network Analysis*, 2014).

Table 4.1: Properties of the datasets used in performance evaluation.

Dataset Name	Type	Nodes	Directed edges
Synthetic network	Undirected, unweighted, no ground-truth communities	4,350,248	12,332,112
com-Amazon original	Undirected, unweighted, with ground-truth communities	334,863	1,851,744
com-Amazon modified	Undirected, unweighted, with ground-truth communities	319,948	1,760,430
com-DBLP original	Undirected, unweighted, with ground-truth communities	317,080	2,099,732
com-DBLP modified	Undirected, unweighted, with ground-truth communities	260,998	1,900,118
com-LiveJournal	Undirected, unweighted, no ground-truth communities	3,997,962	34,681,189
Foursquare	Undirected, unweighted, no ground-truth communities	5,499,157	169,687,676

ground-truth community.

Since small ground-truth communities having less than 3 nodes have 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. Such a modification enables subsequent comparison of communities produced by the community detection algorithm and the ground truth communities. 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 Chapter 3, 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 [24]. According to the dataset information [26], 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 co-authorship 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 com-DBLP dataset was also modified to enable comparison with ground-truth communities as described above for the com-Amazon network. Since com-DBLP is also undirected, the same modification was applied to this dataset as to the com-Amazon network.

Another network from [24] 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 [27] describes com-LiveJournal as a free on-line blogging community where users declare friendship with each other. com-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. 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 real-world dataset that we used to test the performance of our

parallel solutions is a snapshot of the Foursquare network as of October 11, 2013. There is no information about ground truth communities available.

4.3 Testing methodology

We calculated speedup using formula shown in (4.1) and efficiency according to (4.2).

$$Speedup = \frac{T_1}{T_p} \tag{4.1}$$

where *Speedup* is the actual speedup calculated according to equation 4.1 and p is the number of processors or computing cores.

$$Efficiency = \frac{Speedup}{p} \tag{4.2}$$

Community detection runs for the synthetic network were performed using 100 iterations. Experiments with all real-world networks were run with 1,000 iterations (the value of $maxT$ was set to 1,000). 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 is 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 count) taking a few days, it was still feasible to complete all runs on all four real-world networks in under two weeks.

We conducted one set of measurements by considering only time for the label propagation phase since it is this stage that differs in our multithreaded implementation from the original sequential version. Time necessary to read an input file and construct in-memory representation of the nodes and edges as well any auxiliary data structures was not included in this timing. All post-processing 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, for real-world datasets 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 was considered. A further analysis of the sequential part of the algorithm and the limit on scalability that it imposes is given below in Section 4.5.

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 execute 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 for a number of threads larger than 32. This performance penalty is most likely caused by the memory banks organization of our machine. Speedup and efficiency are calculated using (4.1) and (4.2) 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 compiler version and compilation flags can play a crucial role not only in terms of how efficiently the code runs but in 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 compiler generate effective code capable of flawlessly executing multiple threads.

For instance, when compiler runs with either -O2 or -O3 optimization flag to compile the multithreaded SLPA 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 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 and that the more threads are used the smaller is a subset of nodes that each thread processes.

4.4 Analysis of results

The results of performance runs of our multithreaded parallel implementation are presented in Figures 4.1–4.22 below. (Data export was performed using Daniel's XL Toolbox add-in for Excel, version 6.51, developed by Daniel Kraus, Würzburg, Germany.)

Figure 4.1 shows the time it took to complete the label propagation phase of the multithreaded SLPA on a synthetic network for the number of cores varying from 1 to 32. Figures 4.3, 4.5, 4.7, and 4.9 show the time it took to complete the label propagation phase of the multithreaded parallel SLPA on four datasets (com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, respectively) for the

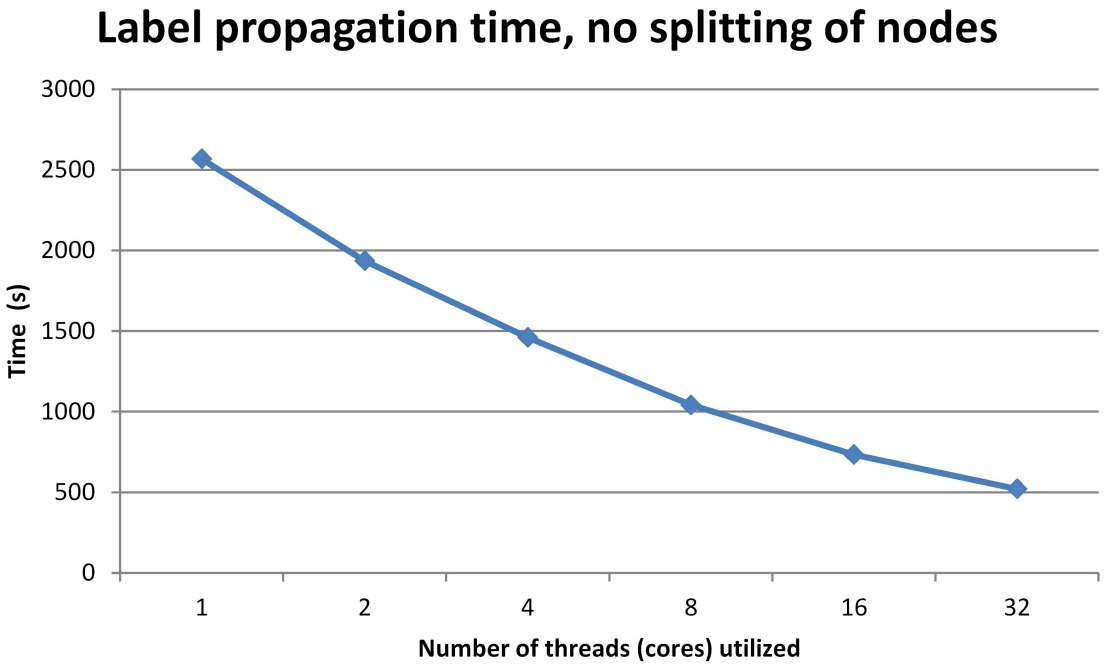

Figure 4.1: Label propagation time for a synthetic network with no splitting of nodes.

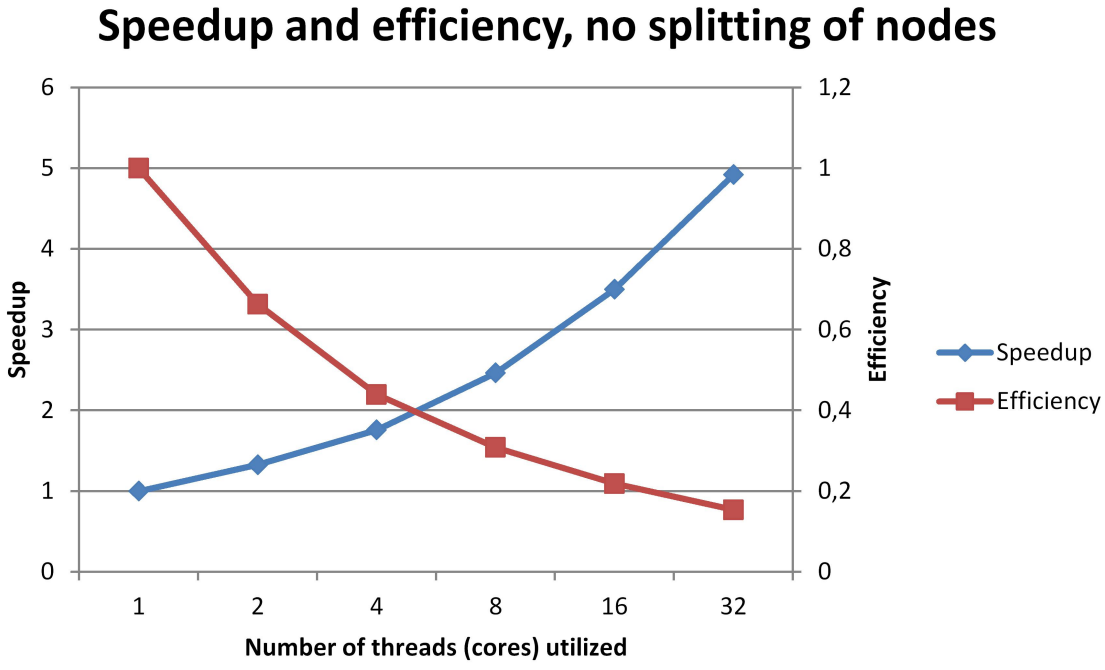

Figure 4.2: Speedup and efficiency for a synthetic network with no splitting of nodes.

