2011 IEEE International Parallel & Distributed Processing Symposium

Fast Community Detection Algorithm With GPUs and Multicore Architectures

Jyothish Soman
IBM Research India
New Delhi, India
jyosoman@in.ibm.com

Ankur Narang
IBM Research India
New Delhi, India
annarang@in.ibm.com

***Abstract*—In this paper, we present the design of a novel scalable parallel algorithm for community detection optimized for multi-core and GPU architectures. Our algorithm is based on label propagation, which works solely on local information, thus giving it the scalability advantage over conventional approaches. We also show that weighted label propagation can overcome typical quality issues in communities detected with label propagation. Experimental results on well known massive scale graphs such as Wikipedia (100M edges) and also on RMAT graphs with 10M - 40M edges, demonstrate the superior performance and scalability of our algorithm compared to the well known approaches for community detection. On the *hep-th* graph (352K edges) and the *wikipedia* graph (100M edges), using Power 6 architecture with 32 cores, our algorithm achieves one to two orders of magnitude better performance compared to the best known prior results on parallel architectures with similar number of CPUs. Further, our GPGPU based algorithm achieves $8\times$ improvement over the Power 6 performance on 40M edge R-MAT graph. Alongside, we achieve high quality (modularity) of communities detected, with experimental evidence from well-known graphs such as Zachary karate club, Dolphin network and Football club, where we achieve modularity that is close to the best known alternatives. To the best of our knowledge these are best known results for community detection on massive graphs (100M edges) in terms of performance and also quality vs. performance trade-off. This is also a unique work on community detection on GPGPUs with scalable performance.**

***Keywords*-Parallel Community Detection; Parallel Time Complexity; Performance Analysis; Multi-core Architectures; GPGPU; Social Networks;**

I. Introduction

Graphs have become an important tool to structure relations among entities. Scientific domains such as computational biology and bioinformatics, computational genomics and IT domains such as the world wide web, telecommunications domains involve massive scale graphs interacting via numerous underlying mechanisms. These graphs are in general directed and the weights on the edges of the graph represent flow capacity or impedance or affinity between nodes (entities) of the graphs. These complex networks require topological structural analysis to understand the complex nature of interactions amongst the entities. One important analysis in these complex networks is finding clusters of graph vertices, i.e. the division of the network into groups (clusters or modules) of nodes having dense intra-connections, and sparse inter-connections[10]. We focus on the problem of finding communities from networks where the number of communities as well as any prior information related to the clusters is not available. This problem is referred to as the *Community Detection* problem. The term *graph* implicitly implies a directed graph and is used interchangeably with the term *network* in this paper.

Community detection is closely related to network partitioning, where the network has to be partitioned into fixed number of groups, such that each group has additional constraints, such as edge/node count maximum limit, or intra group edge count maximum limit and so forth. Graph partitioning is well defined, but community detection algorithms rely on intuitive definitions to form heuristics, which can find communities in a graph [22], [11]. The algorithms assume that the underlying graph has dense subgraphs, which are denser than the original graph. Community detection algorithms, hence depend on finding these dense subgraphs whose structure is domain dependent. It has been shown that many graphs in real world show such topological properties such as dense subgraphs, power law degree distribution and low diameter and these are common across many domains such as social networks, metabolic networks, gene networks and so forth.

Communities in graphs have only been loosely defined, though multiple definitions and methods of estimating quality of the communities exist [19]. A majority of current work focuses on improving the *modularity* function, which tries to increase the number of edges inside a cluster, compared to edges between nodes of different clusters. We use the modularity measure to evaluate the quality of communities detected by our algorithm and compare this value with the modularity obtained from the best algorithms known so far. The literature survey on community detection in [10] provides in-depth overview of techniques known so far. A majority of the techniques used for detecting communities can be divided into two types of hierarchical approaches: *agglomerative* and *divisive* [10]. The implicit assumption in both these approaches is that the communities have a nested topological structure. However, it has been shown that communities in large graphs do not necessarily show a nested structure. Hence, hierarchical algorithms do not necessarily provide the best solution.

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.61

Community detection is known to be an NP-complete problem [6]. Thus, exhaustive enumeration of the solutions of the same is impractical and one has to employ heuristics to achieve high modularity solutions. Community detection can be considered as a super set of graph partitioning. Though efficient parallel algorithms for graph partitioning are present, parallel implementations for community detection rely on inherent parallelism in sequential algorithms. Both *divisive* (top-down) approach, or an *agglomerative* (bottom-up) approaches, have an inherent sequential control flow, with scope for parallelism higher in the early stages than the later. Further, parallel graph partitioning algorithms show scalability with thread/core count increase (*strong scalability*) and data increase (*data scalability*), but they do not fit the community detection framework. Further, due to high computational overhead of typical community detection algorithms such as those involving spectral clustering [18] one cannot use them for massive graphs with 100s of millions of edges. In addition to this, the irregular structure of the graphs leads to poor cache performance on multi-core architectures and synchronization overheads at the level of nodes or edges can be detrimental to the performance and scalability as well. Thus, parallel and scalable community detection for massive graphs with high quality (modularity) is an extremely challenging problem.

Current and emerging multi-core / many-core architectures, such as *Power 6*, *Power 7* and *GPGPUs* have large number of threads and cores and provide an opportunity to design new scalable parallel community detection algorithms with fine-grained parallelism. For GPGPUs, the algorithm has to designed carefully to have low *thread divergence* and good computation-communication overlap to hide memory access latencies. In this paper, we present the design of a novel scalable parallel algorithm, to find communities in massive graphs. We have designed a variant of the *label propagation* technique that scales with number of cores/threads and also delivers high modularity for well known and large sized RMAT graphs. Our algorithm tries to find communities such that the number of loops within each community is maximized. The time complexity of our algorithm is near linear to the number of edges, $O(m.(k+d))$, where k is the number of iterations (typically a small constant around 10 for massive graphs) and d is the average degree of a node in the graph. This is superior to the typical community detection algorithms that have $O(mn)$ time complexity where, n is the number of nodes in the graph. Our algorithm is designed to handle both undirected and directed weighted graphs. We have optimized our parallel community detection algorithm for both multi-core architectures such as Power 6 as well the GPGPU architecture.

