Parallel Community Detection in Large Graphs

A Survey of the State-of-the-Art Algorithms

Xinghuang Xu

Wichita State University

Wichita, Kansas

xxxu3@wichita.edu

*Abstract*—the problem of finding communities in graph has been proven valuable in a lot of areas such as computer science, physics and sociology where systems can be modeled as graph. Although a lot of work has been devoted into finding an elegant solution, no satisfactory solution has been found so far. The problem has been made more challenging under the era of big data when graph can easily have millions of nodes and billions of edges and can grow in a high velocity. This paper mainly reviews several types of parallel community detection algorithms including algorithms targeting overlapping communities. It serves as an introduction as well as a summary of the field of parallel community detection.

Keywords—community detection; network; graph; parallel; big data; algorithm;

# Introduction

Graph has become a very useful tool in representing complex networks. Complex network graphs in biology, computational genomics and computational biology within scientific domains, World Wide Web and social networks in the IT domains require topological structural analysis to understand the complex nature of interactions amongst the entities. One important analysis in mining complex networks is to find the communities inside them. A lot of times communities can have overlapping between each other. For example, a person is usually simultaneously belonging to multiple communities (family, college, work and etc) in a social network. So it doesn’t make sense to put such a person in any single one of those communities, detecting overlapping community is a very important research areas that have been neglected in a lot of community detection literatures.

The problem of community detection (CM) and the concept of community is not rigorously defined. There are many different definitions of the problem and what constitutes a good community in past years literatures. Among these definitions, the lowest common denominator is that a community has dense intra connections and sparse inter-connections [2]. Community detection should not rely on prior information related to clusters such as the number of clusters and communities can overlap with each other which distinguish the problem from graph clustering and graph partitioning [2]. A wide variety of algorithms have been proposed and most of them can be found in the surveys by Schaeffer [1] and Fortunato [2]. Community detection is known to be an NP-complete problem so exhaustive enumeration of all the possible solutions is impractical. Most of the algorithms in [2] apply approximation techniques to provide an approximate solution with the advantage of lower complexity.

DIMACS challenges are scientific competitions in which the participants solve problems from a specified test set, with the aim of high solution quality and high speed. In 2012, the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering tries to assess the state-of-the-art community detection algorithms. Only two of the 15 submitted implementations based on modularity approximation relied on parallelism and only very few could handle graphs with billions of edges in reasonable time.

In the age of big data, it is not uncommon to see graphs with millions to billions of edges. For example online social network Facebook reports as of May 2015 have more than 1.23 billion monthly active users. The gap between networks processable with current algorithms and many networks of interest is at least one order of magnitude. So far, algorithms proposed for community detection in networks are almost exclusively non-distributed, in-memory and single threaded algorithms. This paper intends to survey the state-of-the-art algorithms or frameworks which can handle graphs with millions of nodes in a reasonable time. In the rest of the paper, in section II a prerequisite will be given to familiarize readers with some basic building blocks of the upcoming algorithms. A detailed survey of the most recent parallel algorithms are followed in Section III. Section IV will conclude and touch on future works.

# Prerequisite

## Quality Measurement

Because different algorithms have different objective functions based on their goals, it is generally hard to compare community detection algorithms. The most widely accepted quality function is the modularity proposed by Newman and Girvan [3]. It is based on the idea that a random graph is not supposed to have community structures. Modularity can be defined as Q = (Ein) - (Expected number of such edges), where Ein is the number of intra community edges. A large Q would indicate that there are more edges within communities than we would expect on the basis of chance. The expected fraction of edges is typically evaluated within the configuration model, a random graph conditioned on the degree sequence of the original network. The probability of an edge between two vertices I and j is kikj/2m, where ki is the degree of vertex i and m is the total number of edges in the network. The modularity can then be written as:

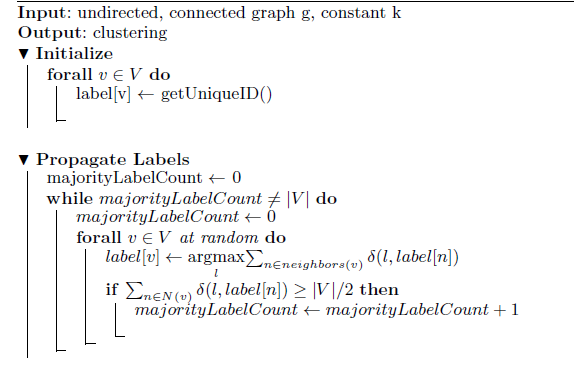
Where is an element of the adjacency matrix, is the Kronecker delta symbol, and is the label of the community to which vertex i is assigned. A lot of community detection algorithms works by gradually improving Q. Even though it has been shown that modularity suffers from a resolution limit which can be partially overcome by different technique, it remains to be method of choice for benchmarking community detection algorithms.