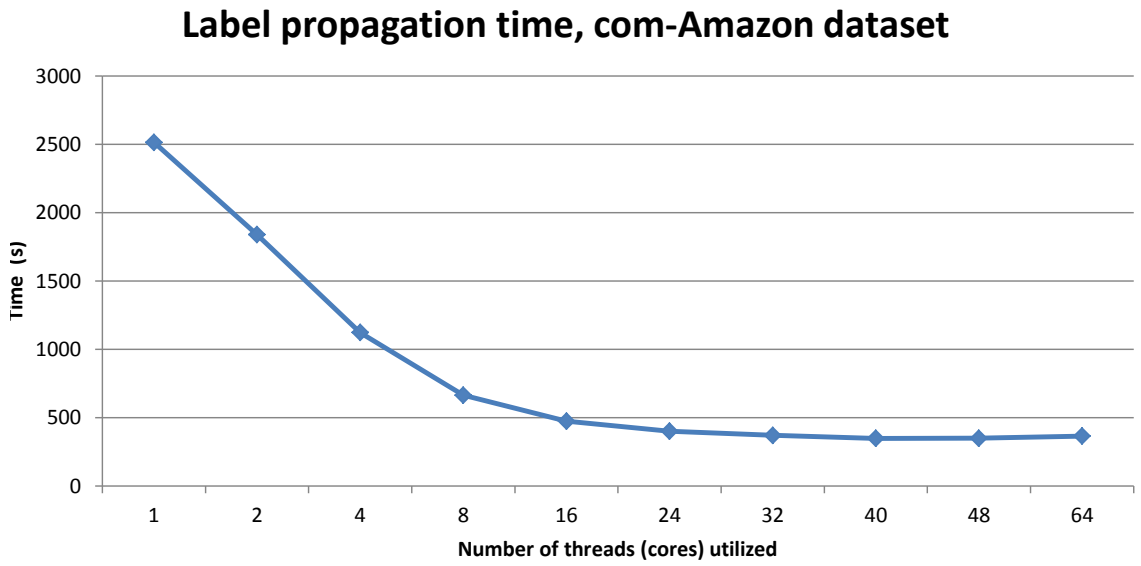

Figure 4.3: Label propagation time for com-Amazon network at different number of cores.

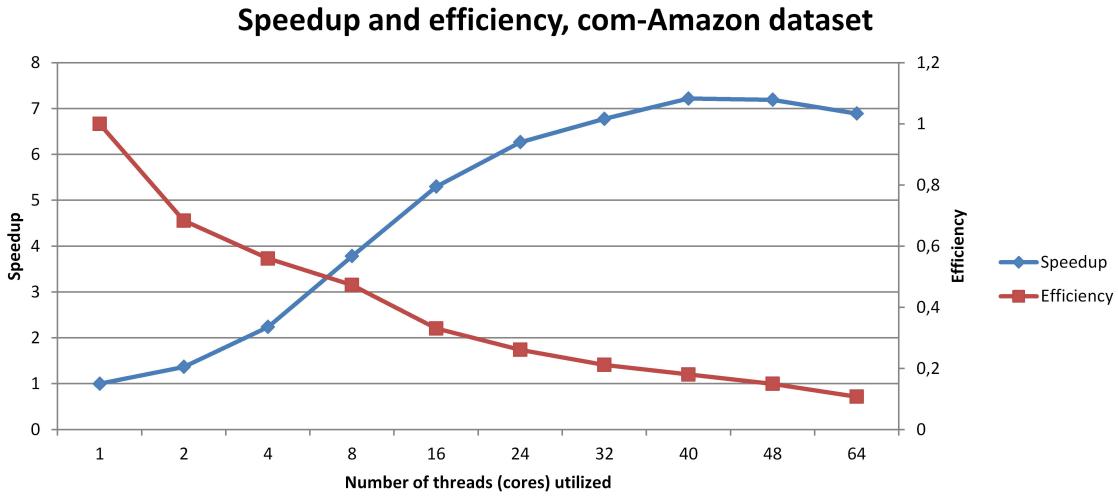

Figure 4.4: Speedup and efficiency for com-Amazon network (considering only label propagation time) at different number of cores.

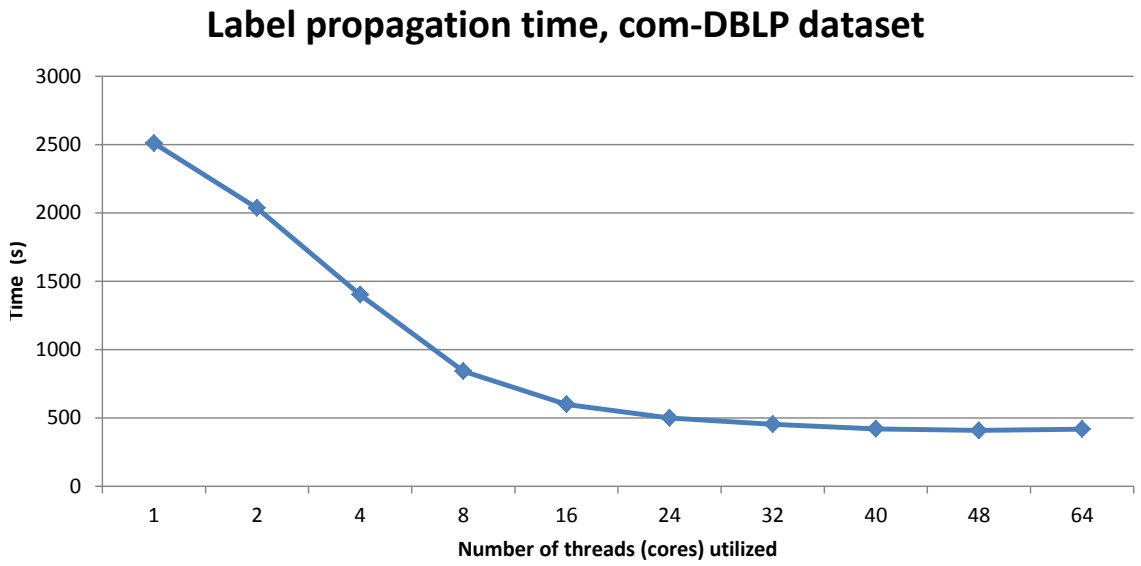

Figure 4.5: Label propagation time for com-DBLP network at different number of cores.

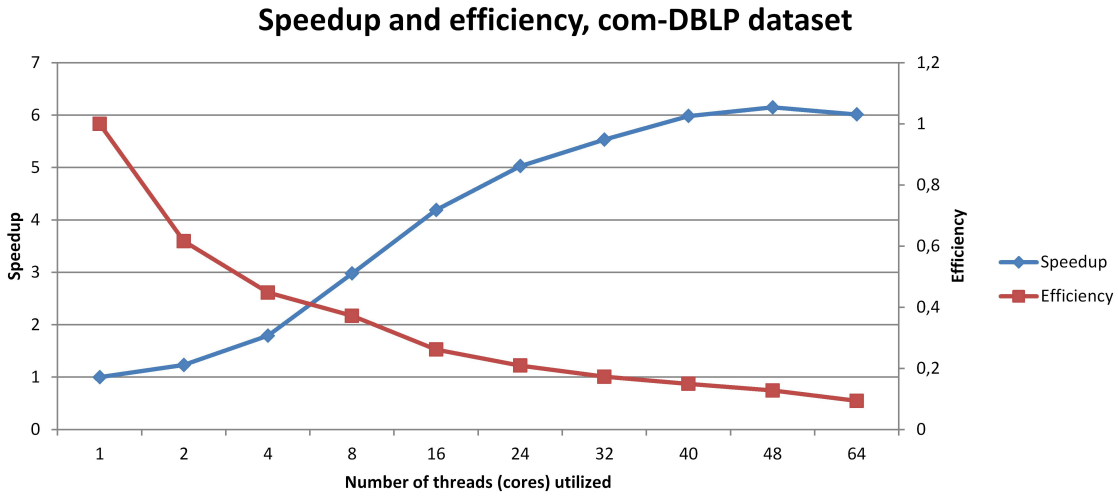

Figure 4.6: Speedup and efficiency for com-DBLP network (considering only label propagation time) at different number of cores.

