How to Partition a Billion-Node Graph

ABSTRACT

Billion-node graphs pose significant challenges at all levels from storage infrastructures to programming models. It is critical to develop a general purpose platform for graph processing. A distributed memory system is considered a feasible platform supporting online query processing as well as offline graph analytics. In this paper, we study the problem of partitioning a billion-node graph on such a platform, an important consideration because it has direct impact on load balancing and communication overhead. It is challenging not just because the graph is large, but because we can no longer assume that the data can be organized in arbitrary ways to maximize the performance of the partitioning algorithm. Instead, the algorithm must adopt the same data and programming model adopted by the system and other applications. In this paper, we propose a multi-level label propagation (MLP) method for graph partitioning. Experimental results show that our solution can partition billion-node graphs within several hours on a distributed memory system consisting of merely several machines, and the quality of the partitions produced by our approach is comparable to state-of-the-art approaches applied on toy-size graphs.

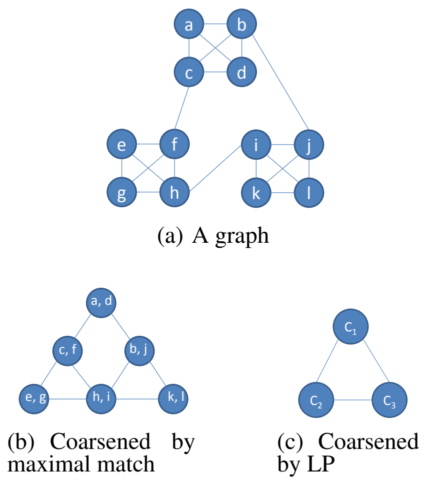
1. INTRODUCTION

Many large graphs have emerged in recent years. The most well known graph is the WWW, which now contains more than 50 billion web pages and more than one trillion unique URLs [1]. A recent snapshot of the friendship network of Facebook contains 800 million nodes and over 100 billion links [2]. LinkedData is also going through exponential growth, and it now consists of 31 billion RDF triples and 504 million RDF links [3]. In biology, the genome assembly problem has been converted into a problem of constructing, simplifying, and traversing the de Brujin graph of the read sequence [4]. Each vertex in the de Brujin graph represents a k-mer, and the entire graph in the worst can contain as many as 4^k vertices, where k generally is at least 20.

We are facing challenges at all levels from system infrastructures to programming models for managing and analyzing large graphs. We argue that a distributed memory system has the potential to meet both the memory and computation requirements for large graph processing. One of the biggest challenges is how to partition a graph so that it can be deployed on a distributed system. In this paper, we propose a general-purpose, scalable, and semantic (community-aware) approach to partition web-scale graphs.

1.1 Distributed Graphs

A distributed memory system is a suitable infrastructure for online query processing over billion node graphs [17]. The reason is twofold. First, as we explore a graph, we often invoke random, instead of sequential, data accesses, no matter how the graph is stored. To address this problem, a simple solution is to put the graph in the main memory. Second, although the topology of a billion node graph may not be prohibitively large, it is still unlikely that it can be stored in the memory of a single machine. Thus, a distributed system is necessary. A distributed system is also beneficial as graph analytics is often computation intensive. Using the memory and the computation power of all the machines, we may be able to operate on graphs of any size.



(a) A graph

(b) Coarsened by maximal match

(c) Coarsened by LP

Figure 1: An example graph and its coarse-grained graph

To deploy a graph on a distributed memory system, we need to divide the graph into multiple partitions, and store each partition in one machine[1]. Network communication is required for accessing non-local partitions of the graph. Thus, how the graph is partitioned may cause significant impact on load balancing and communication. Consider performing a BFS on a graph, which needs to access each edge of the graph. Whenever an edge crosses machine boundaries, we need to send and receive a network message. The cost of the BFS largely depends on how many network messages are needed. As an example, Figure 1(a) shows that different partitionings may lead to different communication overheads in distributed systems. Assuming we have 3 machines, and

[1]In this work, we assume no overlap between any two partitions.

each can hold at most 4 vertices and if we partition the graph as $\{a, b, c, d\}, \{e, f, g, h\}, \{h, i, j, k\}$, we need 3 network communications for a BFS. However, we end up requiring 17 remote accesses for a BFS if we partition it into $\{c, d, i, f\}, \{a, e, j, k\}, \{b, g, h, l\}$.

1.2 Existing Graph Partitioning Methods

The graph partitioning problem has been studied extensively in many application areas (e.g., VLSI design). The problem of finding an optimal partition is NP-Complete [23]. As a result, many approximate solutions have been proposed [5, 6]. However, as we show below, none of the existing solutions are capable of partitioning web-scale graphs on distributed memory systems.

Scalability. Most current graph partitioning algorithms are for small, memory-based graphs. A class of local refinement algorithms, most of which originated from the Kerninghan-Lin (KL) algorithm [5], bisect a graph into even size partitions. The KL algorithm incrementally swaps vertices among partitions of a bisection to reduce the edge-cut of the partitioning, until the partitioning reaches a local minimum. There are many variations based on the KL algorithm, including the FM algorithm [6]. The local refinement algorithms are costly, and are designed for memory-based graphs only.