The main contributions of this paper are as follows:

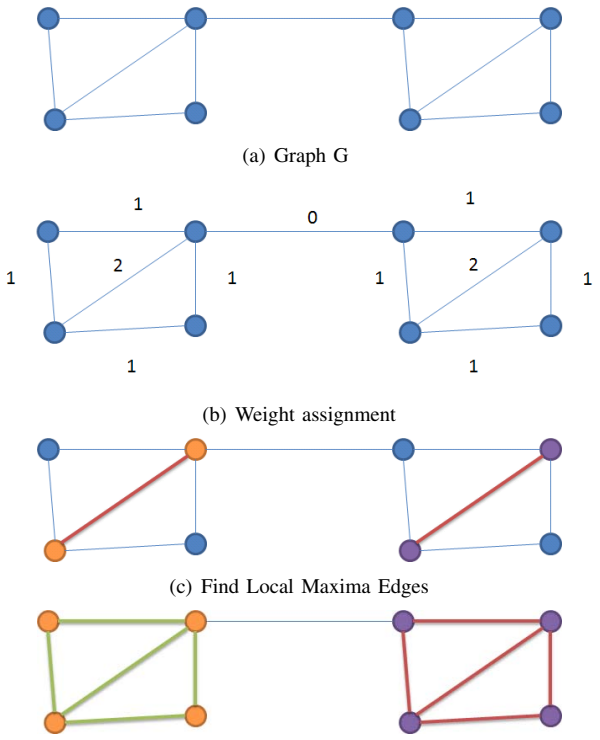
- We have designed a novel parallel scalable algorithm for community detection over massive (100M edges) graphs. Our algorithm has near linear time complexity ($O(m)$) and has very low synchronization overhead. Our algorithm applies to general weighted directed graphs and also detects overlapping communities.
- The algorithm has been optimized for multi-core architectures (such as Power 6). The quality (modularity) achieved by our algorithm is close to the best achieved by prior approaches. Simultaneously, our algorithm outperforms prior approaches by one to two orders of magnitude in performance on real-world graphs such as *hep-th*, showing collaborations between authors of papers submitted to High Energy Physics and *wikipedia* linkage graph. Our algorithm demonstrates strong, weak and data scalability on large R-MAT graphs with $40M$ edges and also real-world graphs such as *wikipedia*.
- We have also optimized our parallel algorithm for GPGPU architectures. Experimental results on massive graphs including RMAT graphs with 20M to 40M edges demonstrate the superior performance and scalability of our GPGPU based algorithm as compared to all the best known parallel community detection algorithms. This is the first work on parallel community detection on GPGPUs. Also, to the best of our knowledge these are the best demonstrated results in terms of performance vs. quality trade-off for parallel community detection over massive graphs.

The paper is organized as follows, Section 2 presents related work for community detection. Section 3 presents the background for label propagation algorithm and GPGPU architectures. Sections 4 & 5 present the design our parallel algorithm for multi-core architectures and GPUs. Section 6 presents the experimental results on real graphs as well as synthetic graphs. Section 7 presents conclusions and future work.

II. Related work

Community detection is a well studied research area but achieving strong scalability along with high quality of the generated communities remains an open problem. The first algorithm to find communities from a graph efficiently was the betweenness centrality based algorithm by Girvan and Newman[14]. The run time of this algorithm is $O(m^2n)$. Since then, there have been a large number of algorithms to solve the problem,the major categories in which these algorithms can be classified are as follows: Hierarchical clustering, Partitional clustering, and Spectral clustering

Hierarchical clustering has been the method of choice for most algorithms, Here, a *dendogram* of partitions of the graph is made. Each node in the dendogram represents a given modularity increase over the nodes below. Such a tree is used to find the best modularity partitions. Examples of algorithms that use such a method include the Fast greedy algorithm[20], random walks by Pascal Pons [22]



(a) Graph G

(b) Weight assignment

(c) Find Local Maxima Edges

(d) Weighted Label Propagation

Figure 1. Our algorithm to find communities using weighted label propagation

and betweenness centrality based community detection of Girvan and Newman[14]. Radicchi $et.al.$ [23] presented a local algorithm, which runs in $O(n^2)$ time. This algorithm assigns weight to each edge in the graph by computing its *edge clustering co-efficient*. Edge clustering co-efficient is defined as the ratio of the number of loops the edge actually participates in the graph to the maximum number of possible loops the edge could participate in, given the degrees of the nodes at the endpoints of that edge. The edges which are central to a network will tend to have a large value of edge clustering coefficient. The edges with the lowest weights are deleted, as they are considered to be connecting two communities (inter-community edges). The edge clustering coefficient is a locally normalized parameter but it does not capture the importance of an edge to a node succinctly. We use a near linear order community detection algorithm using weighted label propagation, where weights in the directed edges represent how the labels should propagate for the endpoint vertices to have the correct community labels.

Newman proposed a greedy algorithm [5] to find communities in a large graph, named as Fast Greedy method. Here, pairs of nodes which cause the highest rise in the modularity value of the graph are merged together. The algorithm of Pascal Pons claims a fair tradeoff between performance and time complexity. We used label propagation based algorithm that iteratively refines the community structure and achieves near linear time complexity, $O(m)$. This is one of the best reported so far in the literature. A comparison of the time complexity of well known community detection algorithms is given in the table I.

Eckmann and Moses [9]	$O(mk^2)$
Zhou and Lipowsky [28]	$O(n^3)$
Latapy and Pons [22]	$O(n^3)$
Newman [20]	$O(nlog^2n)$
Newman and Girvan [19]	$O(m^2n)$
Girvan and Newman [14]	$O(n^2m)$
Duch and Arenas [8]	$O(n^2logn)$
Fortunato et al. [11]	$O(n^4)$
Radicchi et al. [23]	$O(n^2)$
Donetti and Munoz [7]	$O(n^3)$
Palla et al. [21]	$O(exp(n))$
Zhang et al. [27]	$O(m)$
Raghavan et al. [24]	$O(m)$
Our Method	$O(m)$

Table I
COMPARISON OF TIME COMPLEXITY OF POPULAR COMMUNITY DETECTION ALGORITHMS

A comprehensive list of the such work can be found in the survey paper [10]. The algorithms presented in this survey are designed for sequential computing, though there is inherent parallelism in many algorithms.

Community detection algorithms have another classification based on the definition of community: local vs global. Some algorithms use a local definition, wherein local measures are sufficient to find the community of the algorithm. Global definitions on the contrary use a definition that requires global information of the topology of the graph. Hence, local algorithms make decisions purely based on the limited topological knowledge around the node. Global algorithms on the contrary make decisions based on the global topology. An example of local algorithms is the label propagation algorithm [24], global algorithms include betweenness centrality[3], [12]. Time efficient parallel algorithms with low communication overheads can be achieved by using algorithms that are local in nature. An algorithm presented for parallel community detection using propinquity dynamics [27] employs only local information based decisions. We employ primarily local topology information in computing the communities but also use global objective function to achieve high modularity.

