Proceedings of the IEEE Social Computing Conference, Washington DC, September 8-14, 2013

Parallel Overlapping Community Detection with SLPA

Konstantin Kuzmin, S. Yousaf Shah, and Boleslaw K. Szymanski
Department of Computer Science and Network Science and Technology Center
Rensselaer Polytechnic Institute (RPI), Troy, NY, USA
Email: {kuzmik, shahs9, szymab}@rpi.edu

***Abstract*—Social networks consist of various communities that host members sharing common characteristics. Often some members of one community are also members of other communities. Such shared membership of different communities leads to overlapping communities. Detecting such overlapping communities is a challenging and computationally intensive problem. In this paper, we investigate the usability of high performance computing in the area of social networks and community detection. We present highly scalable variants of a community detection algorithm called Speaker-listener Label Propagation Algorithm (SLPA). We show that despite of irregular data dependencies in the computation, parallel computing paradigms can significantly speed up the detection of overlapping communities of social networks which is computationally expensive. We show by experiments, how various parallel computing architectures can be utilized to analyze large social network data on both shared memory machines and distributed memory machines, such as IBM Blue Gene.**

***Keywords*-Social Networks; Community; Speedup; Message Passing Interface (MPI)**

I. Introduction

Social network analysis is a fast evolving field and social scientists are analyzing social behaviors from various facets. Social scientists loosely define community as a group of individuals sharing certain common characteristics [1]. They have been able to separate socially active agents into different groups based on certain characteristics. Yet, these social groups or communities overlap and social agents show up in various communities. Such social agents form overlapping communities and play roles in multiple social groups. These overlapping communities are very important due to the fact that they are powerful means of information dissemination. Advertising agencies, companies finding target markets, competitors looking for tipping points, social and political activists etc., are keen to find such overlapping communities to target them. Identifying and targeting such communities can lead to information cascades through which an opinion held by a minority of members may become the opinion of the community majority.

Most of the clustering algorithms have strong data dependencies, which poses a challenge to parallelization of such algorithms. Data sharing in parallel clustering algorithms can become a bottleneck. On the other hand, data synchronization can also reduce the performance of parallel algorithms in multiprocessor systems. Researchers have investigated shared-nothing, master–slave, and data replication techniques, which have their merits and drawbacks. Increasing data size is also an issue when parallelizing clustering algorithm. In our *MPI* based approach, we use minimal synchronization and no data replication. Every processor reads into memory only its own chunk of data. Every processor sends and receives data asynchronously. Processors ensure that their send and receive requests have completed before posting more such requests. In our proposed parallel algorithms, the increasing data size does not limit performance of the overall system. Moreover, results show that with increasing data size, the speedups of our parallel algorithms increase.

The rest of the paper is organized as follows. Section II provides an overview of relevant research. Section III describes parallel linear time overlapping community detection algorithm with message passing. We use MPI on Blue Gene parallel machine and evaluate speedup and efficiency of our approach on up to 1,204 processors. Section IV discusses multithreaded community detection on shared-memory multiprocessor that utilizes busy-waiting and implicit synchronization to ensure correct execution. We describe data partitioning and node rearrangement done to improve performance. We also discuss the speedup and efficiency accomplished by our approach. Finally, in Section V, we provide some final remarks and conclusions and briefly describe future work.

II. Related Work

Researchers interested in clustering and social network community detection have designed and investigated different algorithms of various complexities. The clique percolation technique [2] finds communities in a fully connected network by finding adjacent cliques in the graph. The k-means clustering algorithm [3] partitions a population in k clusters. Every node in a graph is assigned to cluster with the closest mean. In [4], an iterative scan technique is employed. In such an approach, nodes are iteratively added or removed in order to improve a density function. The algorithm is implemented using shared-nothing architectural approach. The approach spreads data on all the computers in a distributed 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

data size increases. Label propagation based community detection algorithms, such as LPA [5], COPRA [6] and SLPA [7], are based on the idea of simulating the propagation of labels in the network, where node ids can be used as labels. Nodes store, propagate and rank labels in each step. When the algorithm stops, nodes with the same highest probability label form a community. In this paper, we use SLPA as the basis for our parallelization.