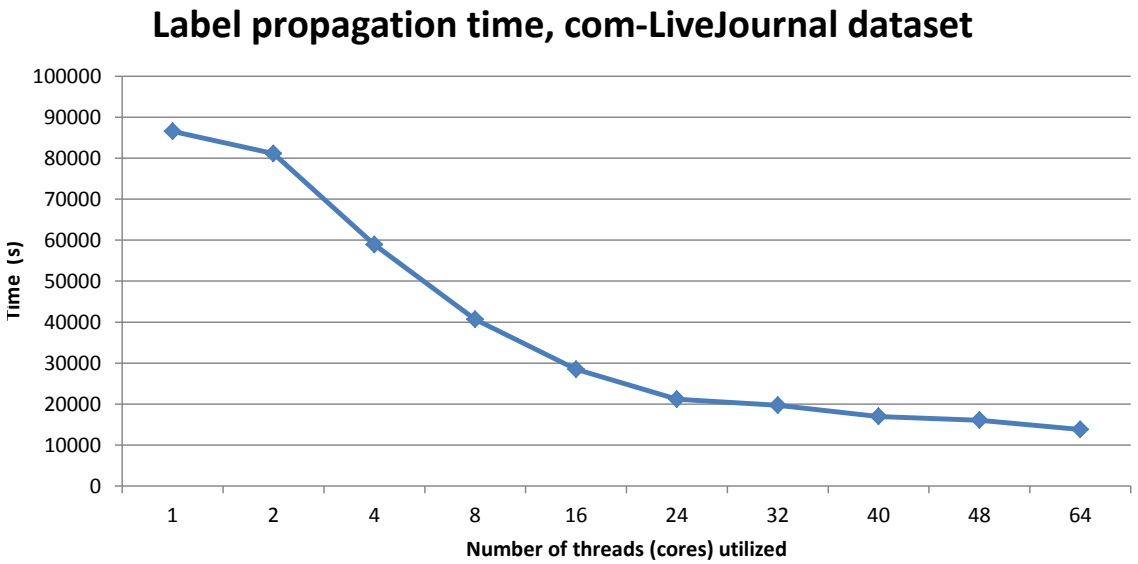

Figure 4.7: Label propagation time for com-LiveJournal network at different number of cores.

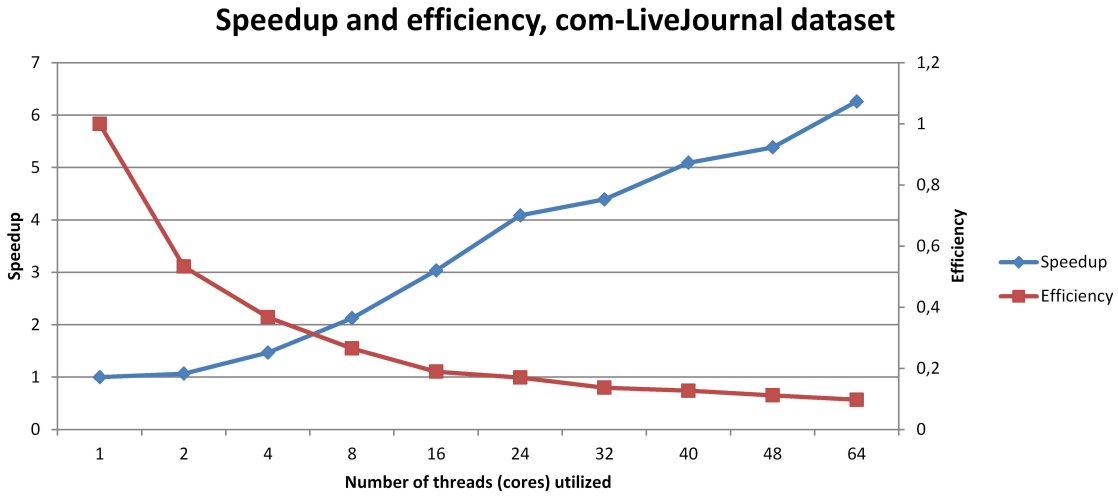

Figure 4.8: Speedup and efficiency for com-LiveJournal network (considering only label propagation time) at different number of cores.

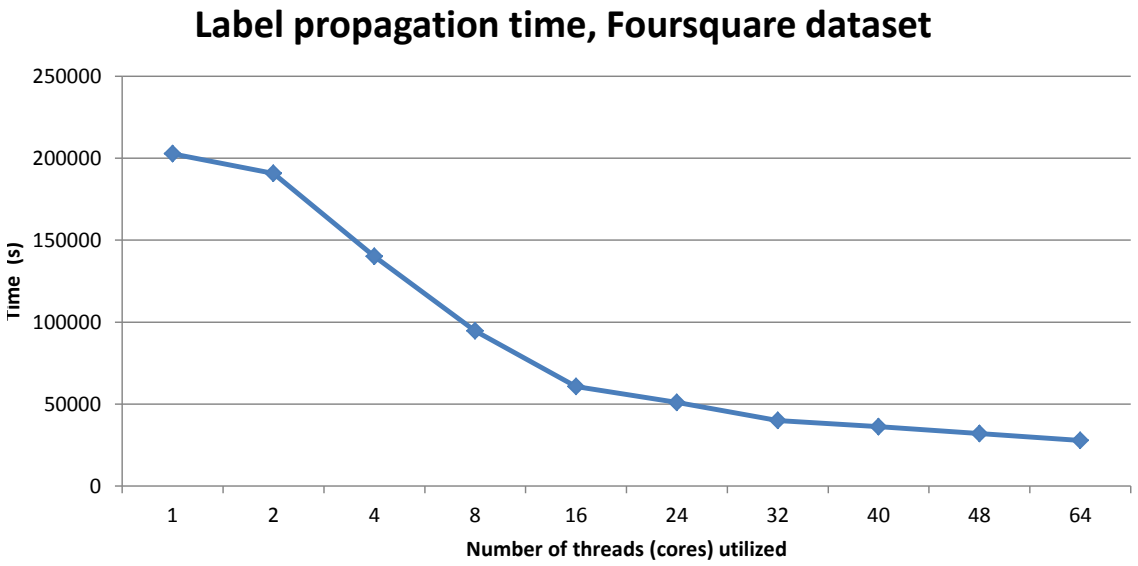

Figure 4.9: Label propagation time for Foursquare network at different number of cores.

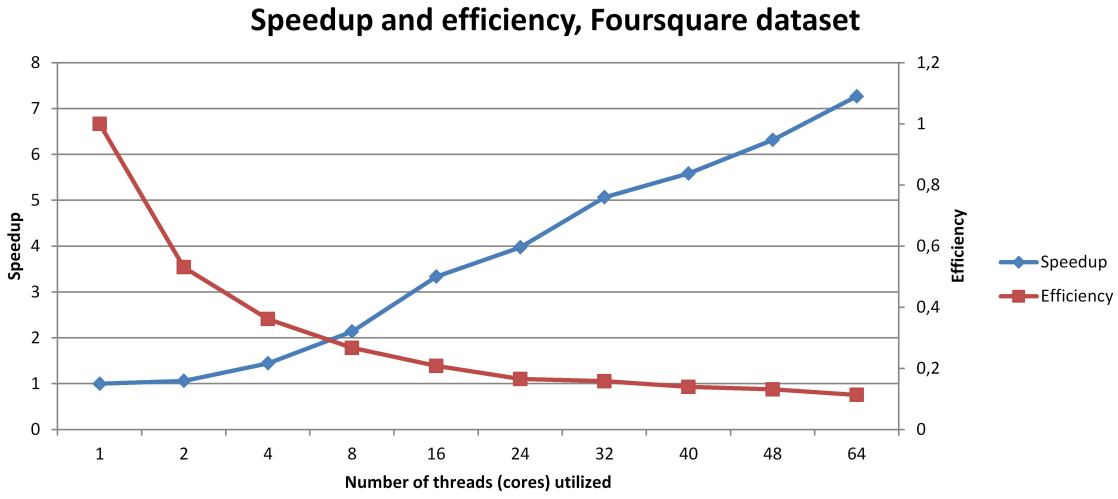

Figure 4.10: Speedup and efficiency for Foursquare network (considering only label propagation time) at different number of cores.