# Basic Community Detection Algorithms

This section presents several fast community detection algorithms which runs in sublinear time. They are also easy to parallel and run on high performance computing architectures. The detailed parallel version of the algorithms will be covered in Section V.

## Label Propagation (LP)

Raghavan, Albert and Kumara [6] proposed a label propagation algorithm which runs in near linear time which has become quite popular lately because of its speed and ease of parallelization. The algorithm does not optimize any specific measure or function like most other community detection algorithms. The main idea behind of label propagation algorithm is the following. Suppose that a node x has neighbors x1, x2... xk and that each neighbor carries a label denoting the community to which they belong to. Then x determines its community based on the labels of its neighbors. We assume that each node in the network chooses to join the community to which the maximum number of its neighbors belong to, with ties broken uniformly randomly. Each node is initialized with unique labels and let the labels propagate through the network. As the labels propagate, densely connected groups of nodes quickly reach a consensus on a unique label (see figure 2). When many such dense (consensus) groups are created throughout the network, they continue to expand outwards until it is impossible to do so. At the end of the propagation process, nodes having the same labels are grouped together as one community. The above process is perform iteratively, each process would take O(m) time, where m is the edge of the graph. Even mathematical convergence is hard to prove, it’s easy to observe that the LP algorithm beginning to converge significantly after a small number of iterations.



## Speaker-listener Label Propagation Algorithm (SLPA)

Based on the Label Propagation Algorithm, Xie and Szymanski [11] proposed Speaker-listener Label Propagation Algorithm SLPA which has the ability of discovering overlapping communities by allowing nodes to possess multiple labels. The simplified version of the algorithm is described in Figure x. Due to step c, the label history of each node increase by one in each step. The communities is performed in the post-processing where communities are extracted from analyzing the label histories of each nodes. If the probability of seeing a particular label in the label history is less than the threshold parameter r [0,0.5] is deleted. Nodes having the same label are grouped together as a community. Since nodes can have multiple labels after the threshold filtering, they can be assigned to multiple communities and be treated as overlapping nodes.

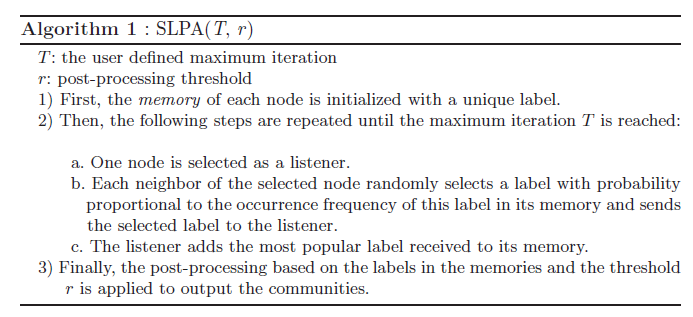


Figure from [11]. SLPA in a nutshell

## Weighted Label Propagation Algorithm (WLPA)

WLPA based on LP and only make use of local information. In WLPA, the assignment of weights to edges determines how the labels propagate through the network. Edge weights that implicitly represent accurate topological structure of the inherent communities in the network should be preferred. The following method is being used in WLPA to assign weights to edges. Considering an unweighted directed graph, the weight for the directed edge is as follow:

For undirected graph, each edge is replaced by two directed edges. For weighted graph, the product of the given weight and topological loops based weight mentioned above is consider the new weight for the edge given by:

The propagation function for label propagation can be defined as:

where is the set of neighboring vertices of vertex I; the total weight for the label L(j) in the neighborhood of vertex i.