Recently, several multi-level partitioning algorithms have been proposed [7, 8, 10]. The idea is to "coarsen" a large graph into a small graph and apply algorithms such as KL and FM on the small graph. However, as we will discuss in more detail in Section 2.2, the assumption that (near) optimal partitions on coarsened graphs implies a good partitioning in the original graph may not be valid for real life, scale-free graphs. Furthermore, the coarsening algorithm (maximal matching) is very costly, and does not scale on billion-node graphs.

To improve the scalability, some parallel partitioning solutions have been proposed, including ParMetis [11] and PT-Scotch [12]. Still, they cannot scale to billion-node graphs without significant improvement. For example, ParMetis uses maximal match to coarsen a large graph. To find the maximal match, it needs to perform random accesses. This aspect limits its extension on disk resident large graphs with billions of nodes. In fact, the largest graph reported by these approaches only has 23M nodes [12].

Generality. It is important to develop a general purpose infrastructure where graphs can be stored, served, and analyzed, especially for web-scale graphs. Current partitioning methods are not built on top of a general-purpose graph infrastructure. Instead, they are designed exclusively for the purpose of partitioning. Hence, they assume that the data can be organized or manipulated in ways that maximize the performance of the partitioning algorithm. To partition an existing billion-node graph stored in a general-purpose graph system, we must take the data out of the system, convert it into a partition friendly format, and after partitioning convert it back to the format on the system. In general, previous solutions are not conducive to a general-purpose graph infrastructure. For example, before ParMetis [11] can work, it requires that the graph is partitioned *two-dimensionally*, that is, the adjacency list of a single vertex is divided and stored in multiple machines. This helps reduce the communication overhead when coarsening a graph. However, such a design may be disruptive to other graph algorithms. Even if we adopt this approach, the cost of converting data back and forth is often prohibitive for web-scale graphs.

Then, the question is, is any infrastructure currently available appropriate for web-scale graphs? MapReduce is an effective paradigm for large-scale data processing. However, MapReduce is not the best choice for graph applications [13, 14]. Besides the fact that it does not support online graph query processing, many graph algorithms for offline analytics cannot be expressed naturally and intuitively. Instead, they need a total rethinking in the "MapReduce language." For example, graph exploration, i.e., following links from one vertex to its neighbors, is implemented by MapReduce iterations. Each iteration requires large amount of disk space and network I/O, which is exacerbated by the random access pattern of graph algorithms. Furthermore, the irregular structures of the graph often lead to varying degrees of parallelism over the course of execution, and overall, parallelism is poorly exploited [13, 14]. Only algorithms such as PageRank, shortest path discovery that can be implemented in vertex-centric processing and run in a fixed number of iterations can achieve good efficiency. The Pregel system [15] introduced a vertex-centric framework to support such algorithms. However, graph partitioning is still a big challenge. We are not aware of any effective graph partitioning algorithms in MapReduce or even in the vertex-centric framework.

Semantics. Real life graphs are not random or regular. Social networks and WWW are well-known for their irregularity and complex structures. Most partitioning algorithms ignore the complex structures. Many of them are designed for relatively regular graphs (such as meshes) that have generally uniform degree distribution. Some recent approaches take into consideration the power-law degree distribution exhibited by many real life networks [16]. However, many other features in complex networks such as *small-world*, *community structure* are not given enough considerations. Thus, whether existing state-of-the-art approaches work well on real life, complex, and web-scale networks remains an open problem. In our approach, we take the community structure of real networks into consideration when we partition the graphs.

1.3 New Challenges

Motivated by the above facts, *we have developed a scalable and semantic-aware graph partitioning solution on a general-purpose distributed memory system.* Our goal is to partition web-scale real graphs. To do so, we must address challenges introduced by real-life large graphs:

- The irregular structure of real graphs leads to *poor parallelism.* In general, the logic of a partitioning algorithm is complex and many computation steps may depend heavily on each other, leading to poor parallelism.

- The skewed degree distribution (such as power-law) of real graphs generates *unbalanced load distribution* over different processors.

- The data access pattern on real graphs exhibits *poor locality.* In general, computation involved in partitioning a real complex graph, such as social network, shows poor locality. As a result, the data access pattern is hard to predict, which means message passing is hard to optimize.

To meet the above challenges, we introduce an *efficient* (both in time and space), *highly parallelized* graph partitioning solution with *small communication overhead.* We present a *multilevel label propagation* (MLP) framework and its optimized implementation on a typical distributed memory system. Experimental results show that our solution is efficient and effective. For example, our solution can partition a graph with 512M nodes and 6.5G edges within 4 hours on a distributed memory system consisting of merely 8 machines. The quality of the resulting partitions is comparable to that of the current best approach, METIS [8].

1.4 Paper Organization

The rest of the paper is organized as follows. Section 2 introduces the graph infrastructure, as well as some background information about graph coarsening and label propagation. In Section 3, we discuss measures for graph partitioning and give the estimation of random partitioning on synthetic graphs. In Section 4, we present the MLP algorithm. Section 5 presents a disk-based implementation of MLP. Section 6 presents experiment results, and Section 7 reviews related works. We conclude in Section 8.