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 cores and more, 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 Chapter 1, this is clearly something to be expected as 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 4.2, 4.4, 4.6, 4.8, and 4.10 which plot the values of speedup and efficiency for the five datasets (synthetic, com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, respectively) and the number of cores from 1 to 64 (1 to 32 for synthetic). As the number of cores increase, 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 grow while the network (and hence the number of nodes and edges) stays the same, each thread gets fewer nodes and edges to process. In limit, it can cause the overhead of creating and running threads to outweigh the benefits of parallel execution for a sufficiently small number of nodes. Furthermore, as the number of cores grows, the number of neighbors of nodes with external dependencies increases (both because each thread gets fewer nodes and more threads to execute them). More nodes with external dependencies, in turn, means that threads are more dependent on other threads.

Figure 4.11 shows synthetic network label propagation times of the multi-threaded version of SLPA which splits nodes into a subset of nodes that have neigh-

bors processed by other threads, and a subset of nodes that do not have such neighbors. The splitting ratio is fixed in this case at 0.2. Figure 4.12 provides an insight on the speedup and efficiency for this configuration. The general shape of curves is similar to those in the version with no splitting. However, it can be seen that the absolute values are better (times are lower, speedup and efficiency are larger) for the version with splitting of nodes. The data collected supports our expectations that minimizing the waiting time that a thread spends in a busy-waiting loop while other threads are catching up makes code run faster and more efficiently.

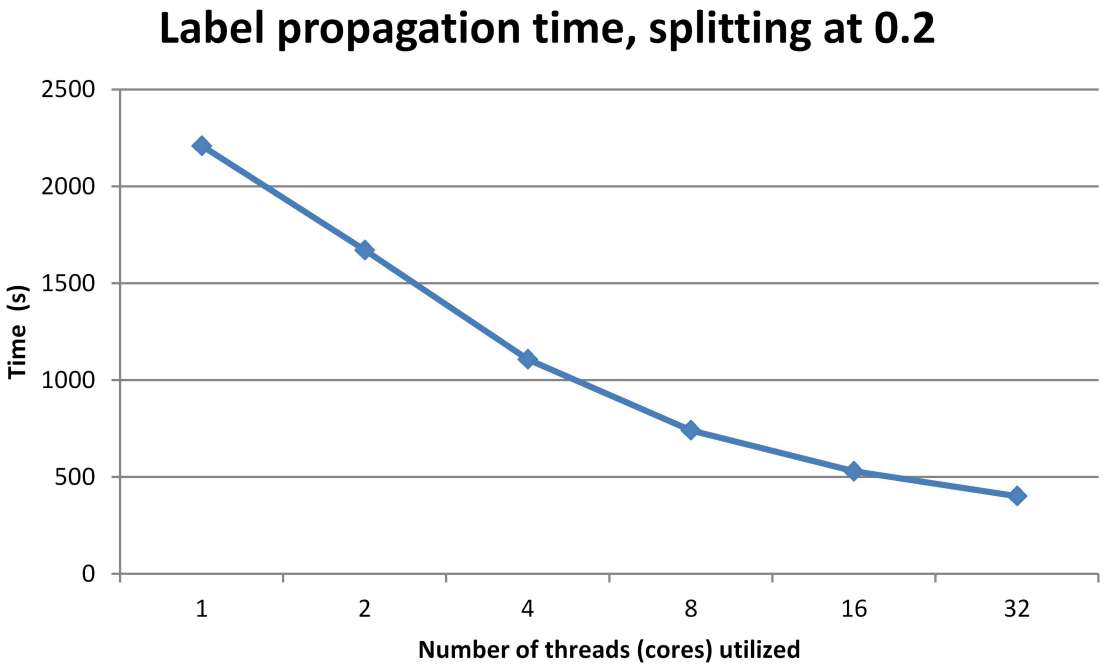

Figure 4.11: Label propagation time for a synthetic network with splitting of nodes.

The benefit of splitting the nodes can further be examined by looking at Figures 4.13 and 4.14. For the entire range from 1 to 32 cores the version with splitting outperformed its no-splitting counterpart in terms of label propagation time. The advantage is low for 2 cores and also declines as the number of cores increase. Both speedup and efficiency of the version with splitting were worse with 2 cores than for a version with no splitting, but it was better for all the other cases. Speedup and efficiency also degrade when the number of cores is increased to more than 8. These effects are related to the behavior of split subsets of nodes when the number

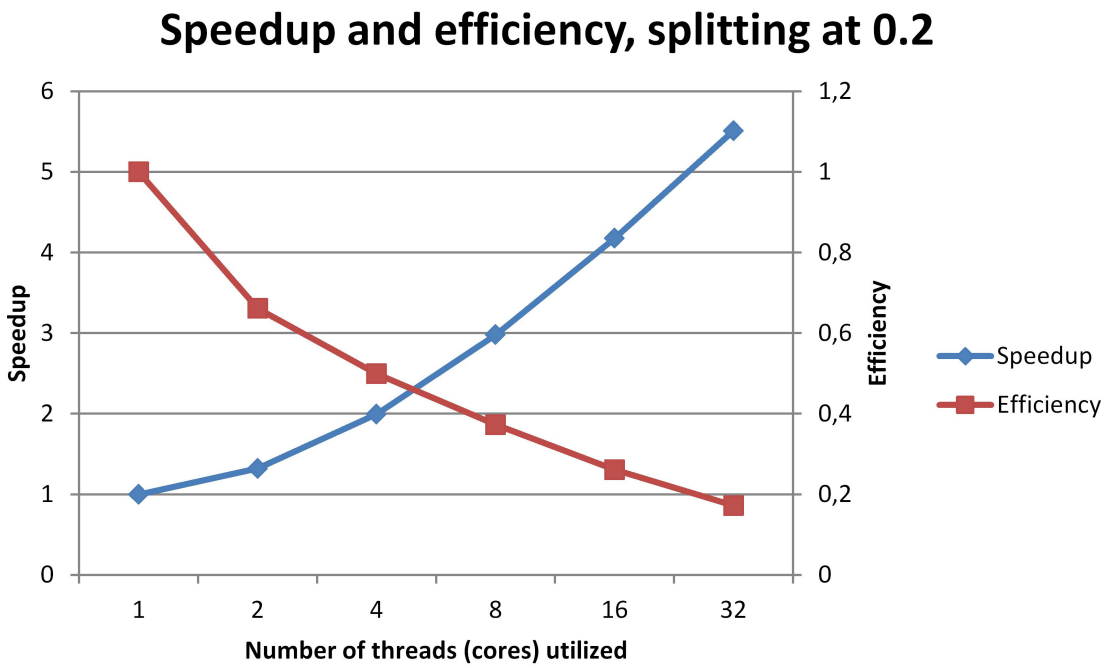

Figure 4.12: Speedup and efficiency for a synthetic network with splitting of nodes.

of cores increases, as discussed above.

However, for the sake of fair data interpretation it should be reminded that the definition of efficiency which we are using here is based on Equation 4.2. 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.

Although the hardware platform that was used in our experiments (Silicon Mechanics Rackform nServ A422.v3) has been discontinued and pricing information is no longer available, the next version of the same system, Rackform nServ A422.v4, is offered. Using an online price calculator [28] maintained by the vendor it is possible to compare the cost parameters of different server configurations.