Following the explosive growth of online social communities, recently the parallel approaches to community detection has been investigated. A parallel clustering algorithm is suggested in [8], which is a parallelized version of DBSCAN [9]. In [10], the authors coded their community detection algorithm based on propinquity using a vertex-oriented Bulk Synchronous Parallel (BSP) model to enable large scale parallelization. In [11], the authors implemented community detection algorithm on massively multithreaded Cray XMT and ran on it networks with over 100 million nodes and over a billion edges. Yet, we are not aware of parallelization efforts for overlapping community detection algorithms considered in this paper.

III. Parallel Linear Time Community Detection with Message Passing

The SLPA [7] 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 [7] 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 independent 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.

A. Parallel SLPA with Message Passing

In *MPI* based parallelized SLPA, we split the network into p partitions of nodes to be processed on p processors. Each processor gets its allocation of nodes that are flagged as 'local' and recreates network induced by local node by creating duplicates of nodes that are allocated to other processors but have an edge whose other node is the local one. These duplicated nodes are flagged as 'nonlocal'. After building a local network, each processor runs modified SLPA. At the end of each j iteration $where\ j \leq t$ in SLPA each processor p sends label lists of its local nodes to all other processors, so that they can update their duplicated nonlocal nodes. Also p updates its local duplicates of nodes with list of labels that it received from other processors.

Algorithm 1 shows our parallel SLPA algorithm. It runs on every processor p. The $numProcs$ is the total number of processors on which the algorithm is executed, $myRank$ is the $MPIRank$ of the processor, $InputFile$ is the social network input file. We use *Zoltan* partitioning library [12] to create load balanced partitions in parallel. In Algorithm 1, n is the total number of objects in a local network on a given processor whereas $Labels_i$ is the list of labels contained by node i. This parallel version of SLPA does not change its complexity, therefore the algorithm remains linear in time.

Algorithm 1 : Parallel SLPA

```
myPartition ← CreateZoltanPartition(InputFile)
CreateLocalNetwork(myPartition)
for j = 0 to j < t do
  for i = 0 to i < n do
    if i = myNode then
      l ← label from randomly picked neighbor
      Labels_i ← l
      SendBuffer[] ← i
      SendBuffer[] ← l
    end if
  end for
  for p = 0 to p < numProcs do
    if p ≠ myRank then
      Send SendBuffer to p
    end if
  end for
  Receive Labels_i from all p ≤ numProcs
  Update Label_i ∀i < n
  Reshuffle list of Nodes in Local Network
  SendBuffer[] ← NULL
end for
print  Most frequent labels to ouputfile
```

At the end of each run, we calculate the total execution time and also the time spent in MPI communication. We calculated speedup using formula shown in (1), efficiency according to (2).

$$Speedup = \frac{T_1}{T_n} \tag{1}$$

where $Speedup$ is the actual speedup calculated according to equation 1 and p is the number of cores.

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

After all processors have exchanged labels and the main label exchange execution has finished, every processor finds out the most frequent label in the label list of each node that it processed. These labels represent communities and are written to an output file. From all the output files produced by the processors, unique labels are extracted. They are representing the detected overlapping communities.

B. Performance Evaluation of the MPI Based Approach

We performed runs on a hyper threaded Linux system operating on 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 (2.1 GHz, 16-core, G34, 16 MB L3 Cache) central processing units operating over a shared 512 GB of Random Access Memory (RAM) (32 x 16 GB DDR3-1600 ECC Registered 2R DIMMs) running at 1600 MT/s Max. We also performed runs on IBM Blue Gene/Q stationed at The Computational Center for Nanotechnology Innovations facility at RPI, Troy, NY. Runs with number of processors varying between 1 and 32 were conducted on *GANXIS* whereas all runs with 64 processors and above were performed on Blue Gene/Q system. We used network with 1,087,562 nodes and average degree of 2.84, so the number of edges was over 3 million. We implemented the algorithm in C++ using *MPI* for parallelism.