Dynamic algorithms which change the graph structure, producing iterative densification of the graph have shown interesting results. Zhang $et.al.$ [27] presented such a dynamic algorithm. Their technique has a large search space per iteration and the number of comparisons in first iteration is of the order of $O(d^2)$. Further, this approach might keep adding edges and making the graph denser even when that community has been clearly identified which leads to high overall compute time.

The label propagation algorithm presented by [24] is a near linear time algorithm, which finds communities by iteratively spreading labels across a graph. Each node picks that label in its neighborhood that has the maximum frequency. The labels are allowed to spread synchronously or asynchronously till the system reaches a near stable state. Label propagation has been shown to be equivalent to a Pott's model approach [25]. Another body of work [15] presented that label propagation model has limitations, such as epidemic spread of labels especially for large communities, which overshadow smaller communities. Also, the results are variable with each experiment instance, especially in the asynchronous mode of label propagation. A method based on hop attenuation was presented there to mitigate the issue. We have designed a variant of the label propagation algorithm for parallel community detection that prevents oscillations in labels by proactively identifying such points of oscillation in the graph. This helps our algorithm in converging quickly in very few iterations even for massive graphs with 100s of millions of edges. Further, our algorithm uses global objective function to prevent the epidemic spread of labels and also assigns weights to the edges based on a combination of local topology of its neighbourhood and the prior weight specified on the input graph. Our algorithm has been optimized for both multi-core and GPGPU architectures and shows superior performance compared to all known community detection algorithms.

III. Background and Notation

In this section we provide an overview of the GPGPU architecture as well as the Label Propagation algorithm. Next, we provide notations used in the paper including definition of modularity and the Potts model.

A. GPU Computational model

The GPU is a massively multi-threaded architecture containing hundreds of processing elements (cores). The Fermi GF100 has 16 Streaming Multiprocessors (*SM*) with 32 SIMD cores per SM. The cores inside an SM execute the same instruction in a SIMD fashion. Thus, a Fermi architecture GPU, consists of total 512 cores. It can usually complete 512 32-bit integer or 32-bit floating-point operations per 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 12 GB/second. This architecture includes support for Simultaneous Multithreading in addition to the fine-grained multi-threading found in previous GPUs, which can improve hardware utilization. Fermi has a configurable 64KBprivate user manageable memory with every Streaming Multiprocessor(SM) and a 768 KB shared second-level cache. In addition 128 KB ($16 * 128$) of scratch pad memory is available per SM chip as the register file.

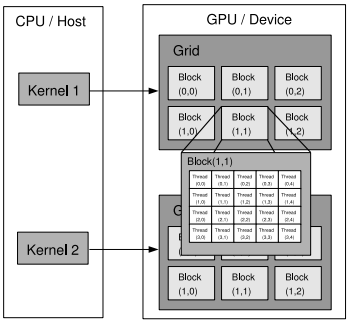


Figure 2. Execution model of the GPU

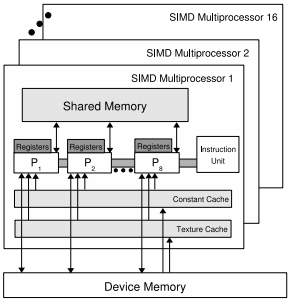


Figure 3. Computational model of the GPU

The CUDA API (Fig.4) allows a user to create a hierarchy of large number of threads to execute on the GPU. Threads are grouped into blocks and blocks make up a grid. Blocks are serially assigned for execution 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 allows for zero overhead scheduling, which enables warps that are stalled on a memory fetches to be swapped for another warp with immediately available computation. For this purpose, NVIDIA recommends at least 512 threads be assigned to an SM to keep an SM fully occupied.

Computations that are to be performed on the GPU are specified in the code as explicit kernels. Prior to launching the kernel, all the data required for the computation must be transfered from the Host (CPU) memory to the GPU (Global) memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronization for all the 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.

B. Label Propagation

The Label propagation algorithm was proposed by Raghavan [24]. In label propagation, each node is initialized with a

unique label, a label is equivalent to a community. Hence, the graph is initialized with the same number of communities as the number of nodes. In each iteration of the algorithm, each node tries to attach itself to the *majority* label in its vicinity. The algorithm repeats till the label propagation algorithm reaches a state of equilibrium, i.e. when no more variation in the labels of a node takes place. It has been shown that label propagation is equivalent to a Pott's [25] model, wherein the algorithm inherently tries to maximize the function:

$$Quality = \Sigma A_{i,j}\delta_{(C_i,C_j)} \tag{1}$$

where, $A(i,j)$ denotes edge weight between nodes i and j. $\delta(i,j)$ is the Kronecker delta function ($\delta(i,j) = 1$, *if* $i = j$; *else* $\delta(i,j) = 0$). C_i and C_j denote the communities to which node i and j belong.

Based on this quality function, some important observations can be made. The global maxima of the function occurs, when all the nodes have the same label. Another feature of the quality function is the lack of quality maximization in the global context. A label which is spread over a large number of vertices is referred to as a dominant label. Due to the lack of a global criteria (to block their spread), dominant communities increase their size over iterations. This leads to the spread of labels which are spread over a large number of nodes to spread further. As in all community detection algorithms for massive graphs, label propagation also focuses on finding local maximas. One key strength of the label propagation is that it is inherently parallel in nature, is suitable for fine-grained multi-threading and obtains scalable performance on multi-core / many-core architectures.

C. Definitions & Notations

The premise of the modularity optimization method is that a good division of a network into communities will give high values of the benefit function Q, called the modularity, intuitively defined as

$$Q = (E_{in}) - (\text{Expected number of such edges}) \tag{2}$$

Here E_{in} is the number of intra community edges. Large positive values of the modularity indicate when a statistically surprising fraction of the edges in a network fall within the chosen communities; it tells us when 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 so-called the configuration model, a random graph conditioned on the degree sequence of the original network, in which the probability of an edge between two vertices i and j is $k_i k_j/2m$, where k_i is the degree of vertex i and m is the total number of edges in the network. The modularity can then be written as:

$$Q = \frac{1}{2m}\sum_{i,j}[A_{(i,j)} - \frac{k_i k_j}{2m}] \cdot \delta_{(C_i,C_j)} \tag{3}$$

where, $A_{(i,j)}$ is an element of the adjacency matrix, $\delta_{(i,j)}$ is the Kronecker delta symbol, and C_i is the label of the community to which vertex i is assigned. Then one maximizes Q over possible divisions of the network into communities, the maximum being taken as the best estimate of the true communities in the network. Neither the size nor the number of communities need be fixed; both can be varied freely in the attempt to find the maximum. Modularity measure comes under the general framework of the Pott's model [25]. The Pott's model based equation is as shown below:

$$Q = \frac{1}{2M}\Sigma(A_{(i,j)} - \gamma p_{(i,j)})\delta(c_i, c_j) \tag{4}$$

Here, A is the adjacency matrix of the graph, $p_{(i,j)}$ is the probability of having an edge between nodes i and j. The equation tries to compare the true distribution of links in the graph against a null model. The modularity function is based on a popular null model which tries to compare the distribution of the links against an equivalent random graph, having similar degree distribution. This model was suggested by Newman. Newman proposed that:

$$p_{(i,j)} = \frac{d_i . d_j}{2M} \tag{5}$$

It can be seen that higher the modularity, higher the variation, hence the better the partition.