The pricing data presented in Table 4.2 is given for different models of CPU and varying number of processors. The base system is the same for all configurations,

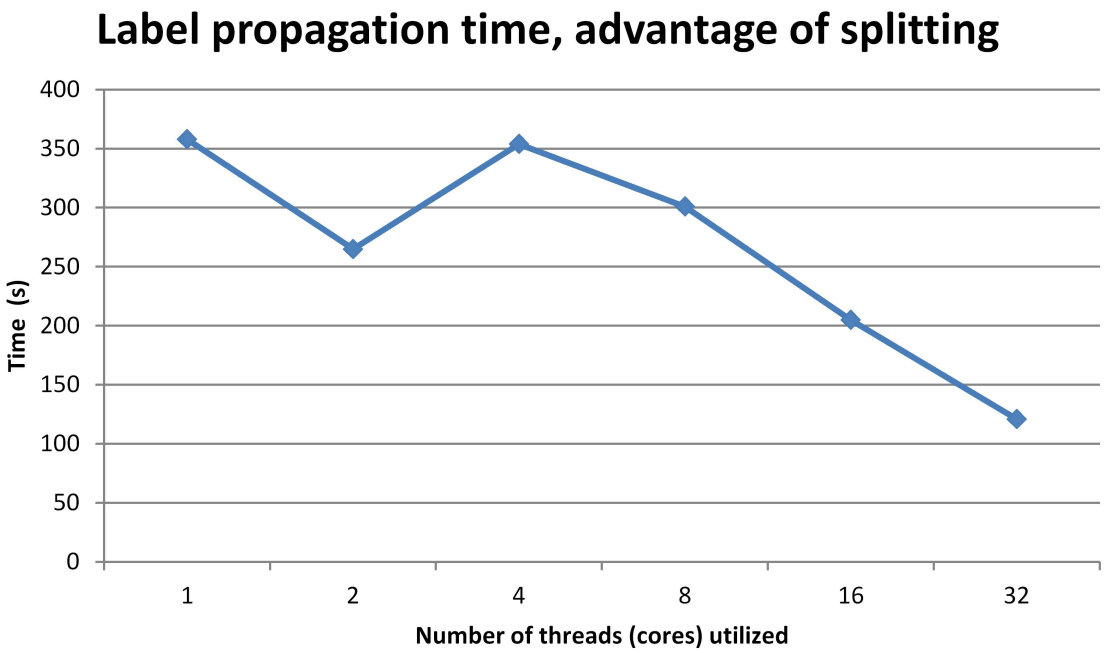

Figure 4.13: Label propagation time advantage (as a difference between running time for a version without and with splitting).

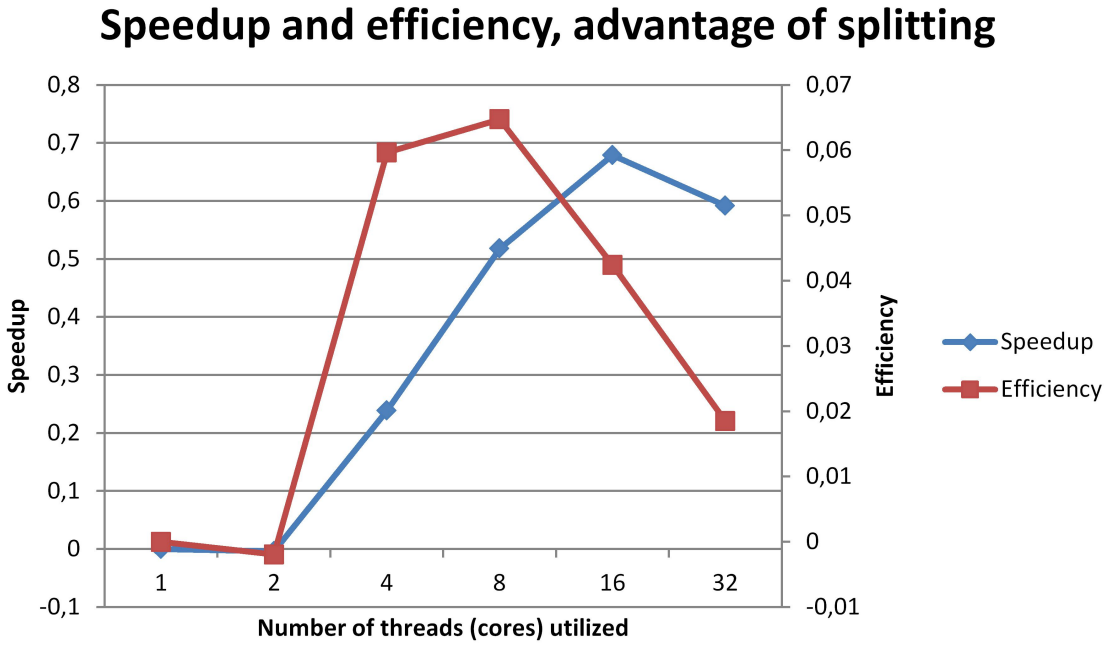

Figure 4.14: Speedup and efficiency advantage (as a difference between speedup and efficiency for a version without and with splitting).

and it is similar to the server that was used in our experiments (512 GB bank of Random Access Memory (RAM) (32 x 16 GB DDR3-1600 ECC Registered 2R DIMMs) running at 1333 MT/s Max; all other options are standard for the base configuration, and no additional items are selected). Clearly, there is an order of magnitude of difference in the cost per core between the lowest and the highest performance configurations. Amazingly, comparing these two extremes we can find that 16 times more cores costs only 1.5 times more money.

Consequently, the pure parallel efficiency defined by Equation 4.2 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 Equation 4.2. For example, it takes close to 41 hours to completely perform community detection using 1,000 iterations on the Foursquare network with 4 threads (at 4 cores), and just under 10 hours at 64 cores. If we assume that the same running times can be obtained on two corresponding extreme Rackform nServ A422.v4 configurations from Table 4.2 (with one Opteron 6308 and four Opteron 6380, respectively), then the cost efficiency of using a 64–core configuration vs. a 4-core system is going to be around 2.93, i.e. "superlinear" (a 1.5 times increase in cost brings a 4.38 times improvement of the running time).

Figures 4.15, 4.17, 4.19, and 4.21 present the total community detection time of the multithreaded parallel SLPA on four datasets (com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, respectively) 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 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 4.16, 4.18, 4.20,

Table 4.2: The cost of different configurations of a Silicon Mechanics Rackform nServ A422.v4 server system.

CPU	Number of CPUs	Total number of cores	Price per CPU, $	Price per core, $	System price, $
Opteron 6308 (3.5GHz, 4-Core, G34, 16MB L3 Cache)	1	4	9047.00	2261.75	9047.00
Opteron 6308 (3.5GHz, 4-Core, G34, 16MB L3 Cache)	2	8	4815.00	1203.75	9630.00
Opteron 6308 (3.5GHz, 4-Core, G34, 16MB L3 Cache)	4	16	2699.00	674.75	10796.00
Opteron 6328 (3.2GHz, 8-Core, G34, 16MB L3 Cache)	1	8	9135.00	1141.88	9135.00
Opteron 6328 (3.2GHz, 8-Core, G34, 16MB L3 Cache)	2	16	4903.00	612.88	9806.00
Opteron 6328 (3.2GHz, 8-Core, G34, 16MB L3 Cache)	4	32	2787.00	348.38	11148.00
Opteron 6348 (2.8GHz, 12-Core, G34, 16MB L3 Cache)	1	12	9135.00	761.25	9135.00
Opteron 6348 (2.8GHz, 12-Core, G34, 16MB L3 Cache)	2	24	4903.00	408.58	9806.00
Opteron 6348 (2.8GHz, 12-Core, G34, 16MB L3 Cache)	4	48	2787.00	232.25	11148.00
Opteron 6380 (2.5GHz, 16-Core, G34, 16MB L3 Cache)	1	16	9730.00	608.13	9730.00
Opteron 6380 (2.5GHz, 16-Core, G34, 16MB L3 Cache)	2	32	5498.00	343.63	10996.00
Opteron 6380 (2.5GHz, 16-Core, G34, 16MB L3 Cache)	4	64	3382.00	211.38	13528.00