Fig. 1 shows how running time varies with increasing processors. Clearly, the total running time decreases as we engage more and more processors to analyze the network. The total running time includes the time spent on communication between processors and time spent on execution of the algorithm itself. As we can see, increasing number of processors initially affects the execution time significantly but as more and more processors are added, the impact of increased number of processors declines. This is because as we increase the number of processors, each processor gets a smaller chunk of data to process. The running time increases on 64 processors compared to that time on 32 processors because of a change in system architecture. For 64 processors, we switch from *GANXIS* machine to Blue Gene. Trends in execution on both architectures are similar as execution time decreases with increasing number of processors. The *GANXIS* machine is a shared memory machine with 512 GB of RAM and the individual processors of *GANXIS* are more powerful than those of *Blue Gene/Q*. We used *Zoltan* partitioning software [12] to partition graph in a way that balances the load of processors. Yet, as shown in Algorithm 1, for the large number of processors (512 or more) even small imperfections of partitioning negatively impact speedup. If we look at the computation time trend in Fig. 2, which shows time spent on running the actual algorithm by excluding the communication time, we can see that we spent even less time in actual computation than indicated on the previous plot.

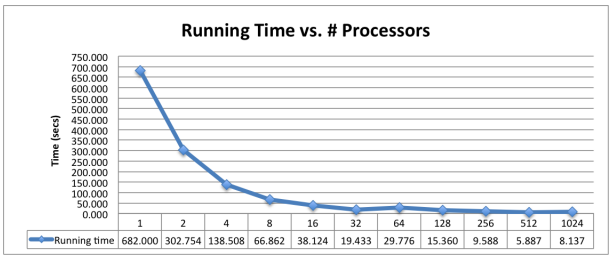

Figure 1. Total running time taken for detecting communities in the network.

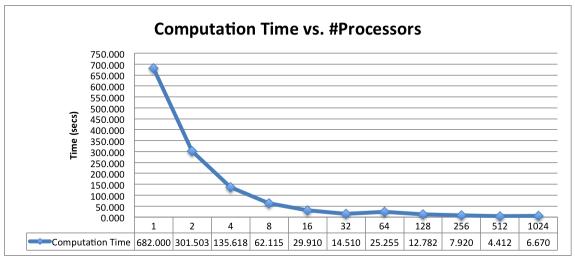

Figure 2. Actual computation time taken for detecting communities in the network.

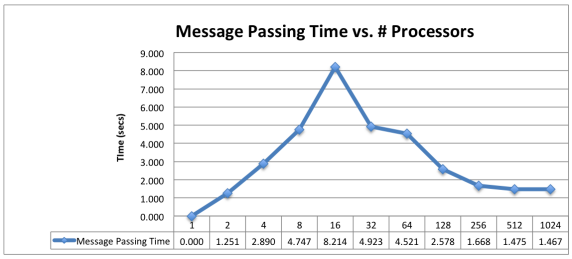

Figure 3. Time spent on communication during community detection in the network.

We can see from Fig. 3 that as we increase the number of processors, the time spent on communication initially increases, even though it is still very small. The increase is specially noticeable on GANXIS (1–16 processors). In communication time plot, just like execution time plot, the line trend changes as we switch to Blue Gene. On GANXIS machine, there is a significant increase in communication time with increasing number of processors (the exception is the case of 32 processors resulting from Zoltan improved partitioning) whereas the communication time is decreasing in case of Blue Gene. The sharp decline in communication time on 64 processors upwards as compared to 2–32 processors is caused by the high bandwidth torus network that is in place for communication among processors in Blue Gene/Q. The plots also show how efficient the Blue Gene communication backbone is as compared to a hyper threaded multi-core processor system in which the bandwidth is limited by the bus architecture.

Fig. 4 shows the speedup achieved by increasing the number of processors. Higher speed up is obtained by

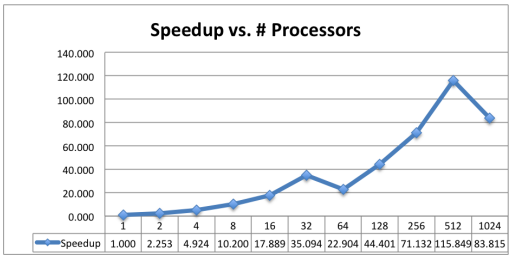

Figure 4. Speedup achieved with increasing number of processors.