IV. Community Detection Algorithm Design

In this section, we present the design of our community detection algorithm based on weighted label propagation. In the following sections we present the key design features of our algorithm that lead to not only fast execution time but also high modularity.

A. Weight Assignment & Propagation Function

Consider an unweighted directed or undirected graph. We will extend our framework to weighted graphs as well later. For label propagation, the assignment of weights to edges determines how the labels propagate through the network resulting in the community structure generated by the algorithm. Thus, edge weights that implicitly represent accurate topological structure of the inherent communities in the network are desirable. However, such prior knowledge of the inherent communities is not available. To address this challenge, we consider the weight of an edge as a measure of the importance of that edge to the nodes at the endpoints of that edge. In case of an undirected graph, each edge is replaced by two directed edges.

The weight of a directed edge, $e = (i,j)$ (from vertex i to vertex j), is defined as the ratio of the number of triangles that the edge participates in, to the total number of triangles the node, i, participates in. For an edge $e = (i,j)$, let edge e represent the highest weighted edge in the locality of i, then i has a higher chance of being assigned the same label as j, as compared to any other label in the vicinity. The directed

edges with large weights correspond to connections that have a stronger importance to a node. Also, the edges with low weights represent weak relations, hence the chance of both nodes being in the same community is lower. Hence, weight of an edge $e = (i, j)$ is given by:

$$w_t(i,j) = \frac{z_{(i,j)}}{\Sigma_{(i,k)}(z_{(i,k)})}; k \in N(i) \tag{6}$$

where, $z_{(i,j)}$, represents the number of triangles with edge (i, j) as one of the edges in the triangle.

In case of a weighted graph given as input, we take the product of the given weight and topological loops based weight mentioned above. Thus, if an edge $e = (i, j)$ has weight given by the user as $w_u(i, j)$, then the weight of the edge considered for our label propagation algorithm is:

$$w(i,j) = w_u(i,j) * w_t(i,j) \tag{7}$$

The propagation function to transfer labels from one node to another is then defined as:

$$L(i) = argmax\Sigma_{j \in N_i} s(L_j) \tag{8}$$

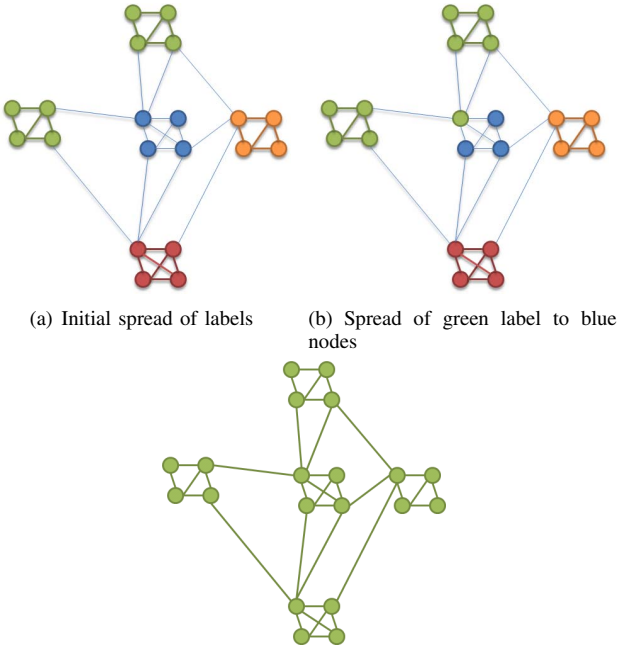
where, N_i is the set of neighboring vertices of vertex i; $s(L_j)$ is the total weight for the label $L_(j)$ in the neighborhood of vertex i.

B. Core edge detection

For a given weighted graph, for each node i there exists node j^* such that for node i , edge (i, j^*) has the maximum weight in its neighborhood. There will exist node pairs (v_1, v_2) such that v_1 is paired to v_2 using the maximum edge weight criterion and also conversely, v_2 is paired to v_1 using the maximum edge weight criterion. One can see that using the propagation function defined by the equation (8), the labels on two such nodes within a pair can oscillate without ever converging. The oscillatory behavior weakens community detection, as meaningful communities are not formed. This will lead to low modularity output as well as higher number of iterations in the algorithm. Such node pairs forms a local maxima and have the tendency to form the cores of communities. This oscillation problem needs to be addressed meaningfully. Labeling such local maxima pairs with the same label will improve the qualitative performance of the algorithm as well as the overall running time. Hence, we find such pairs *before* the label propagation iterations, and the same label is given to both the nodes in each pair. An extension of this issue is the presence of multiple overlapping pairs, where a single node can form such pairs with multiple nodes. Such overlapping nodes represent local communities in the graph. Hence, such pairs should be part of the same community. In our algorithm, we find the connected components over such overlapping pairs, and assign the same label to all the nodes within each component.

C. Epidemic spread Control

Label propagation algorithm has a natural global minima when all the nodes in the graph have the same label. This is caused by a large community dominating over all the other communities. In disease networks, this can be seen as the spread of an epidemic [17]. Hence, we call this phenomenon as epidemic spread. Though, the presence of weak edges between communities can reduce the epidemic spread to a large extent, in graphs with relatively low variation in edge density, the algorithm can still be susceptible to epidemic spread. The Figure 4 shows an example of disease spread over a graph. To tackle epidemic spread, we present here methods that work at node level and as well as use statistics of the spread of the labels in the graph.



(a) Initial spread of labels

(b) Spread of green label to blue nodes

(c) Final stable state after epidemic spread of labels

Figure 4. Spread of disease on a network

At node level, the duration over which a node is attached to a given label can be seen as the momentum of that label for the given node. For each node v, if the node label has not changed in the last k iterations, then the node has a strong correspondence with that label. Any change can thus suggest a variation in the neighborhood of v. To strengthen this label further and hence to improve the tolerance of network components to epidemic spread, a self loop is added to each node. It can be noted that the presence of self loops prevents the core of a large community from associating to a smaller label. The weight function of the self loop edges is given by:

$$W(i,i) = k \tag{9}$$

Here, k is a user defined parameter.