Several problems with LP and WLPA are presented in the paper. The first one is due to the fact that the labels on two nodes (v1,v2) such that v1 is paired to v2 using the maximum edge weight and vice versa can oscillate without every converging. The solution is to give the same label to local maxima pairs. Another problem is caused by large communities dominating over small ones called epidemic spread. To avoid this problem, a weight to each label is being assigned such that the large the community the smaller the weight as follow:

where, is the label of community c; dc is sum of the degrees of all nodes inside the community c; and M is the total number of edges in the graph. The new propagation function after incorporating the above label weight becomes:

WLPA can also be used to find overlapping communities by adding a post processing step. A node can be in multiple communities if the connectivity of the node is strong to more than one community by analyzing the edge weight sum of each communities the node is close to. Figure x shows the pseudocode of the algorithm.

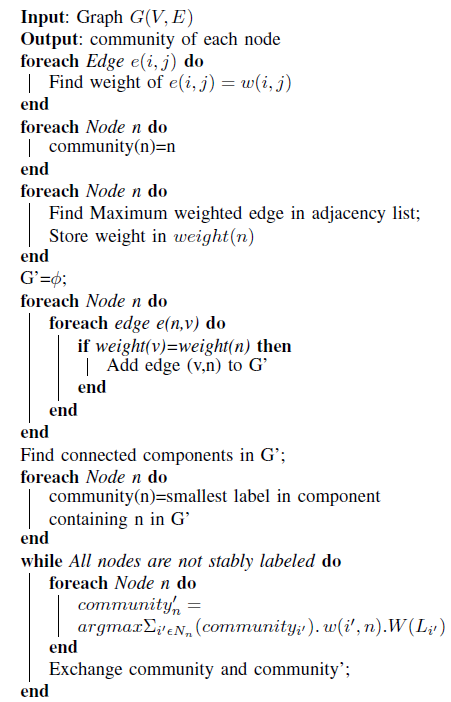


Figure x from [7]. The weighted label propagation algorithm

## Randomized Greedy (RG)

RG is one of hierarchical agglomerative algorithms. The hierarchical algorithms start with each point in its own cluster. Two clusters are combined in each step if the combination would cause of maximal increase in modularity. The result is a cut of the dendrogram with the maximal modularity. A dendrogram is show at Figure x. The complete search over all combination of clusters is extremely slow and also leads to an unbalanced merge process. Some clusters grow faster than others and the size difference is a bias for later merge decisions. Large cluster would eat up small ones. RG is a fast algorithm which does not suffer from the unbalance merge process. RG works by selecting in every step a small sample of clusters and determines the best merge inside this sample. Because of the sampling technique, RG has a complexity of roughly. See figure x for the pseudocode.

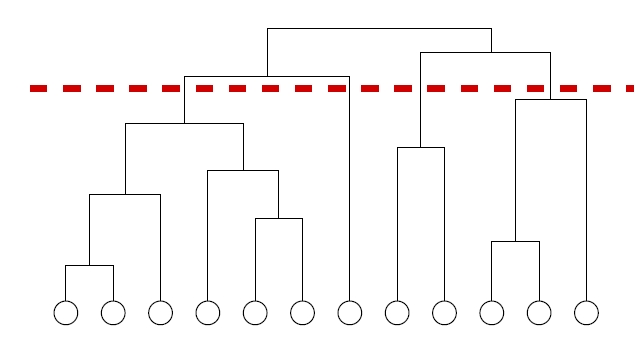
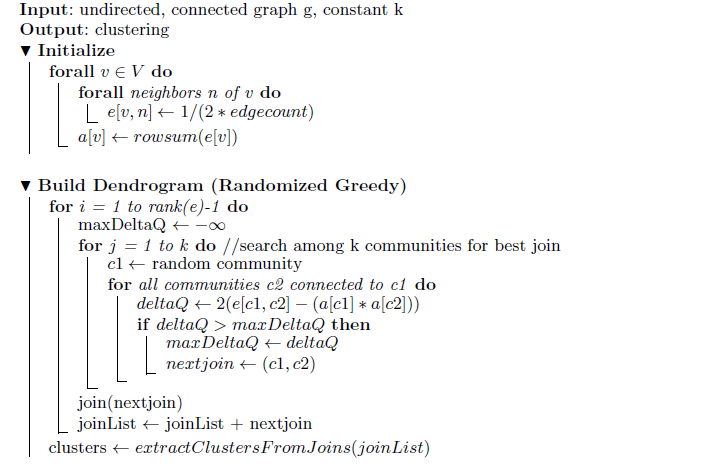


Figure x from [2]. A dendrogram. Horizontal cuts correspond to partitions of the graph in communities.