2. BACKGROUND

In this section, we first introduce the Trinity infrastructure, which is used as a general-purpose computation platform for web scale graphs. Then, we introduce two techniques related to our approach for graph partitioning: graph coarsening and label propagation.

2.1 The Trinity Graph System

We use Trinity [17] as the infrastructure for handling web-scale graphs. Trinity is essentially a memory cloud created out of the RAM of multiple machines, and it offers a unified memory space for user programs. Most graph applications need efficient random data accesses on graphs, and Trinity's efficient in-memory graph exploration and bulk message passing mechanisms answered this need and enable it to handle large graphs.

Trinity supports very efficient memory-based graph exploration. In one experiment, we deployed a synthetic, power-law graph in a 15-machine cluster managed by Trinity. The graph has Facebook-like size and distribution (800 millions nodes, 100 billion edges, with each node having on average 130 edges). We found that exploring the entire 3-hop neighborhood of any node in the graph takes less than 100 milliseconds on average. In other words, Trinity is able to explore $130 + 130^2 + 130^3 \approx 2.2$ million edges in one tenth of a second.

Making the graph topology memory resident makes fast random graph access possible. On the other hand, some computation allows us to predict the access pattern on the graph. In this case, we can store the entire graph on the disk and schedule parts of the graph to be memory resident when they are needed for computation. This enables Trinity to handle extremely large graphs using a small number of machines, and enables small organizations that cannot afford a large memory cloud to perform large-scale computations on graphs. In this paper, we propose a graph partitioning algorithm that allows us to predict the access pattern. Thus, we can partition billion-node graphs even if the memory cloud is not big enough to hold the entire graph. Our experiments show that we can partition billion-node graphs with eight machines that each has 48G memory.

Trinity also provides an efficient bulk message passing mechanism. Using this mechanism, we can build an offline computation platform for web-scale graph analytics on Trinity. For instance, we can implement the Pregel-like [15] Bulk Synchronous Parallel (BSP) computation model. In this model, the programmer writes a vertex-based algorithm, and the system takes care of its parallel execution on all vertices. Trinity's bulk message passing mechanism allows for a high performance by BSP. In one experiment, using just 8 machines, one BSP iteration on a synthetic, power-law graph of 1 billion nodes and 13 billion edges takes less than 60 seconds.

The efficient graph exploration and bulk message passing mechanism of Trinity lays the foundation for developing our graph partitioning algorithm. Still, there are many challenges to devising graph partitioning algorithms for vertex-based computation. In this paper, we introduce a novel label propagation based algorithm for graph partitioning.

2.2 Graph Coarsening

Graph partitioning algorithms such as KL [5] and FM [6] are effective for small graphs. For a large graph, a widely adopted approach is to "coarsen" the graph until its size is small enough for KL or FM. The idea is known as multi-level graph partitioning, and a representative approach is METIS [8].

METIS works in three steps: (1) coarsening the graph; (2) partitioning the coarsened graph; (3) uncoarsening. In the 1st step, METIS coarsens a graph by finding the *maximal match*. A maximal match is a maximal set of edges where no two edges share a common vertex. After it finds a maximal match, it collapses the two ends of each edge into one node, and as a result, the graph is "coarsened." The coarsening step repeats until the graph is small enough. Then, in the 2nd step, it applies KL or FM directly on the small graph. In the third step, the partitions on the small graph are projected back to the finer graphs.

Before we discuss potential problems of coarsening for real life graphs, we first look at an example:

EXAMPLE 1 (MAXIMAL MATCH). *For the graph shown in Figure 1(a), the following edge set is a maximal match:*

$$\{(c, f), (e, g), (h, i), (k, l), (j, b), (a, d)\}$$

Figure 1(b) is the result of coarsening (obtained after collapsing the two ends of each edge in the maximal match).

The correctness of METIS is based on the following assumption: *A (near) optimal partitioning on a coarser graph implies a good partitioning in the finer graph.* However, in general, the assumption only holds true when the degree of nodes in the graph is bounded by a constant [9]. For example, 2D or 3D meshes are graphs where node degrees are bounded. However, for today's real life graphs, the assumption does not hold any more. It is well established that the degree distribution of real life networks are right-skewed, and there are many hub vertices with very large degrees. In other words, the degree is not bounded by a small constant, but is related to the size of the graph. As a result, a maximal match may fail to serve as a good coarsening scheme in graph partitioning. For example, the coarsened graph in Figure 1(b) no longer contains the clear structure of the original graph. Thus, partitions on the coarsened graph cannot be optimal for the original graph.

Furthermore, the process of coarsening by maximal match is inefficient for billion-node graphs. Two maximal match strategies are used in various versions of METIS: Random matching (RM) and Heavy Edge Matching (HEM). In RM, the vertices are visited in a random order. If a vertex u has not been matched yet, then one of its unmatched neighbors will be randomly selected and matched with u. HEM is similar to RM, except that it selects the unmatched neighbor v if edge (u, v) has the largest weight. As we can see, in the above mentioned approaches, vertices are matched in a random order. For disk resident graphs, random access leads to bad performance. In a multi-level framework, graphs generated at each level and the mappings between them are stored in memory. These intermediate results can be very large. For example, for LiveJournal[2], a real social network that contains more than four million vertices, METIS (using either RM or HEM) will consume more than 10G of memory. The heavy usage of memory makes the approach unfeasible for billion-node graphs.