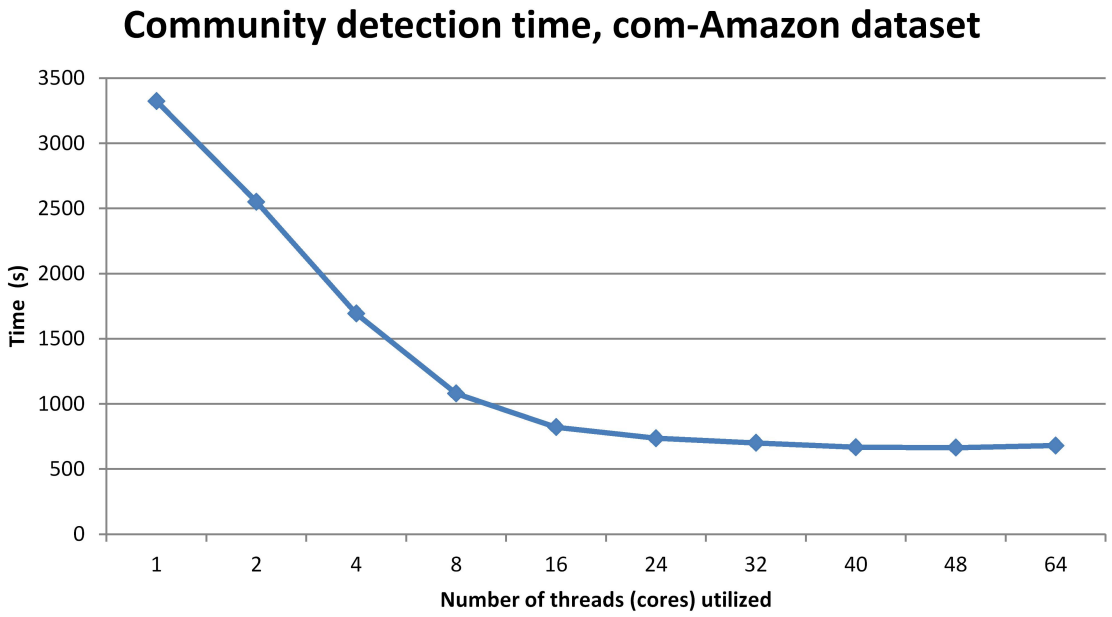

Figure 4.15: Total execution time for com-Amazon network at different number of cores.

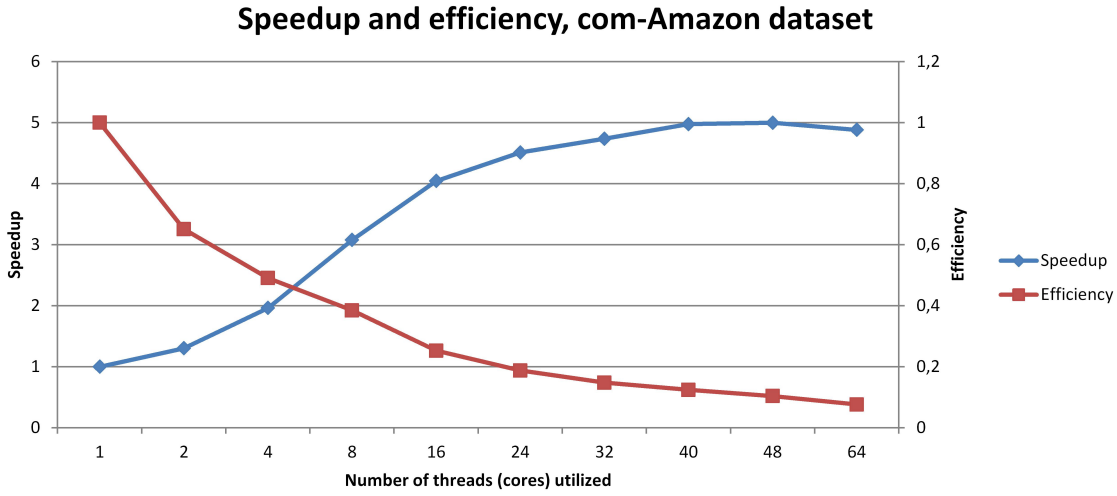

Figure 4.16: Speedup and efficiency for com-Amazon network (considering total execution time) at different number of cores.

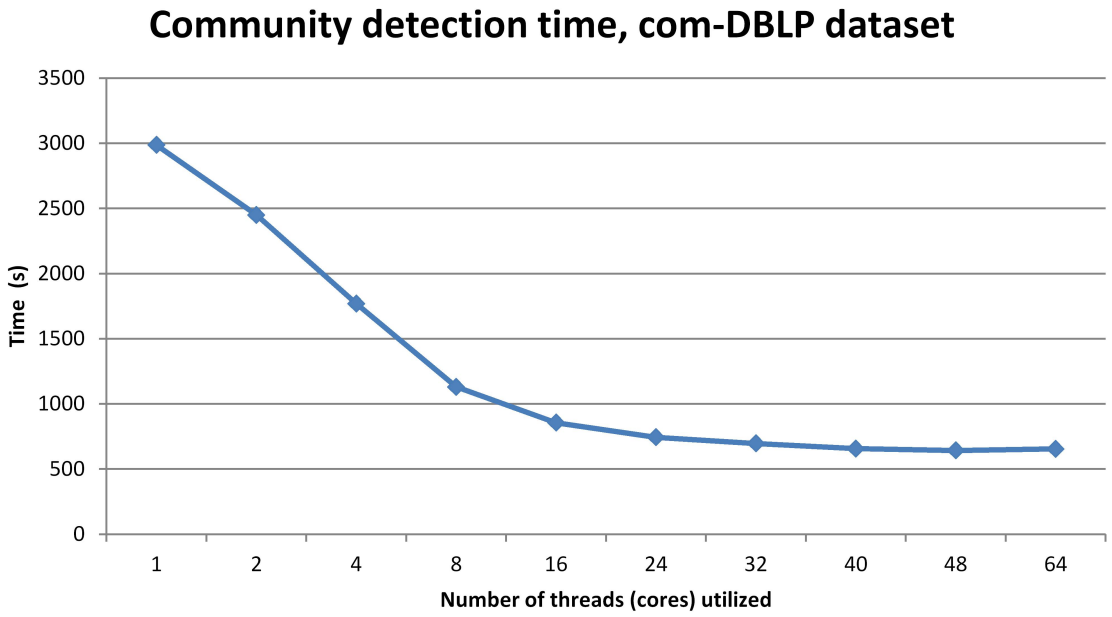

Figure 4.17: Total execution time for com-DBLP network at different number of cores.

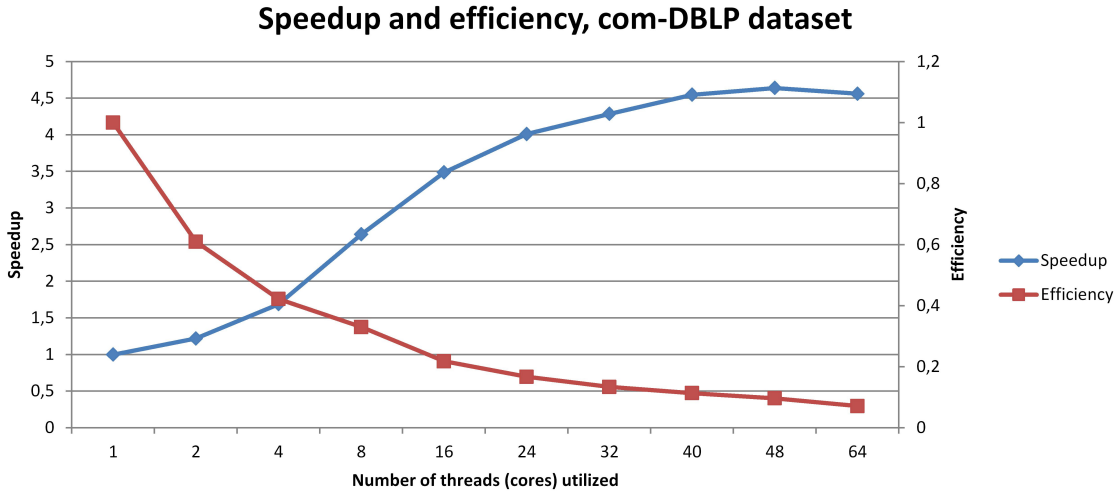

Figure 4.18: Speedup and efficiency for com-DBLP network (considering total execution time) at different number of cores.