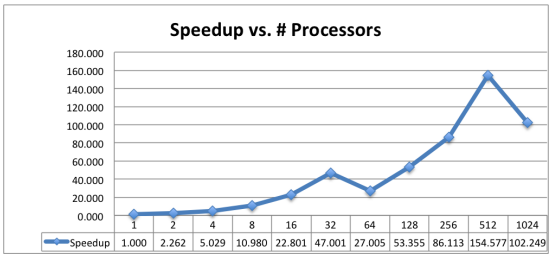

Figure 5. Speedup achieved with increasing number of processors based on computation time.

Blue Gene with increasing processors in the range 64–512 processors thanks to high bandwidth network. The speedup comes down for 1024 processors because with such a high number of processors, they do not get enough data to process compared to the communication between processors lowering the efficiency of parallel execution. Fig. 5, shows speedup calculated based on the actual computation time (excluding communication time). As expected, we see higher speedup when only the actual computation time is taken into consideration.

IV. 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 Fig. 6. $T_1, T_2, ..., T_p$ 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 $m_1, m_2, ..., m_k$. 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 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. Fig. 7 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.

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

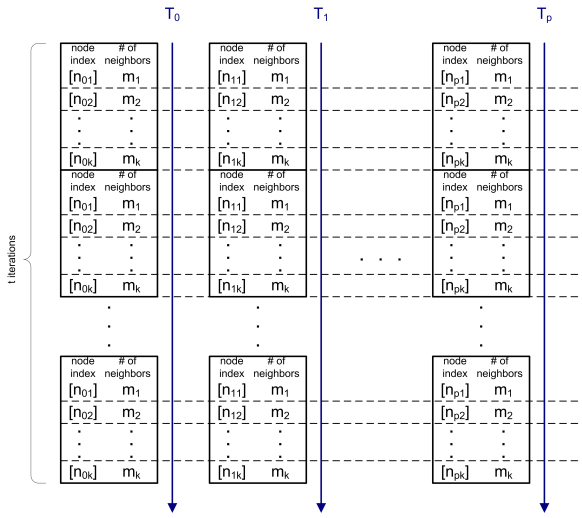

Figure 6. Ideal partitioning of the network for multithreaded SLPA.

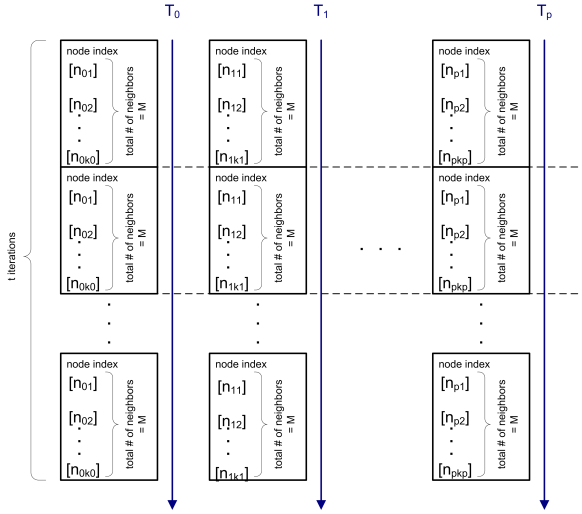

Figure 7. A better practical partitioning of the network for multithreaded SLPA.

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 2 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.

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 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

Algorithm 2 : 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
```

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 Section IV-A) 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 3 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 2 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.

Algorithm 3 : Multithreaded SLPA with splitting of nodes