2.3 Label Propagation

We propose a method for large scale graph partitioning based on the idea of *label propagation* (LP), which was originally proposed for community detection in social networks. A naive LP runs as follows. We first assign a unique label id to each vertex. Then, we update the vertex label iteratively. In each iteration, a vertex takes the label that is prevalent in its neighborhood as its own label. The process terminates when labels no longer change. Vertices that have the same label belong to the same partition.

There are two reasons we adopt label propagation for partitioning. First, the label propagation mechanism is lightweight. It does not generate big intermediary results, and it does not require sorting or indexing the data as in many current graph partitioning algorithms. This makes label propagation feasible for web scale graphs deployed on Trinity. With Trinity's efficient graph exploration and

[2]http://snap.stanford.edu/data/soc-LiveJournal1.html

message passing mechanism, label propagation can be implemented with ease. Indeed, pure label propagation can be implemented using the vertex-centric computation model in 2 lines of code. More specifically, label propagation has low complexity, as long as the number of iterations is bounded: Let $G(V, E)$ be a graph, where V is the set of vertices, and E is the set of edges. In each iteration, labels need to be propagated along all edges, which takes $\Theta(|E|)$ time. Thus, the time complexity is $O(t|E|)$, where t is the number of iterations. On real-life networks, label propagation tends to converge in a constant number of iterations. Thus, it runs in almost linear time.

Second, label propagation is "semantic-aware" as it is able to discover inherent community structures in real networks: Given the existence of local closely connected substructures, a label tends to propagate within such structures. Since most real-life networks demonstrate clear community structures, a partitioning algorithm based on label propagation may divide the graph into meaningful partitions. Compared to maximal match, LP is more semantic-aware and is a better coarsening scheme. We illustrate this in Example 2.

EXAMPLE 2 (COARSENING BY LP). *Using LP, we obtain a coarser graph shown in Figure 1(c) for the original graph shown in Figure 1(a). In Figure 1(c),* $C_1 = \{a, b, c, d\}$, $C_2 = \{e, f, g, h\}$ *and* $C_3 = \{i, j, k, l\}$. *In this coarsened graph, the community structure of the original graph is completely preserved.*

However, although label propagation is lightweight and can be implemented on web-scale graphs, there are many obstacles to using pure label propagation for graph partitioning:

- *Imbalance.* Label propagation is fundamentally a clustering approach instead of a partitioning approach. One critical requirement for graph partitioning is to produce balanced partitions. This is important for load distribution, and communication overhead minimization. The results of label propagation are determined by the network structure. For real life graphs, it often results in skewed distribution of the community size: It is very likely that we end up with a few very extremely large partitions and many tiny ones. This goes against the purpose of graph partitioning.

- *Efficiency.* Previous approaches using label propagation for community detection focused on the quality of the solution and had little consideration of efficiency. Processing web-scale graphs calls for efficient algorithms. How to accelerate label propagation without sacrificing partitioning quality is a challenging problem. For instance, if we can reduce the number of iterations, we can improve performance dramatically.

- *Parallelization.* Although the label propagation mechanism can be easily parallelized, how to design and implement an efficient message-passing framework in a distributed environment is still a challenging problem. Problems such as how to reduce the total number of communications and how to avoid the generation of space-costly intermediate results need to be carefully addressed.

- *Convergence.* Label propagation does not have a theoretic guarantee for convergence. It may get trapped in an oscillation state. Consider a complete bipartite graph $G = V_1 \cup V_2$, once all vertices in V_1 have the same label a and all vertices in V_2 have the same label b, in the following iterations, each vertex in V_1 will change its label to b and each vertex in V_2 will change its label to a. As a result, the iterative process will oscillate between the two labeling states. Another kind of oscillation occurs when a vertex has the same number of connections to more than one communities. Then, in each iteration, the vertex will randomly select a label.

3. PROBLEM AND BASELINE

We formalize the problem of graph partitioning. Then, we study the quality of random partitioning, which serves as a baseline for other partitioning algorithms.

3.1 Problem Definitions

First, we need to decide how to measure the goodness of a partitioning. A natural goodness measure is the size of *edge cut*, that is, edges whose two end points are in two different partitions. In general, we want to minimize the size of *edge cut*. Particularly, in a distributed memory system, navigating along such edges means performing remote accesses. Too many remote accesses bring costly communication overheads.

DEFINITION 1 (SIZE OF EDGE CUT[3]). *For a partitioning* $\mathcal{P}$ *on graph* $G(V, E)$, *the size of edge cut is* $ec(\mathcal{P}) = \sum_{v \in V} ec(v)$, *where* $ec(v)$ *is the number of* v*'s neighbors that do not belong to* v*'s partition.*

Minimizing the total number of cross-partition edges may not always be the goal we want to optimize for. For example, in BSP, in each iteration, we assemble individual messages between two machines into a single message, which incurs a single network communication. This makes sense because the cost mostly comes from the number of network communications rather than the size of the message (given that the size of each message is often very small). Thus, instead of minimizing the total number of cross-partition edges, we need to minimize the total communication volume.