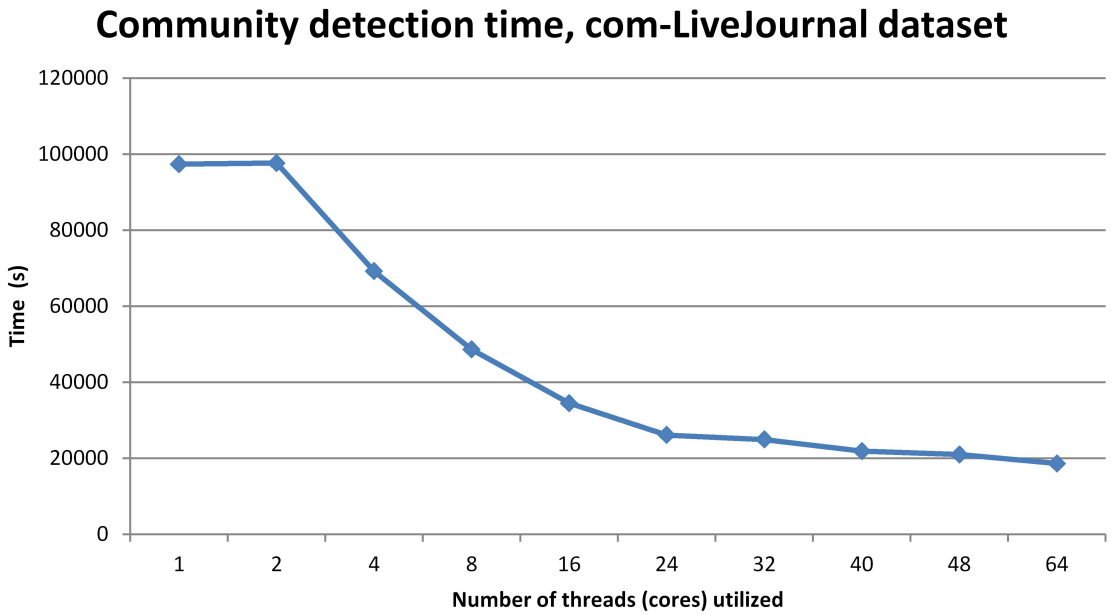

Figure 4.19: Total execution time for com-LiveJournal network at different number of cores.

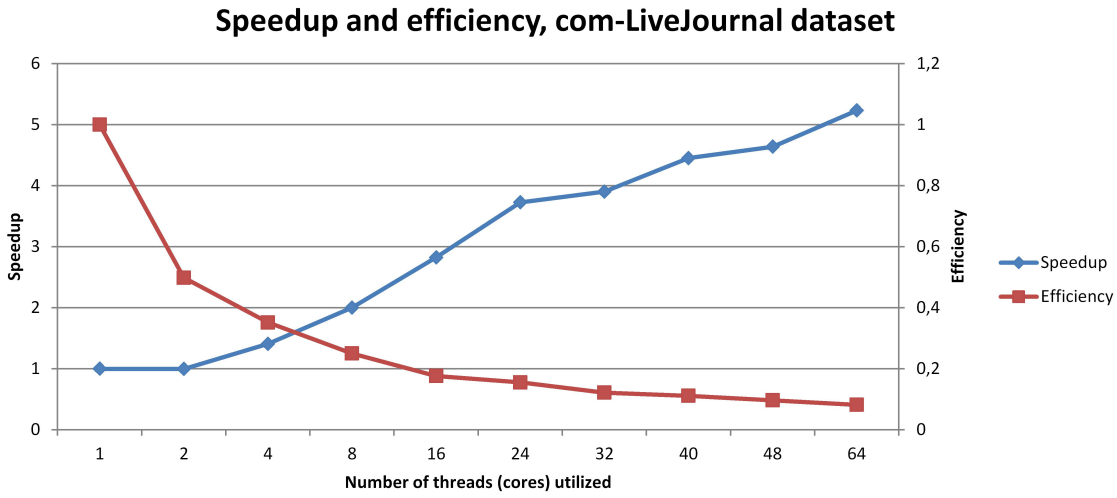

Figure 4.20: Speedup and efficiency for com-LiveJournal network (considering total execution time) at different number of cores.

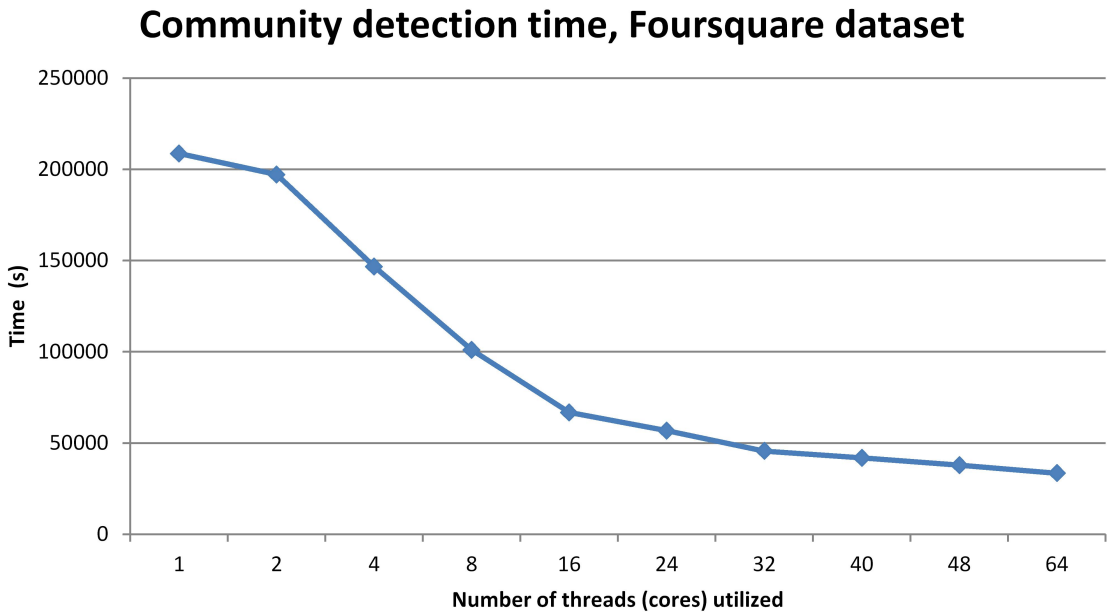

Figure 4.21: Total execution time for Foursquare network at different number of cores.

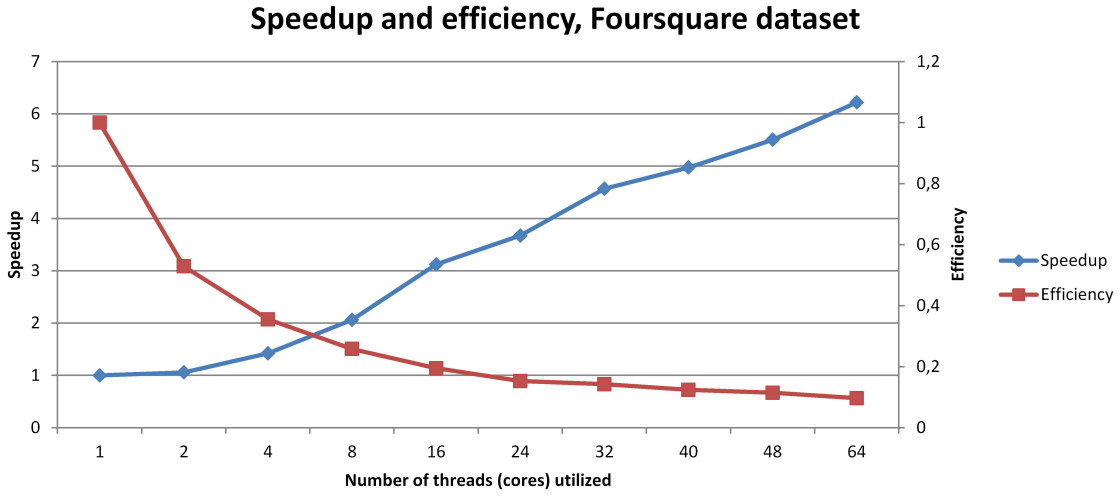

Figure 4.22: Speedup and efficiency for Foursquare network (considering total execution time) at different number of cores.

and 4.22 for the four datasets (com-Amazon, com-DBLP, com-LiveJournal, and Foursquare, respectively) 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.

4.5 Scalability limit

Designers of parallel applications need to be aware of the inherent limit on the maximum overall speedup that can be possibly achieved for a certain application on a multiprocessor system. A dependency between the fraction of the computational load that appears to be sequential and the performance of the entire parallel system was first outlined by Gene M. Amdahl in [29]. He noted that even if this sequential portion was run on a separate processor, an upper limit on the overall system throughput would render all efforts on increasing performance of the parallel portion useless. This observation was later restated in a form the became known as Amdahl's Law [30]. It can be expressed by the following formula:

$$Speedup = \frac{1}{f_s + \frac{1-f_s}{p}} \tag{4.3}$$

where $Speedup$ is the overall speedup of running on p processors a program that has a sequential fraction of f_s. From Equation 4.3 it follows that if f_s is large, then even if the parallel portion is infinitely parallelizable, the $\frac{1-f_s}{p}$ term will tend to zero in the limit which will make the $Speedup$ close to 1 and independent of the value of p.