Another technique of improving the epidemic resistance is to control the size of a community. We assign a weight to each label based on the total degree of the nodes that have that label. Thus, the weight of a label is given by:

$$W_l(L_i) = 1 - \frac{d_c}{2M} \tag{10}$$

where, L_c is the label of community c; d_c 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 becomes:

$$L(i) = argmax\Sigma_{j \in N_i}[s(L_j) * W_l(L_j)] \tag{11}$$

where, N_i is the set of neighboring vertices of vertex i; $s(L_j)$ is the total weight for the label $L_(j)$ in the neighborhood of vertex i. As the size of a community (number of nodes with same label) increases, the weight of that label decreases. Thus, the ability of a label to propagate reduces with its size. The weight attached with each label thus acts as a global objective function and helps in controlling the size of the communities. The quality objective function which our algorithm tries to maximize is,

$$Quality = \Sigma_{(i,j)} S_{i,j} \cdot A_{i,j} \cdot \delta_{(C_i, C_j)} \tag{12}$$

Here, $S_{i,j}$ is the dynamic weight of the edge (i, j), which is dependent on the size of the community of which j is a member and the initial weight assigned to edge (i, j). This effectively can be reduced to the following equation:

$$Q = \frac{1}{2m}[\sum_{i,j}[A_{(i,j)}\delta_(C_i, C_j)] - \sum_i \frac{din_i d_i}{2m}] \tag{13}$$

Here, din_i represents the in degree of Community C_i at node i. d_i represents the degree of node i. It is notable that the quality measure of our method has similarities to the Pott's model. In practice, this means that the global maxima of our method does not lie in a single community.

D. Overlapping Community extraction

We can also find overlapping communities using our algorithm, by a post processing step. A node is part of multiple communities, if the edge weight of all the intra community edges for that node is less than half of the total sum. Thus, the connectivity of the node is strong to more than one community. Thus, the node can be considered as an overlapping node. The degree of overlap of a node with the communities depends on the local topological neighborhood around that node. Thus, in a linear time pass over the nodes in the graph and their neighboring edges, one can determine which nodes belong to multiple communities (labels). This is very useful in real-world graphs where overlapping communities are very common.

It can be noted that our algorithm works well with weighted as well as directed graphs. The complete Algorithm is shown as Algorithm 1. It can be seen that our algorithm has near linear time complexity, $O(m.(k + d))$, where k is the number of iterations (typically a small constant around 10 for massive graphs) and d is the average degree of a node in the graph. Further, the space complexity is also linear, $O(m)$.

Input: Graph $G(V, E)$
Output: community of each node
foreach *Edge* $e(i, j)$ **do**
  Find weight of $e(i, j) = w(i, j)$
end
foreach *Node n* **do**
  community(n)=n
end
foreach *Node n* **do**
  Find Maximum weighted edge in adjacency list;
  Store weight in $weight(n)$
end
G'=ϕ;
foreach *Node n* **do**
  foreach *edge e(n,v)* **do**
    if *weight(v)=weight(n)* **then**
      Add edge (v,n) to G'
    end
  end
end
Find connected components in G';
foreach *Node n* **do**
  community(n)=smallest label in component containing n in G'
end
while *All nodes are not stably labeled* **do**
  foreach *Node n* **do**
    $community'_n =$
    $argmax\Sigma_{i' \epsilon N_n}(community_{i'}).w(i', n).W(L_{i'})$
  end
  Exchange community and community';
end

Algorithm 1: The weighted label propagation algorithm

V. Parallel Community Detection Algorithm Design

This section presents the design of our parallel community detection algorithm for multi-core architectures and also for GPGPU architectures.

A. Parallel Design for Multi-core Architectures

Our weighted label propagation algorithm is data parallel, has inherent fine-grained parallelism and requires minimal synchronization. Another important feature of our algorithm , is the lock free nature of the algorithm. Updating the

Input: Graph $G(V, E)$
Output: community of each node
foreach *Processor p* **do**
  Assign nodes and edges to each processor;
  foreach *Edge* $e(i, j)$ *assigned to p* **do**
    Find weight of $e(i, j) = w(i, j)$
  end
  `Synchronize processors;`
  foreach *Node n assigned to p* **do**
    community(n)=n
  end
  `Synchronize processors;`
  foreach *Node n assigned to p* **do**
    Find Maximum weighted edge in adjacency list;
    Store weight in $weight(n)$
  end
  `Synchronize processors;`
  G'_p=ϕ;
  foreach *Node n assigned to p* **do**
    foreach *edge e(n,v) assigned to p* **do**
      if *weight(v)=weight(n) assigned to p* **then**
        Add edge (v,n) to G'_p
      end
    end
  end
  `Synchronize processors;`
  Merge G'_p to G';
  `Synchronize processors;`
  Find connected components in G';
  foreach *Node n* **do**
    community(n)=smallest label in component containing n in G'
  end
  `Synchronize processors;`
  while *All nodes are not stably labeled* **do**
    foreach *Node n* **do**
      $community'_n =$
      $argmax\Sigma_{i' \epsilon N_n}(community_{i'}).\, w(i', n).W(L_{i'})$
    end
    `Synchronize processors;`
    Exchange community and community';
  end
end

Algorithm 2: The parallel version of weighted label propagation algorithm

similarity matrix of each edge is an embarrassingly parallel operation, where each edge operation is independent of other edge operations. We have performed connected components using boundary expansion based parallel BFS [13], to find the connected components in the graph.

For label propagation, the synchronous mode of operation provides stable results as well as reduces the synchronization overhead. Herein, each node updates its labels based on the labels in its neighborhood from the previous iteration. In an asynchronous mode of operation, the updated labels in the current iteration also get used for further updates in the same iteration. The asynchronous method can cause epidemic spread of labels and has high overhead synchronization. We use the synchronous label propagation technique, as it has low synchronization overhead in the label propagation stage, and the algorithm is well suited for parallel scalable performance.

B. Parallel Design for GPGPU Architecture

GPU amenable algorithms are typically data parallel in nature. The cache behavior of graph algorithm tends to be poor, hence GPU implementations have an inherent issue as such. As can be seen from the algorithm, the irregular reads in an iteration of the algorithm is restricted to reading the label information of the neighbors of a node. The memory latency is hidden by the computation involved in finding the dominant label in the vicinity of each node. To implement label propagation on to the GPU, we map each component of the algorithm into a known data parallel algorithm. Finding the label of maximum weight in the neighborhood of a node is converted to a segmented sort on an array. Finding the weight of each edge is embarrassingly parallel. Hence it is performed on the GPU itself. The algorithmic complexity of parallel connected components algorithm can change the lower bounds of the algorithm. Hence we have performed connected components using boundary expansion based parallel BFS on the CPU [13].