DEFINITION 2 (COMMUNICATION VOLUME OF $\mathcal{P}$). *For a partitioning* $\mathcal{P}$ *on graph* $G(V, E)$, *the communication volume of* $\mathcal{P}$ *is given by* $cv(\mathcal{P}) = \sum_{v \in V} cv(v)$, *where* $cv(v)$ *is the number of partitions (except* v*'s partition) that contain the neighbors of* v.

Besides edge cut or communication volume, we measure the goodness of a partitioning by its *balance*. A partitioning is balanced if each partition has more or less the same amount of nodes. This is desired in a distributed environment for load balance. Given k machines, we expect the graph equally distributed over machines, i.e., each machine has approximately $\lfloor \frac{|V|}{k} \rfloor$ vertices. In general, relaxation is allowed so that the exact number of vertices in a single machine is $(1 \pm \epsilon)\lfloor \frac{|V|}{k} \rfloor$ with $0 < \epsilon \ll 1$.

Based on the goodness measures given above, the graph partitioning problem is: *how to divide a graph into k parts with approximately identical size so that the edge cut size or the total communication volume is minimized in a distributed memory system?* We state it in a more formal manner in Definition 1.

PROBLEM DEFINITION 1 (GRAPH PARTITIONING). *Given a graph* $G(V, E)$ *and a positive integer k, we divide* V *into a set of non-overlapping partitions* $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ *such that (1)* $|C_i| \approx |V|/k$ *for each* C_i*; (2)* $ec(\mathcal{P})$ *(or* $cv(\mathcal{P})$*) is minimized.*

3.2 The Random Partitioning Baseline

We evaluate the quality of random partitioning on two typical classes of graphs: ER graphs [18] and scale free graphs [19]. By doing so, we build the baseline for more advanced partitioning mechanisms.

First, we define the *random partitioning* on a graph. One widely used random partitioning mechanism is that *each vertex chooses a machine uniformly at random from all k possible ones*. After all the vertices finish their selection, the number of vertices on each machine ($|V_i|$) will follow the *Binomial* distribution $B\left(n, \frac{1}{k}\right)$, where n is the vertex number of G. Under this random partitioning, an edge's two end nodes have the probability $1 - \frac{1}{k}$ to be on different machines. The edge cut therefore will follow the Binomial distribution $B\left(|E|, 1 - \frac{1}{k}\right)$.

Given a set V of n vertices, we generate an ER graph G by linking each vertex pair with probability p. We denote a graph G generated in this way as $G \in \mathcal{G}_{n,p}$. Without a loss of generality, in the following analysis, we assume that n and k (number of machines) are the power of 2, and k, p are constants.

LEMMA 1. *If $G \in \mathcal{G}_{n,p}$ and $\mathcal{P}$ is a random partitioning over G on k machines, then we have*

1. $E[ec(\mathcal{P})] = \frac{n^2 p(k-1)}{k}$;
2. $E[cv(\mathcal{P})] = n(k-1)\left(1-(1-p/k)^{n-1}\right)$;

where $E[ec(\mathcal{P})]$ and $E[cv(\mathcal{P})]$ are the expected value of $ec(\mathcal{P})$ and $cv(\mathcal{P})$, respectively.

PROOF. The first equation can be directly derived from the distribution of edge cuts of the random partitioning. We just note that the number of total degrees in an ER random graph is pn^2.

To prove the second equation, consider any vertex u. The probability that u has at least one link to another machine is $\left(1-(1-p/k)^{n-1}\right)$. Thus, the expected number of machines that u is linked to is $(k-1)\left(1-(1-p/k)^{n-1}\right)$. This is because, for each vertex v except u (overall $n-1$ these vertices), it has a probability $1/k$ to be resident in a certain machine and a probability p/k to be additionally linked to u. Sum up all vertices and we have the second equation. $\square$

For scale free graphs, the degree of a vertex follows the power-law distribution: $p(d) = c \times d^{-\beta}$ where $\beta > 1$. That is, the probability that a vertex has degree d is proportional to $d^{-\beta}$ for some constant β. A variety of real networks observe the power-law degree distribution. The constant $c = \zeta(\beta)^{-1}$ is a normalized constant and $\zeta(\beta) = \sum_{1\le n<\infty} n^{-\beta}$, which in fact is the Riemann Zeta Function. β specifies a class of scale free graphs. If G has power-law degree distribution with exponent β, we say $G \in \mathcal{G}_\beta$. Similarly, β and k are given as constants in the following analysis.