## Ensemble Learning Based Algorithm (CGGC)

Ensemble learning is a paradigm in machine learning, where several intermediate classifiers (called weak or base classifiers) are generated and combined to finally get a single classifier. The algorithms used to compute the weak classifiers are called weak learners. An important notion is, that even if a weak learner has only a slightly better accuracy than random choice, by combining several classifiers created by this weak learner, a strong classifier can be created [10]. Instead of regarding community detection as an optimization problem (optimize objective functions like modularity), it could be seen as a learning problem as well.

Ovelgönne and Geyer-Schulz [9] apply the ensemble learning paradigm to community detection. They develop what they call the Core Groups Graph Clusterer (CGGC) scheme, using sublinear runtime algorithms Randomized Greedy, and Label Propagation as base algorithms. The general outline of a CGGC for community detection is roughly as follows [9]:

First, a set of partitions with some (weak) learning algorithms are created. Second, the maximal overlap of the partitions as depicted in Figure 1 is identified. Third, continue the search from the maximal overlap with the algorithm used in the first step or any other appropriate algorithm.

Because some data points are harder to assign than others, an iterated scheme is introduced. In this schema, the core communities are again assigned to an ensemble creating a hierarchy of solutions/coarsened graphs until quality does not improve any more. The performance is evaluated by running the CGGC scheme using RG or LP as base algorithm in iterated mode and non-iterated mode. The tested dataset comes from the 10th DIMACS implementation challenge. The result indicates that CGGC achieves modularity close to the best modularity obtained from the DIMACS challenge. Another surprising result is that CGGCiRC have little performance different compare to CCGCiLP after a few iteration given that LP is a much weaker learner than RC. Hence the choice of the base algorithm for CGGC has little impact on the clustering quality.

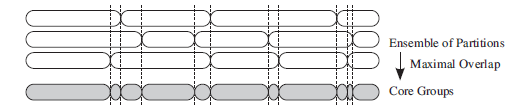


Figure 1 from [8]. The bottom row is the core groups partition that are in the same communities in all partitions of the ensemble (3 top rows).

# Parallel Programming Models and Frameworks

This section covers a few important parallel programming models and frameworks. It’s necessary to understand the ideas in this section to comprehend the parallel algorithms in Section V.

## GPU Computational Model

General-purpose computing on graphics processing units (GPGPU) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU). Unlike CPUs, however, GPUs have a massively multi-threaded architecture containing hundreds of processing elements (cores) that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly.

For example, the new Fermi GF100 is a GPU architecture from NVIDA that provides several new capabilities. It has 16 Streaming Multiprocessors (SM) with 32 SIMD cores per SM. The cores inside an SM execute the same instruction in a SIMD fashion. With a total of 512 cores, 512 32 bit integer or floating point operations can usually be completed in one clock cycle. The internal memory provided uses GDDR5 DRAM. A Bidirectional PCI express link connects it to the system memory with the transfer rate of 12GB/second.

CUDA stands for Compute Unified Device Architecture is a parallel computing platform and programming model created by NVIDIA and implemented by the GPUs that they produce. CUDA provides developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs through CUDA-accelerated libraries, compiler directives, and extensions to industry standard programming languages including C, C++ and FORTRAN. Figure x shows the processing flow on CUDA. In CUDA, threads are grouped into blocks and blocks make up a grid. Blocks are serially assigned for exection on each SM. The blocks themselves are divided into SIMD groups called warps, each containing 32 threads. An SM executes 1 warp at a given instance, with up to 48 active warps at a given time. CUDA enables zero overhead scheduling, which allows warps that are stalled on a memory fetches to be swapped for another warp with immediately available computation.

Computations that are to be performed on the GPUs are specified in the code as explicit kernels. Prior to launching the kernel, all the data required for the computation must be transferred from the Host memory to the GPU memory. A kernel invocation wil hand over the control to the GPU, and specified GPU code will be executed on this data. Barrier synchronization for all threads in a block can be defined by the user in the kernel code. All the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches.