Due to scalability concerns on resource constrained GPUs, we have used Bitonic sort for finding the majority node in the neighborhood of a node. Bitonic sort is the top down partition of an array into two equal halves, such that elements in the first are smaller than the second for an ascending sort. In parallel bitonic sort[2], blocks of size $2^n, 0 < n < \log(N)$ are sorted, such that after one pass of $\log(N)$ iterations, the array is split into two halves, relatively sorted to each other. These blocks are then internally sorted to fully sort the array.

Bitonic sort is an inplace sort, hence no additional space is required for sorting. This makes it a relevant sorting technique for resource constrained massively multi-threaded architectures like the GPU. Bitonic sort has lower performance as compared to Radix based sorts, but is able to perform sorting without any auxiliary data structures. Hence, no additional space is required for Bitonic sort. Also, since

only segments of the array are to be sorted, we implement *segmented* variant of a bitonic sort. The typical size of the segments are highly variable. As the target graphs are social networks, a power law distribution of degrees can be expected. Hence the variability of the degree of the nodes is large.

Bitonic sort, though designed for arrays of length 2^n, can be naturally extended for any arbitrary sized arrays. Given an array A, Bitonic sort divides an array A into two parts A_1, A_2, Also, size of both subarrays is equal to half the size of A. The two subarrays are sorted, A_1 in ascending order, and A_2 in descending order. Then element wise swap is performed on the two arrays such that each element of A_1 is smaller than every element in A_2. The subarrays are then sorted recursively, in effect sorting A.

For arbitrary length arrays, let length of the two arrays A_1 and A_2 be $length(A_1) = length1$, $length(A_2) = length2$. We further assume that $length1 \geq length2$. A_1 is always in ascending order, and A_2 is always in descending order. Based on the principle of bitonic sorting, swapping elements of two equal sized arrays will cause one to have the larger elements as compared to the other. Hence, we need only swap the last $length2$ elements of A_1 with that of A_2. This is the basis of our sorting algorithm. In our method, we sort the neighborhood communities of each node. Hence, for each node we assign a block of thread per node, and let that block sort the segment related to the node.

As can be seen from the description above, we have managed non coalesced memory access by masking it with a computationally intensive compute phase. To optimize the performance of the algorithm, known optimized GPU algorithms such as sort and scan are used.

VI. Experimental Results & Analysis

In this section, we report the experimental setup and performance and quality results of our algorithm as compared to the best known previous work.

GPU Implementation The GPU implementation of the algorithm is as follows: Fermi supports 1024 threads per block. We have used 1024 threads per block. The number of assigned threads per node though is a smaller number. All of our implementations provide 128 threads for each node of the graph.

A. Experimental Setup

The multithreaded CPU experiments were run using a 32 2-way threaded IBM Power6 Shared Memory Processors with Linux operating system. The maximum memory usage our program was allowed was 4GB. The code was compiled with the GNU C Compiler(gcc) version 4.2.1. The programs were compiled with the *-O3* flag. Parallel implementation was done using the pthreads NPTL API (Native Posix Threads Library). The GPU code was ran on a Fermi based Tesla S2050. The code is written in C using the CUDA API. The CUDA SDK 3.1 is used here.

While collecting performance statistics of the algorithm, each experiment was run multiple times, and the average values are reported here. The quality of the communities generated is measured by the modularity function. Higher the modularity higher is the quality of the output and better is the community structure. The performance of our algorithm is compared against prior approaches on well known graphs. We also demonstrate the scalability of our algorithm using the following measures:

- *Weak scalability*: Weak scalability tries to capture scalability of the solution as the number of processors increase, the problem size per processor is kept constant. The weak scalability of our algorithm is computed using R-MAT graphs generated using the GTgraph library.
- *Strong scalability*: Strong scalability is the performance increase with increase in the number of processors, with the problem size constant. The scalability of the algorithm , with respect to increase in hardware resources is captured by this method. This evaluation tries to capture issues such as mutual independence of the results and efficient utilization of hardware resources. Here again, we have used R-MAT graphs.
- *Data scalability*: Data scalability refers to the time taken to run the algorithm for varying size of the data, keeping other parameters constant. Constant parameters include number of processing elements and thread count. Data scalability directly characterizes the complexity of the algorithm.

B. Experimental Results

As mentioned in the earlier section, we have tested our algorithm against synthetic as well as real world data sets.

1) Real World Small Datasets: The results of the runs of the algorithm on the real world data sets are presented in the table II. The real world datasets considered are Zachary Karate Club [26] (34 nodes, 78 edges), Football [14] (115 nodes, 1230 edges) and Dolphin [16] (62 nodes, 159 edges). In table II, GN is the Girvan-Newman Betweeness centrality based algorithm. WT is the walktrap algorithm, WT 5 represents that the walk length used is 5, and for WT 2, walk length is 2. WLP is our algorithm. It can be seen that our algorithm performs nearly equivalently against popular algorithms in terms of quality (modularity) of results. The reduction in the modularity of the results that the algorithm provides is marginal. Hence, our algorithm generates quality close to best known fast algorithms. Zachary karate club [26] is a dataset with partitions known from real life. The known partition of the graph is into 2 groups. The groups break down into $1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 17, 18, 20, 22$ and $9, 10, 15, 16, 19, 21, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34$.

Algorithm	Zachary Karate club	Football	Dolphin
GN	.41	.60	0.495
WT 5	.38	.60	.51
WT 2	.38	.60	.49
WLP	.40	.52	.48

Table II
COMPARISON OF MODULARITY OF THE ALGORITHM AGAINST KNOWN IMPLEMENTATIONS AND RESULTS

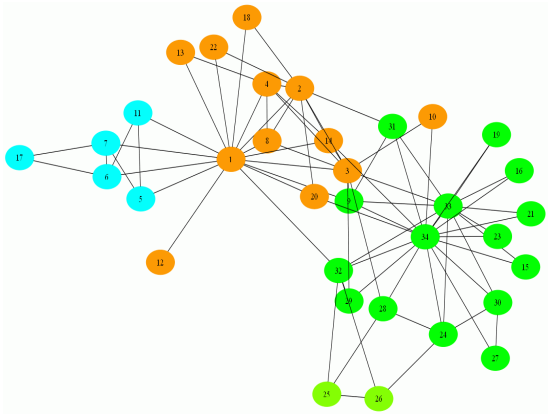


Figure 5. Result of the Zachary karate club graph. The combined orange and blue sections represent one of the groups that formed after the club split. All the nodes have been correctly identified for the two partition

Our algorithm divides one of the two partitions, hence forming three groups. It is notable that the modularity of the 3 partitioned graph is larger than the 2 partitioned result. Figure 5 presents our results. For the Dolphins interaction network [16], we were able to find the known partitions. Here again, one of the known partition is split into two by our algorithm. This lead to a higher modularity result. The resultant community is shown in Figure 6.