LEMMA 2. *Let $\mathcal{P}$ be a random partitioning on a graph $G \in \mathcal{G}_\beta$ on k machines, then we have:*

1. $E[ec(\mathcal{P})] = n\frac{k-1}{k}\frac{\zeta(\beta-1)}{\zeta(\beta)}$;
2. $E[cv(\mathcal{P})] = n\zeta(\beta)^{-1}(k-1)\sum_{i\ge1} i^{-\beta}[1-(1-\frac{1}{k})^i]$

PROOF. The expected degree of a vertex is

$$E[deg(v)] = \sum_{1\le d\le\infty} d \times p(d) = c \times \sum_{1\le d\le\infty} d^{-(\beta-1)} = \frac{\zeta(\beta-1)}{\zeta(\beta)}$$

Then, due to the linearity of expectation, we have:

$$E[ec(\mathcal{P})] = \sum_{v\in G} E[x(v)] = n(\frac{k-1}{k} \times E[deg(v)])$$

where $x(v)$ is the number of v's neighbors on other machines and $E[x(v)]$ is its expectation value.

The proof of the second equation is similar to that of Lemma 1. We just need to highlight that the probability that a vertex u of degree i has at least one link to a certain machine is $1-(1-\frac{1}{k})^i$. $\square$

4. MULTI-LEVEL LABEL PROPAGATION

We present the Multi-level Label Propagation (MLP) algorithm for partitioning billion-node graphs in a distributed memory system. The algorithm can be easily parallelized, which enables it to take advantage of a distributed system.

4.1 Overview

We outline the major steps of MLP in Figure 2 as well as in Algorithm 1. Given a graph G, the algorithm divides G into k balanced partitions stored in k machines. Initially, each vertex is assigned a unique label, which indicates the partition it belongs to. In the end, the entire graph will have k labels, and each label has the same number of vertices.

The algorithm has three steps. The first step is iterative coarsening. In each iteration, we find densely connected substructures through label propagation (LP). We collapse each connected structure into a single vertex to produce a "coarsened" graph. Then we repeat the process until the graph is small enough. The rationale for iterative coarsening is that a single round is not enough to reduce the number of labels to an acceptable level. The coarsening step is controlled by 3 user specified parameters: We keep coarsening the graph until there are no more than α labels (partitions); The label propagation takes at most β iterations to avoid wasteful iterations; The size of each label (partition) is controlled by $\gamma \ge 1$ – each label (partition) has an upper-limit size of $\frac{|V|}{k\gamma}$. In the second step, we partition the coarsened graph using an off-the-shelf algorithm, such as KL or METIS. In the last step we project the partitioning on the coarsened graph to the original graph. As we will see, this step is trivial, and it is not elaborated in Algorithm 1.

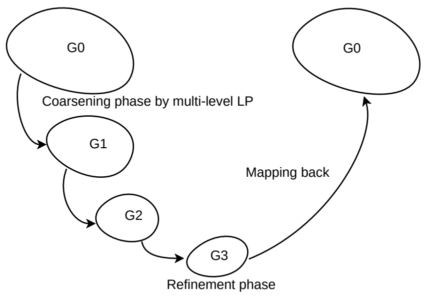


Figure 2: The multilevel label propagation framework

Algorithm 1 Multi-level LP

```
Input: $G(V,E)$, $k$
Output: A balanced partitioning on $V$
   {1. Coarsening phase}
1: $G' \leftarrow G$;
2: while #labels is larger than $\alpha$ do
3:     Run weighted LP under size constraint $\frac{|V|}{k\gamma}$ for $\beta$ iterations; // the result is $\mathcal{P}$;
4:     Construct the coarse-grained graph $G'$ from $\mathcal{P}$; //
5: end while
   {2. Refinement phase}
6: Refine($G'$,k);
   {3. Projecting back}
```

MLP distinguishes itself from previous graph partitioning methods in two aspects: i) Coarsening by LP instead of maximal matching (as in METIS) keeps the semantics or the structures of the graph. It is also more efficient than maximal matching. ii) In MLP, once two vertices are assigned the same label in a coarsened graph, they will always share the same label in coarser graphs in later iterations. METIS, on the other hand, needs to relabel the vertices in the refinement step.

4.2 The Coarsening Step

Let $G_0 = G(V,E)$ be the input graph, and let $G_1, ..., G_i, ..., G_t$ be the intermediate graphs generated during the coarsening step. Let $\mathcal{P}_i = \{C_1, C_2, ..., C_n\}$ be the partitioning derived by running LP on G_i. Each $\mathcal{P}_i$ is defined on G_i. We define a coarsened graph as follows:

DEFINITION 3 (COARSENED GRAPH). *For a graph $G_i(V_i, E_i)$ and a partitioning on the graph $\mathcal{P}_i = \{C_1, C_2, ..., C_n\}$, the coarsened graph G_{i+1} is a graph with vertex set V_{i+1} and edge set E_{i+1}, where $V_{i+1} = \mathcal{P}_i$, and $(C_i, C_j) \in E_{i+1}$ iff $\exists u \in C_i, v \in C_j$ such that $(u, v) \in E_i$.*

As an example, consider the coarsened graph in Figure 1(c). It is produced from the partitions $\{\{a, b, c, d\}, \{e, f, g, h\}, \{i, j, k, l\}\}$ of the graph in Figure 1(a).