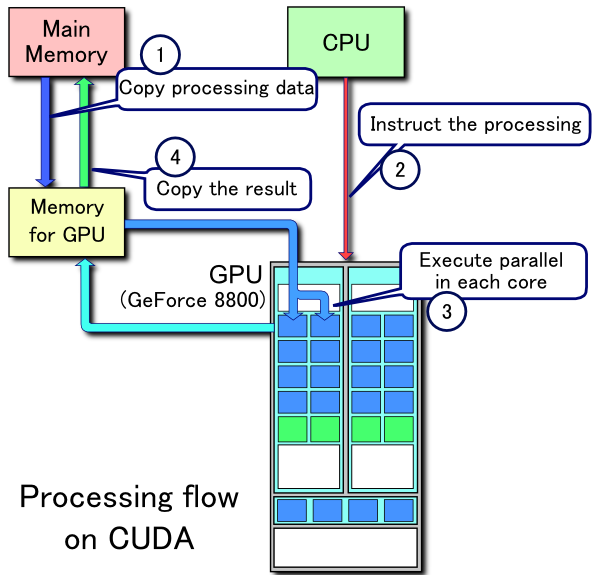


Figure from Wiki.

## MPI and OpenMP

MPI stands for Message Passing Interface. It’s a language-independent communication protocol used in parallel computers. It is developed to address the problem of message passing in parallel programming model. The protocol specifies how to move data from the address space of one process to that of another process through cooperative operations on each process. Originally, MPI was designed for distributed memory architectures. As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory shared memory systems. Most MPI implementations nowadays handle both types of underlying memory architectures seamlessly. Today MPI it runs on virtually any hardware platform, so programs can be easily ported between hardware platforms. As the dominant model used in high performance computing, it has become a de facto standard. Open MIP is an open source implementation of MPI <http://www.open-mpi.org>.

OpenMP, Open Multi-Processing, is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and FORTRAN, on most processor architectures and operating systems including Solaris, Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP and MPI can be considered as complementary programming approaches and can occasionally be seen together in applications e.g. in servers with multiple large shared-memory nodes. Table x list compare MPI and OpenMP

|  |  |  |
| --- | --- | --- |
|  | OpenMP | MPI |
| Pros | * Easier to program and debug than MPI * Directive can be added incrementally * Can still run the program as a serial code * Code is easier to understand and maybe more easily modify | * Runs on either shared or distributed memory architectures * Can be used on a wider range of problems than OpenMP * Each process has its own local variables * Distributed memory computers are less expensive than large shared memory computers |
| Cons | * Can only be run in shared memory computers * Requires a compiler that supports OpenMP * Mostly used for loop parallelization | * Requires more programming changes to go from serial to parallel version * Can be harder to debug * Performance is limited by the communication network between the nodes |

Table x. Pros and Cons of OpenMP and MPI

## Hadoop

Hadoop is an open-source software framework written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. It addresses three main challenges of cluster computing with commodity hardware. The first challenge is that in a cluster of over hundreds of machines, the probability of a single machine failure is high. How to address a machine failure during the run time of a parallel program? The second challenge is the communication cost between machines is too high. Clusters with Ethernet connections can only communicate in 1GB/s. Massive datasets are usually in the size of Terabytes. The third challenge is that parallel programming in MPI or Open MP is too difficult. Debugging parallel programs is a nightmare due to factors like race condition and non-deterministic execution sequence.

Hadoop framework consists of two main components Hadoop Distributed File System (HDFS) and the MapReduce engine. They work together to solve the three challenges above. HDFS stores large files across multiple nodes and provides false tolerance by replicating the data across multiple nodes. The MapReduce engine consists of a JobTracker and one TaskTracker per node. The JobTracker splits up the workload and assigns each part to one of the TaskTrackers. HDFS and MapReduce engine works together to make sure the tasks are running on the machines that also stores the required data which greatly reduce the communication cost between nodes. In case of node failure, MapReduce engine would reassign the task to another node and the HDFS would recover the data on the node by using the redundant information store on multiple other nodes. In order to simplify parallel programming, Hadoop by itself only allows a very restricted programming model called MapReduce that severely limits communication between tasks in order to allow extreme scalability of program execution. Hence not all problems are suitable to be solved in Hadoop. In the MapReduce paradigm, developers only have to implement two functions: the map function and the reduce function. The Hadoop framework would take care of the underlining data flow, job distributions and other parallel optimizations. In a nutshell, a Hadoop MapReduce job consists of three phases as follow:

1. The HDFS divides the data into chunks and distribute them to some number of map tasks. The map tasks turn the chunk into a sequence of key-value pairs. The parsing from the input file to the key-value pairs are defined in the Map function.
2. After the map phase, the key-value pairs from each Map task are collected by a master controller and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key end up at the same Reduce taks.
3. The input to the Reduce tasks is the sorted key-value pairs. The functionality of the Reduce task is defined in the Reduce function. In most cases, the key-value pairs are combined in a way to produce the expected output.