2) Large datasets: In this section, we present the quality and scalability results of our algorithm on datasets, and easily available large real world datasets. We could not compare

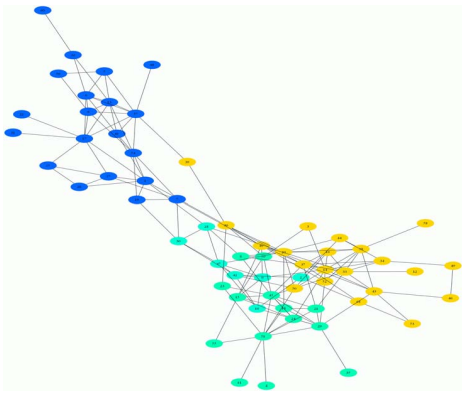


Figure 6. Result of the Dolphin social network

against algorithms such as *Walktrap* and *Fast modularity* due to the lack of data scalability of these algorithms. We compared our method against the data scalable algorithm presented by Zhang et.al. [27]. Zhang et.al. presented results against large datasets on a cluster of machines in the Google data-center. On the *hep-th* dataset ($27K$ nodes, $352K$ edges), our parallel algorithm takes only $0.27s$ using 32 cores, which is $370\times$ (more than *two orders of magnitude*) better than the algorithm by Zhang et.al. [27] ($100s$ using 50 machines each with 1GHz cpu and 1GB RAM). Simultaneously, we achieve modularity value of 0.58 which denotes high quality of the communities generated by our algorithm.

We benchmarked our algorithm using the wikipedia [27] real world dataset. The wikipedia graph is a directed graph with $2,491,887$ vertices and $117,714,397$ edges. The strong scalability curve for the wikipedia data is shown in the Figure 7. For wikipedia, the time decreases from $793s$ with 1 core/thread to $84s$ with 32 cores/threads. This gives a speedup of $9.44\times$ with $32\times$ increase in the total number of cores. Thus, the efficiency of the algorithm is around 30%. Zhang et.al. report parallel time of more than $1600s$ using 62 machines (1GHz CPU, 1 GB RAM per machine). Thus, as compared to Zhang et.al. we obtain more one than *one order of magnitude* ($19\times$)better performance. Using 1000 machines, Zhang et.al. report complete community detection time of $100s$ which is still around $2.4\times$ higher than the time taken by our parallel algorithm using only 32 Power 6 cores.

We also benchmarked our algorithm using massive random graphs generated using an R-MAT [4] graph generator available as part of the Gt-Graph library [1]. The strong scalability curve for R-MAT graphs are shown in Figure 7. Here, the number of edges is 20M and 40M respectively for the two R-MAT graphs. The number of nodes is 1M in both R-MAT graphs. For the R-MAT graph, with 20M edges and 1M nodes, the time decreases from $287s$ with 2 cores/threads to $23s$ with 50 threads. This gives a relative speedup of $12.5\times$ with $25\times$ increase in the number of threads. Thus, here the efficiency of the algorithm is 50%. For the R-MAT graph, with 40M edges and 1M nodes, the time decreases from $647s$ with 2 cores/threads to $60s$ with 50 threads. This gives a relative speedup of $10.8\times$ with $25\times$ increase in the number of threads. Thus, here the efficiency of the algorithm is 43%. The parallel efficiency is less than 50% in all these cases, as the community detection involves random access to the graph data in the memory. Thus, the cache performance is poor which causes the speedup to be limited with increasing number of nodes.

The data scalability plot is shown in Figure 8. It can be noted that the data scalability is near linear. The algorithm shows linear increase in time with the increase in the data. The weak scalability plot is shown in Figure 9, here each processor gets .1M nodes and .5 edges, and .2M nodes and 1M edges. The number of processors is varied from 1 till 32. With 32X increase in cores and corresponding increase in

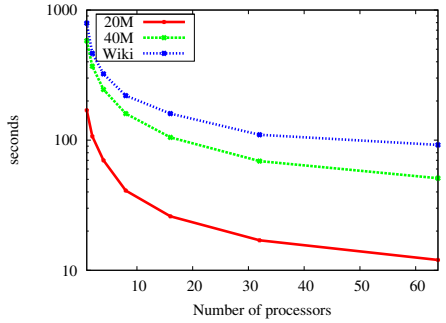


Figure 7. Strong Scalability plot, the number of vertices is 1M, the number of processors is varied

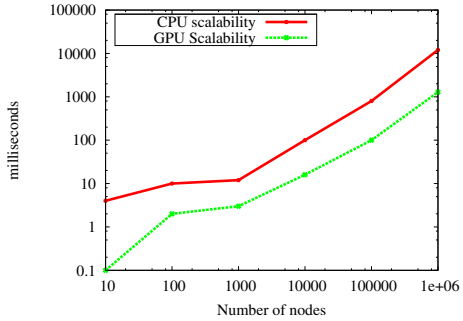


Figure 8. Data scalability plot, the degree of the graph is fixed at 40, the number of vertices is varied

data, the time varies within a 2X factor. Thus our algorithm demonstrates weak scalability.

C. GPU results

The comparison of performance of an algorithm on the GPU can only be done using the data scalability model of comparison. Hence we have also used the data scalability experiments presented in the previous section to benchmark the GPU code.