4.2.1 Label updating

The original label updating mechanism of LP works like this: each vertex is assigned a label which is the most frequent label of its neighbors. If multiple labels have the same top frequency, we pick one randomly. In MLP, we make two extensions to the original label updating mechanism.

Structure-preserving label updating

A good labeling procedure should be 'semantic-aware,' i.e., vertices sharing a lot of common neighbors should be assigned the same label, i.e., grouped together. Instead of using the common practice of randomly choosing among the most frequent labels, we select the minimal label to better support semantic awareness.

To illustrate, consider the first iteration of LP. In the first iteration, each vertex has a unique label. Hence, all of the labels have the same frequency. We compare two strategies. First, we randomly select a label of its neighbors. Second, we select the minimal label of its neighbors (assuming labels can be linearly ordered). Assume both nodes u and v have 10 neighbors. If half of the neighbors are the same, the probability that u and v are assigned the same label is *less than* $1/10$ under the first strategy, while under the second strategy the probability becomes $1/3$. If all of the neighbors are the same, the probability is $1/10$ under the first strategy, while under the second strategy the probability becomes 1.

We formally show this in Lemma 3. The first statement shows that even if when two vertices share a significant number of neighbors, the probability that they are clustered together is less than $\frac{1}{\max\{|N(u)|,|N(v)|\}}$. In contrast, the second statement shows a desired property of the second strategy: The more neighbor sharing, the more likely the two vertices are clustered together. Lemma 3 further ensures that the second strategy is better than the first strategy. The upper bounds of the two probabilities also implies that $p'(u, v)$ may be significantly larger than $p(u, v)$.

LEMMA 3 (RANDOM VS. MINIMAL LABEL). *Let $u, v \in V$ be two vertices and $N(u)$ be the neighbors of u. Suppose each vertex is assigned a unique label and there is a linear order on all labels. The following statements hold:*

1. *The probability that a randomly selected label from $N(u)$ is identical to a randomly selected label from $N(v)$ is*
$$p(u, v) = \frac{|N(u) \cap N(v)|}{|N(u)||N(v)|} \leq \frac{1}{\max\{|N(u)|, |N(v)|\}}$$
2. *The probability that the minimal labels of $N(u)$ and $N(v)$ are identical is*
$$p'(u, v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|} \leq 1$$
3. $p'(u, v) \geq p(u, v)$

PROOF. We first prove statement 1. There are $|N(u)| \times |N(v)|$ choices. Among them, $|N(u) \cap N(v)|$ lead to the same label. Hence, we have $p(u, v) = \frac{|N(u)\cap N(v)|}{|N(u)||N(v)|}$. Since $|N(u) \cap |N(v)| \leq \min\{|N(u)|, |N(v)|\}$, we have $p(u, v) \leq \frac{1}{\max\{|N(u)|,|N(v)|\}}$.

To prove statement 2, we just need to highlight that $p'(u, v)$ is equivalent to the probability that the *unique* minimal label of $N(u) \cup N(v)$ belongs to $N(u) \cap N(v)$. Hence, $p'(u, v) = \frac{|N(u)\cap N(v)|}{|N(u)\cup N(v)|}$.

For statement 3, we distinguish two cases: (1) $N(u) \cap N(v)$ is empty and (2) not empty. In the first case, $p(u, v) = p'(u, v) = 0$. In the second case, we just need to show that $|N(u)||N(v)| \geq |N(u) \cup N(v)|$. Since $N(u) \cap N(v)$ is not empty, it contains at least one element. Hence, $|N(u) \cup N(v)| \leq |N(u)| + |N(v)| - 1$. It is trivial to show that $|N(u)| + |N(v)| - 1 \leq |N(u)||N(v)|$. □

As a direct consequence of the new label updating rule, vertices in the same community tend to share the same labels. This enables the coarsened graph to preserve the structure of the original graph. The labeling rule also leads to faster convergence. If we randomly pick a label when more than one label has the top frequency, even when two vertices lie in the same community, they are quite possibly assigned different labels. Our labeling rule can avoid such wasteful labeling.

Weighted label updating

In order to perform multi-level graph coarsening, we extend LP for weighted graphs. We model each intermediate graph as a weighted graph. For G_0, we assign a unit weight to its vertices and edges. For G_{i+1}, which is constructed from G_i and $\mathcal{P}_i$, the weights are assigned as follows:

DEFINITION 4 (WEIGHT FUNCTION). *Let G_{i+1} be the coarsened graph derived from G_i. We have $V_{i+1} = \{C_1, C_2, ..., C_n\}$, where each node C_i is weighted as:*
$$w(C_i) = \sum_{u \in C_i} w(u) \qquad (1)$$
and each edge $e = (C_i, C_j)$ is weighted as:
$$w(C_i, C_j) = \sum_{e(u,v)\in E_i, u\in C_i, v\in C_j} w(e(u, v)) \qquad (2)$$

With weights, we can further improve the quality of label updating. For node u, assume its neighbor v has label c. Previously, neighbor v contributes a unit weight to c. If c has the highest contributed weights, it becomes the label of u. In our new approach, instead of contributing a unit weight to c, node v contributes weight
$$\frac{w(e(u, v))}{w(v)} \qquad (3)$$

The rationale is as follows. In the coarsened graph, each node v represents a set of nodes in the original graph. Let $V(v)$ denote the set of nodes v represents. For a pair of nodes u and v, node u is more likely to take v's label if the subgraph induced by $V(u)$ and $V(v)$ in the original graph has a higher density. Based on the definition of node and edge weights in Eq 1 and Eq 2, we know that the density of the induced graph $G_{V(u)\cup V(v)}$ is $\frac{w(e(u,v))}{w(u)w(v)}$. When we update u's label, $w(u)$ in the denominator is the same for all of u's neighbors. Hence, $w(u)$ can be omitted, which leads to Eq 3.