In some tasks, the pipelining of multiple MapReduce cycles are needed.

# Parallel Community Detection Algorithms

This section builds on the knowledge in the previous sections and presents the state-of-the-art parallel community detection algorithms for different parallel computing architectures.

## CGGC + Hadoop

As stated at the end of [9], in ensemble learning, the quality of the base algorithm is not that important. Hence label propagation which is extremely fast and only make use of local information becomes most suitable for distributed computing environment. Ovelgönne presented a distributed implementation of the aforementioned CGGC using PL with advance initialization as the base algorithm on the Hadoop big data framework in [8].

A Hadoop job consists of three phases. First the map phase, then the combination phase, follow by the reduce phase. Every iteration consists of sweeping through the set of vertices and updating each vertex label is executed as a Hadoop job. To identify core groups, an ensemble of partitions need to be calculated. It would be unwise to create the partitions for the ensemble in sequence so all the partitions are computing in parallel. Instead of initialize every vertex with a unique label and propagate it through the network, each vertex i a vector consisting of r labels and all r labels of vertex i are updated in parallel but independently. It is best to bundle the propagation of labels as vector instead of single label which greatly reduce the amount of data transfer overhead due to the reason that blocks can be transferred far more efficiently.

Since homogeneous partitions in the ensemble are not desired, it is crucial to adjust the label propagation algorithm to create a diversive and relatively high quality partitions. To solve this problem, an advanced initialization strategy which directs the label propagation convergence process in different directions in each parallel label propagation run by giving a different randomly chosen set of vertices a head start is used.

This distributed implementation could process graphs as large as ~3.3 billion edges on a relatively small Hadoop cluster consisting of 50 nodes in just a few hours. As Hadoop scales very well in the number of nodes, even large graphs could be processed in just a few hours on large Hadoop installations [9].

## SLPA + MPI

In [12], the developer of SLPA devised an MPI based approach to speed up the algorithm. Three different parallel computing algorithms are proposed in [12]. The first one distribute work to each processors by partition the input graph using Zoltan Partition into the same number of partitions as processors. Each processor would handle a partition. Nodes in a partition is classified as local has no replicate in other partitions or nonlocal which is replicated and has at least one local as neighbor. Each processor runs modified SLPA and by the end of an iteration each processor sends label lists of its local nodes to all other processors, so that they can update their duplicated nonlocal nodes. It is showed that increasing processes would reduce the running time of the algorithm. Whether the initial graph partition algorithm creates a load balanced partition is crucial to the performance.

In order to achieve better work load balancing, another algorithm which divide the computation into threads. Each thread handles a subnetwork resulting from the partition of the original network. The runtime of this threaded model would heavily rely on the partition quality. The best partition which makes every thread spend the same amount of time processing each node is hard to achieve due to uncertainty associated with thread scheduling and variations in thread start-up. Multiple partition approach are discussed in the paper. The actual implementation use a very simple approach by simply assign nodes to threads sequentially based on the time it arrives.

The third algorithm is proposed to further alleviate the synchronization burden between the threads and minimize the sequentially of the threads as much as possible by utilizing another optimization technique. The optimization technique split the subset of nodes processed by a thread into two subsets, one contains only nodes with no external dependencies and the other contains the remaining nodes. During the label propagation phase, nodes with external dependencies are processed first in each iteration. As a result, thread can report its completion of the iteration earlier than it has in fact been completed by relying on the fact that the remaining work on node with no external dependencies cannot influence nodes in other threads. This approach works well on networks with high locality of edges especially if the input file is sorted in the order of node numbers. The approach seemingly simple leads to noticeable improvement of the efficiency of parallel execution.

The proposed parallel algorithms are run on two type of systems using a graph consists of 1 million nodes and 3 million edges. A hyper threaded Linux system operating on a Silicon Mechanics Rackform nServ machine (GANXIS). GANXIS has 64 cores organized as four AMD Opteron 6272 (2.1 GHz, 16-core, G34, 16MB L3 Cache) central processing units operating over a shared 512 GB of RAM is used to collect computation time with processors varying from 1 to 32. Statistics on 64 or more processor is perform on an IBM Blue Gene/Q that has more than 1024 cores. Figure x shows that with more processors the computation time decreases and stabilized at around 100 processors usage.