It should be remembered, however, that Amdahl's Law is based on the assumption that a sequential fraction f_s is fully sequential, and the parallel fraction $1 - f_s$ is infinitely parallelizable. This is rarely the case in real life. Besides, it only considers that additional processors can be used to speed up the parallel portion while the sequential fraction does not benefit from increasing the computing power.

With today's silicon technologies, more sophisticated multicore chip designs can be considered [31]: symmetric, asymmetric, and dynamic. However, in order to use these multicore models they need to be implemented in silicon by chip designers.

In practice it is usually infeasible to analytically compute the value of f_s for an actual piece of code. Meanwhile, knowing this value could be useful in deciding how much effort should be put into enhancing performance of the parallel portion. Based on the formula 4.3 for Amdahl's Law, we can roughly approximate the value of f_s using the speedup data that we collected.

Figure 4.23 shows values of the sequential component calculated based on the Equation 4.3 from the running time data of four real-world networks. Naturally, this calculation has to use the total running time, not just the label propagation time. Remarkably, it seems that using running times obtained for different datasets as the number of cores approaches 64 converges to the value of the calculated sequential fraction of around 0.2. It means that by Amdahl's Law the value of speedup which we expect should be about 5. More importantly, even if the parallel fraction of the code tends to zero as we increase the number of cores, the overall speedup is going to be limited by the values of 5 to 8 or so.

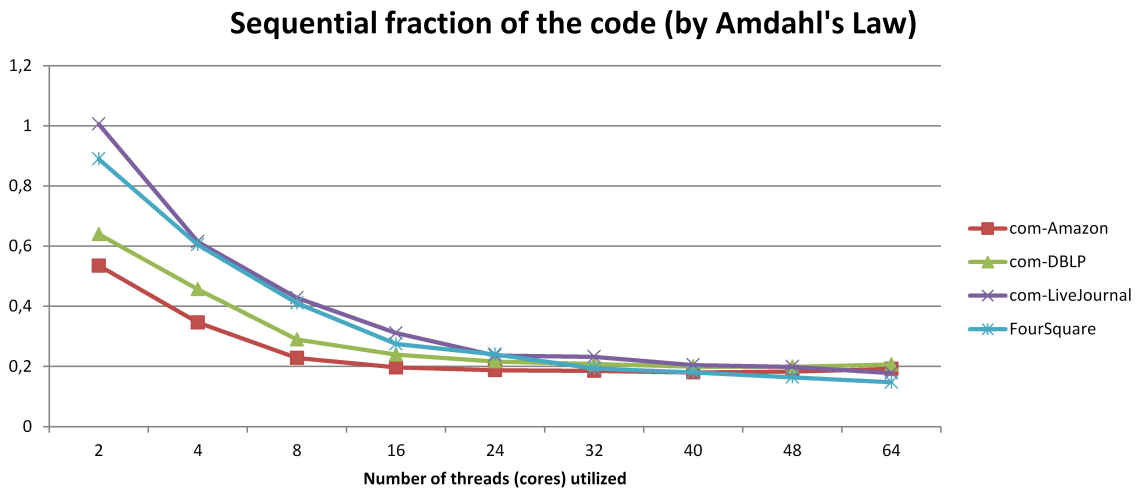

Figure 4.23: Calculated sequential fraction of the code (by Amdahl's Law).

A 0.2 fraction of sequential code is quite high, and this is what limits the

scalability of our solution. It is important to realize that a fraction of sequential code is not solely a property of the code itself. It also reflects data dependencies between nodes of a network since a larger number of such dependencies introduces more synchronization between the threads which in turn causes them to execute less concurrently. Hence, the calculated fraction of sequential code will depend on the properties of the dataset and the partitioning of nodes between the threads. Better partitioning techniques and reducing synchronization can lead to lower values of f_s. Besides, parallelizing operations that are currently performed sequentially (e.g. reading and writing files) is another way to decrease the fraction of sequential code and improve speedup and efficiency.

5. CONCLUSION

In this thesis, 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. Moreover, considering the cost per processing core it might be economically effective to obtain a system with the maximum number of cores. Our multithreaded SLPA implementation was proven to be scalable both in terms of using more cores and analyzing increasingly large networks. 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.

Portions of this chapter previously submitted as: (K. Kuzmin, M. Chen, and B. K. Szymanski, "Parallelizing SLPA for Scalable Overlapping Community Detection," *IEEE Special Issue on Computational Aspects of Social Network Analysis*, 2014).

REFERENCES

[1] R. E. Park, *Human communities: The city and human ecology.* Free Press, 1952, vol. 2.

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

[3] K. Ashton, "That 'Internet of things' thing," *RFiD Journal*, vol. 22, pp. 97–114, 2009.

[4] W. Fan, Z. Chen, Z. Xiong, and H. Chen, "The Internet of data: a new idea to extend the IOT in the digital world." *Frontiers of Computer Science*, vol. 6, no. 6, pp. 660–667, 2012.

[5] J. Xie, S. Kelley, and B. K. Szymanski, "Overlapping community detection in networks: The state-of-the-art and comparative study," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 43, 2013.

[6] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.

[7] Y. Song and S. Bressan, "Fast community detection," in *Database and Expert Systems Applications.* Springer, 2013, pp. 404–418.

[8] J. Yang, J. McAuley, and J. Leskovec, "Community detection in networks with node attributes," *arXiv preprint arXiv:1401.7267*, 2014.

[9] C. L. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," in *Parallel Processing (ICPP), 2013 42nd International Conference on.* IEEE, 2013, pp. 180–189.

[10] 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.

[11] J. Hartigan and M. 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.

[12] J. Baumes, M. Goldberg, and M. Magdon-Ismail, "Efficient identification of overlapping communities," *Intelligence and Security Informatics*, pp. 1–5, 2005.

[13] X. Xu, J. Jäger, and H. Kriegel, "A fast parallel clustering algorithm for large spatial databases," *High Performance Data Mining*, pp. 263–290, 2002.

[14] 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. AAAI Press, 1996, pp. 226–231.

[15] 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*. ACM, 2009, pp. 997–1006.

[16] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 229–238.

[17] U. Kang, B. Meeder, and C. Faloutsos, "Spectral analysis for billion-scale graphs: Discoveries and implementation," in *Advances in Knowledge Discovery and Data Mining*. Springer, 2011, pp. 13–25.

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

[19] J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable multi-threaded community detection in social networks," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012, pp. 1619–1628.

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

[21] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.

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

[23] J. Leskovec. Stanford large network dataset collection. [Online]. Available: http://snap.stanford.edu/data (Date Last Accessed March 3, 2014)

[24] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012.

[25] J. Leskovec. Amazon product co-purchasing network and ground-truth communities. [Online]. Available: http://snap.stanford.edu/data/com-Amazon.html (Date Last Accessed March 3, 2014)

[26] ——. DBLP collaboration network and ground-truth communities. [Online]. Available: http://snap.stanford.edu/data/com-DBLP.html (Date Last Accessed March 3, 2014)

[27] ——. LiveJournal social network and ground-truth communities. [Online]. Available: http://snap.stanford.edu/data/com-LiveJournal.html (Date Last Accessed March 3, 2014)

[28] 2U Quad Opteron Server - Rackform nServ A422.v4 - Silicon Mechanics. [Online]. Available: http://www.siliconmechanics.com/i44149/2u-quad-opteron-server.php (Date Last Accessed March 19, 2014)

[29] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[30] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[31] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE COMPUTER*, 2008.

[32] K. Kuzmin, S. Y. Shah, and B. K. Szymanski, Parallel overlapping community detection with SLPA," in *Social Computing (SocialCom), International Conference on. IEEE,* 2013, pp. 204-212).

[33] K. Kuzmin, M. Chen, and B. K. Szymanski, Parallelizing SLPA for Scalable Overlapping Community Detection," Special Issue on Computational Aspects of Social Network Analysis, *Scientific Programming*, 2014.