```
Internal ← CreateInternalPartition(InputFile)
External ← CreateExternalPartition(InputFile)
p ← number of threads
/* Unchanged code from Algorithm 2 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
```

A. Evaluation of the multithreaded approach

The GANXIS machine detailed in Section III-B was used for all the experiments. The source code was written in C++03 and compiled using g++ 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5).

All the experiments were run with 100 iterations (the value of $maxT$ was set to 100) on a network with 4,350,248 nodes and 12,332,112 edges. The network is 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 node i.

All measurements include 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 is not included in the timing. All post-processing steps and writing an output file have also been excluded.

Since the hardware platform that we utilized provides 64 cores, every thread in our tests executes on its dedicated core. Therefore threads do not compete for central processing unit (CPU) cores, 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 32 because there is a performance degradation for a number of threads bigger 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 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.

Fig. 8 shows the time it took to complete the label propagation phase of the multithreaded SLPA for the number of cores varying from 1 to 32. It can be seen that the time decreases nearly linearly with the number of threads. It can be noticed that the rate at which time decreases with the number of cores falls for larger number of cores. This trend is even more evident in Fig. 9 which shows speedup and efficiency obtained for the same runs. As the number of cores increase, the speedup grows faster but not as fast as the number of utilized cores, so efficiency drops. 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). Larger number of nodes with external dependencies, in turn, makes the thread more dependent on other threads.

Fig. 10 shows label propagation times of the multithreaded version of SLPA which splits nodes into a subset

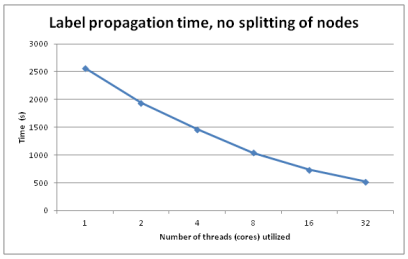

Figure 8. Label propagation time for a network with no splitting of nodes.

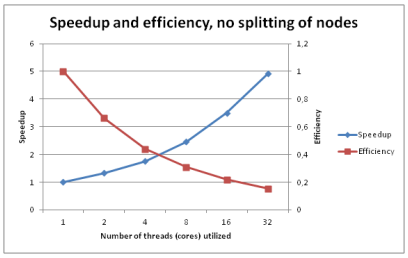

Figure 9. Speedup and efficiency for a network with no splitting of nodes.

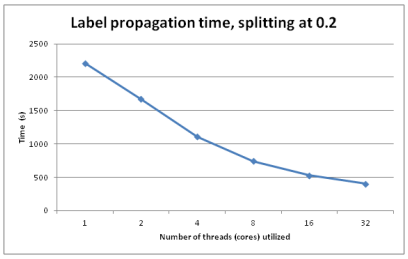

Figure 10. Label propagation time for a network with splitting of nodes.

of nodes that have neighbors 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. Fig. 11 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 support our expectations that minimizing the waiting time that a thread spends in a busy-waiting loop while other threads are catching up will make our code run faster and more efficiently.

The benefit of splitting the nodes can further be examined by looking at Figs. 12 and 13. For the entire range from 1 to 32 cores the version with splitting outperformed the alternative 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

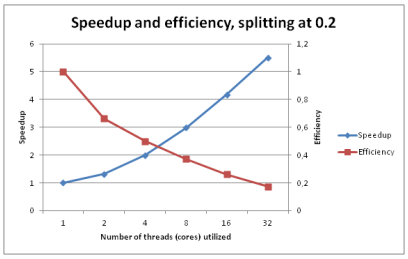

Figure 11. Speedup and efficiency for a network with splitting of nodes.

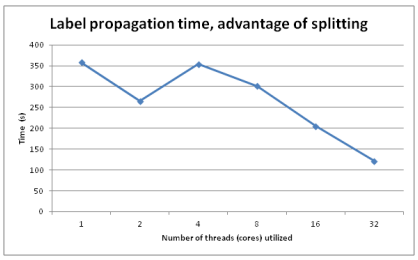

Figure 12. Label propagation time advantage (as a difference between running time for a version without and with splitting).

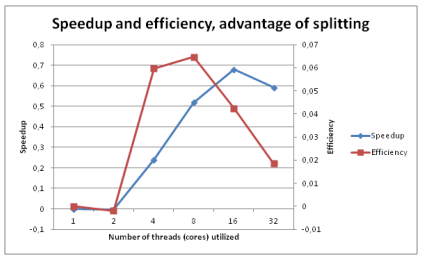

Figure 13. Speedup and efficiency advantage (as a difference between speedup and efficiency for a version without and with splitting).

increased to more than 8. These effects are related to the behavior of split subsets of nodes when the number of cores increases, as described above.

V. Conclusion and Future Work

In this paper, we have investigated parallel algorithms for community detection in social networks and the use of a super computing paradigm in social networks analysis. We found that SLPA runs faster with increasing number of processors. With a sufficient number of processors used, the parallel SLPA can process social networks with millions of nodes in seconds.

In our future work, we plan to explore a combined approach for massive parallelism in which message passing is used to communicate between processors but each processor launches several threads to process network fragment assigned to it in parallel on available cores.

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.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science Engineering*, vol. 4, no. 2, pp. 90–96, 2002.