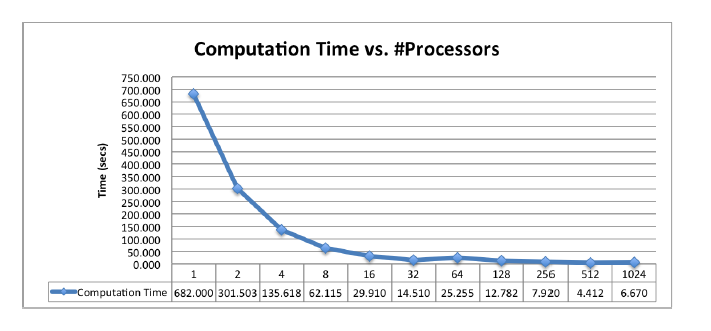


Figure x from [12]. Actual Computation Time running algorithm 1

## WLPA + GPGPU

Because WLPA only use and update local information, it has inherent fine-grained parallelism and requires minimal synchronization. Each edge operation is independent of other edge operations so that updating the similarity matrix is lock free. ­­GPGPU architecture prefers algorithms that are data parallel in nature, unfortunately graph algorithms are typically not in that group. To make WLPA work well on GPU, each component of the algorithm is map to a known data parallel algorithm optimized for the GPU. Figure x shows the parallel version of MLPA. Finding the label of maximum weight in the neighborhood of a node is converted to a segmented sort on an array. To find the majority node in the neighborhood of a node, Bitonic sort is used. Bitonic sort is an in place parallel sort that requires no additional memory.

Two things are of concern when comparing WLPA with other algorithms. One is the community quality, the other the actual run time. Experiments are conducted on both small datasets and large datasets. On small dataset, the modularity performance of MLPA is just slightly worse than the betweenness centrality based algorithm and the walktrap algorithm. On large datasets, the running time is being compare with another algorithm by Zhang et.al.[27]. To process the hep-th dataset consisting of 27 thousand nodes and 352 thousand edges, Zhang et.al. Algorithm takes 100s on a 50 machines cluster each with 1GHz CPU and 1GB RAM comparing to WLPA which only takes 0.27s on a 32 IBM Power6 Shared Memory Processors. The paper also show that running WLPA on GPGPUs is around 8 times faster than running on Multi Cores Share Memory Architecture.

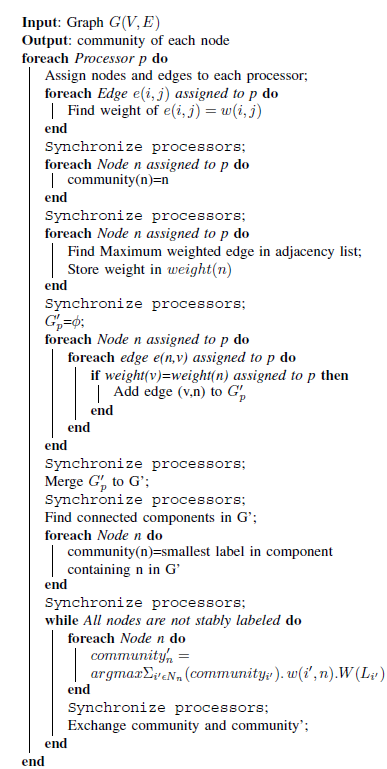


Figure x from [7]. Parallel WLPA

# Future Work and Conclusion

The survey shows that the most popular parallel community detection algorithms are based on LP because it only utilize local information and runs extremely fast. SLPA and WLPA are variants of LP designed to handle a few drawbacks of LP. The quality of the communities found by SLPA and WLPA have similar modularity. CCGCiLP achieves better modularity among the three. Even though it’s hard to compare the run time of those algorithms because they are running on different hardware with different computing architectures (GPGPUs, Hadoop, MPI etc). It’s intuitive to say that CCGCiLP would require more runtime simply because it takes more LP iterations. It’s hard to compare SLPA and WLPA, even they are variance of LP. WLPA is optimized for GPGPU architectures while SLPA is optimized for traditional shared/distribute memory machines. It would be interesting to compare them on the same ground which means running both algorithms on systems costing the same money value.

This paper only surveys three parallel algorithms. There are a lot more out there. An intrinsic problem the field is currently facing is how to compare the performance between different algorithms. It’s chaotic because new algorithms are being evaluated on various different datasets running on different hardware/software architectures.