The data scalability plots of the algorithm on the GPU is shown in the figure 7. The scalability of the graph given

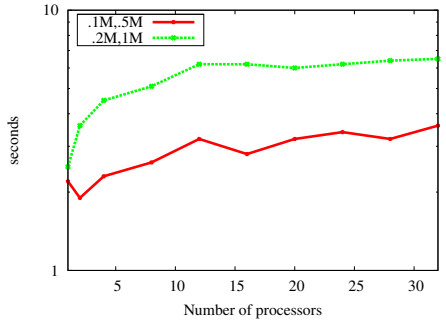


Figure 9. Weak scalability plot, the value of n and m for 1 processor is given, on increasing problem size, n and m also scale linearly

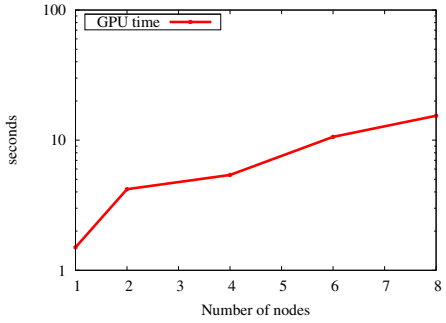


Figure 10. Scalability of our algorithm on the GPU for graphs with constant degree, number of nodes in millions

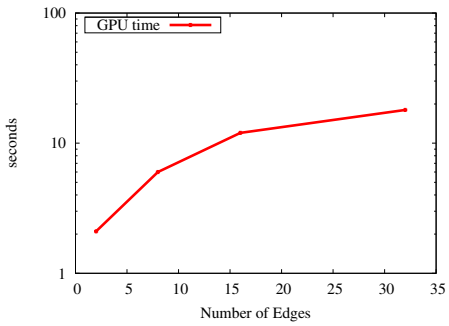


Figure 11. Scalability of our algorithm on the GPU for graphs with constant number of nodes, number of nodes in millions

constant degree is shown in figure 10. The scalability on the GPU with increasing degree of a node is shown in figure 11. The scalability of the algorithm is notable in both the figures. It can be seen that time increases linearly in each of the experiment.

For 1 M nodes and 20 M edges, the GPU took 15 seconds to run. Also, using 1M nodes, and 40 M edges, the GPU took 28 seconds to find communities in the graphs. A speedup of 8X against the Power6 processor is hence noted.

VII. Conclusion and Future Work

We have designed a novel parallel community detection algorithm based on weighted label propagation for multi-core architectures. Our algorithm is optimized for the GPGPU architecture. Our algorithm uses mostly local topological information to make decisions on community structure and has near linear time complexity, $O(m(k+d))$, where k is the number of iterations and d is average degree of a node in the graph. Extensive experimental analysis shows that our algorithm has superior performance over the best known prior techniques. For real-world graphs such as *hep-th* ($252K$ edges) we are more than two orders of magnitude better in performance compared to the best known parallel algorithms while for graphs such as *wikipedia* (100M edges) we are more than one order of

magnitude better in performance compared to best known results with similar number of CPUs/cores. On GPGPUs our algorithm demonstrates $8\times$ speedup over our own Power 6 performance for 40M R-MAT graph. Simultaneously, our approach achieves high modularity comparable to the best known techniques. The space consumption of our algorithm is also small, $O(m)$. Hence, we can conclude that this algorithm is designed for scalable performance on multi-core and hybrid architectures.

In future, we intend to look into mathematical models to predict the number of iterations for our algorithm. A tight bound on the quality of the algorithm, along with minimal tradeoff with the time is another promising future direction.

REFERENCES

[1] BADER, D., AND MADDURI, K. GTgraph: A synthetic graph generator suite. *Atlanta, GA, February* (2006).

[2] BATCHER, K. On bitonic sorting networks. *Parallel processing: proceedings* (1990).

[3] BRANDES, U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology 25*, 2 (2001), 163–177.

[4] CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. R-MAT: A recursive model for graph mining. In *SIAM Data Mining* (2004), vol. 6, Citeseer, pp. 6–1.

[5] CLAUSET, A., NEWMAN, M., AND MOORE, C. Finding community structure in very large networks. *Physical Review E 70*, 6 (2004), 66111.

[6] DANON, L., DÍAZ-GUILERA, A., DUCH, J., AND ARENAS, A. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment 2005* (2005), P09008.

[7] DONETTI, L., AND MUNOZ, M. Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment 2004* (2004), P10012.

[8] DUCH, J., AND ARENAS, A. Community detection in complex networks using extremal optimization. *Physical Review E 72*, 2 (2005), 27104.

[9] ECKMANN, J., AND MOSES, E. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proceedings of the national academy of sciences 99*, 9 (2002), 5825.

[10] FORTUNATO, S. Community detection in graphs. *Physics Reports* (2009).

[11] FORTUNATO, S., LATORA, V., AND MARCHIORI, M. Method to find community structures based on information centrality. *Physical Review E 70*, 5 (2004), 56104.

[12] FREEMAN, L. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[13] GHOSH, R., AND BHATTACHARJEE, G. Parallel breadth-first search algorithms for trees and graphs. *International Journal of Computer Mathematics 15*, 1 (1984), 255–268.

[14] GIRVAN, M., AND NEWMAN, M. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences 99*, 12 (2002), 7821.

[15] LEUNG, I. X. Y., HUI, P., LIÒ, P., AND CROWCROFT, J. Towards real-time community detection in large networks. *Physical Review E 79*, 6 (Jun 2009), 066107+.

[16] LUSSEAU, D. The emergent properties of a dolphin social network. *Proceedings of the Royal Society of London. Series B: Biological Sciences 270*, Suppl 2 (2003), S186.

[17] NEWMAN, M. Spread of epidemic disease on networks. *Physical Review E 66*, 1 (2002), 16128.

[18] NEWMAN, M. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E 74*, 3 (2006), 36104.

[19] NEWMAN, M., AND GIRVAN, M. Finding and evaluating community structure in networks. *Physical review E 69*, 2 (2004), 26113.

[20] NEWMAN, M. E. J. Fast algorithm for detecting community structure in networks. *Phys. Rev. E 69*, 6 (Jun 2004), 066133.

[21] PALLA, G., DERÉNYI, I., FARKAS, I., AND VICSEK, T. Uncovering the overlapping community structure of complex networks in nature and society. *Nature 435*, 7043 (2005), 814–818.

[22] PONS, P., AND LATAPY, M. Computing communities in large networks using random walks. *Computer and Information Sciences-ISCIS 2005* (2005), 284–293.

[23] RADICCHI, F., CASTELLANO, C., CECCONI, F., LORETO, V., AND PARISI, D. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences 101*, 9 (2004), 2658.

[24] RAGHAVAN, U., ALBERT, R., AND KUMARA, S. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E 76*, 3 (2007), 36106.

[25] TIBÉLY, G., AND KERTÉSZ, J. On the equivalence of the label propagation method of community detection and a Potts model approach. *Physica A: Statistical Mechanics and its Applications 387*, 19-20 (2008), 4982–4984.

[26] ZACHARY, W. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research 33*, 4 (1977), 452–473.

[27] ZHANG, Y., WANG, J., WANG, Y., AND ZHOU, L. Parallel community detection on large networks with propinquity dynamics. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (2009), ACM, pp. 997–1006.

[28] ZHOU, H., AND LIPOWSKY, R. Network Brownian motion: A new method to measure vertex-vertex proximity and to identify communities and subcommunities. *Computational Science-ICCS 2004* (2004), 1062–1069.