With the weight defined, we update the label for a node u as follows. For each label c, we define its score as
$$s(c) = \sum_{v\in N(u), L(v)=c} \frac{w(e(u, v))}{w(v)} \qquad (4)$$
where the summarization runs over all of u's neighbors whose label is c. Then, u will select the label c that has the maximum score. In general, it is unlikely that two labels will have the exact same score. When two or more labels have the same maximal score, we apply the strategy of selecting the minimal label as described above. In the real implementation, to allow more potential labels to join the completion, we select the minimum label whose score is larger than a threshold.

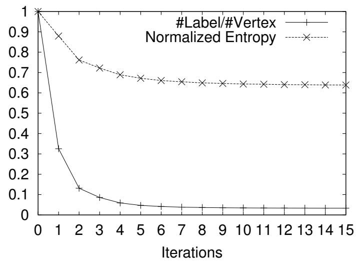


Figure 3: Convergence of LP on LiveJournal network

4.2.2 *γ-size Partitions*

Our goal is to partition a graph so that each partition is stored in a distributed machine. The partitions produced by the naive LP approach may be extremely imbalanced: It may generate many small partitions, and a few very large ones. The size of these large ones is often larger than the capacity of a single machine.

To avoid generating a cluster with too many vertices, we adopt a size threshold. Specifically, when the size of a label (partition) reaches an upper limit, we "freeze" the label, which means the membership of the partition can no longer change. To ensure a balanced partition, each machine should hold around $\frac{|V|}{k}$ vertices, where k is the number of machines. We use a smaller value $\frac{|V|}{\gamma \times k}$ as the upper limit, where $\gamma \geq 1$ is a parameter to control the upper limit. The rationale is as follows.

A larger γ leads to a smaller upper limit, which in turn leads to more labels (partitions) to be produced. Note that in the second step of Algorithm 1, we will merge small labels using high-quality partitioning algorithms such KL [5] or METIS [8]. Large labels whose size is close to a machine's capacity will be excluded from the merge step. In contrast, by setting $\gamma > 1$, we can ensure that all labels (partitions) will join the second step to produce a high-quality final partitioning.

On the other hand, when γ is too large, a closely connected substructure is often fragmented into pieces, which in turn will hurt the overall partitioning quality. Additionally, a larger γ may also produce a large number of labels at the end of the coarsening step. As a result, the coarsened graph may be still too large to be processed by the refinement step. Hence, we need a good trade-off. Through experimental study, we find that for most real networks, setting γ as $\Omega(k)$ produces the best partitioning quality.

4.2.3 *β-depth LP*

In general, we hope the coarsening process shrinks the graph as fast as possible without losing any structure information. Intuitively, the two goals – fast shrinking and information preservation – conflict with each other.

To address this problem, we first reveal an observation we made in the shrinking process. In LP, the number of labels tends to decrease fast in the early iterations and then remain relatively stable. As shown in Figure 3, in a typical real network, after 5 or 6 iterations, the number of labels and the *normalized entropy* will no longer change. Here, we define the entropy as follows. For a partitioning $\mathcal{P}$ defined on n vertices, the entropy of $\mathcal{P}$ is defined as $-\sum p_i \log p_i$, where each p_i is the probability that a vertex belongs to C_i. We normalize the entropy by dividing the maximal entropy value $\log n$ over n vertices. When the entropy remains stable, the distribution of vertices in different labels (partitions) changes marginally. In other words, the labeling changes little.

To act on this observation, we use a parameter β to control the shrinking speed and coarsening quality: We only run LP for β iterations. We call such LP β-depth LP. The overall complexity of MLP is $O(t\beta|E|)$, where t is the number of runs of β-depth LP. With the same complexity, we may choose to run one LP with depth $t\beta$, such a strategy is denoted by $1 \times t\beta$; or alternatively run t β-depth LP, denoted by $t \times \beta$. The following simple analysis shows that generally $t \times \beta$ is better than $1 \times t\beta$.

Let $\mathcal{P}_i$ be the partitioning after running the i-th β-depth LP on G_i. Let $\mathcal{P}'_i$ be the partitioning on $V(G)$ by mapping $\mathcal{P}_i$ back to the input graph G. We can establish a linear order based on the finer relationship among partitionings $\mathcal{P}'_0, ..., \mathcal{P}'_t$ due to Lemma 4. Given two partitionings $\mathcal{P}_1, \mathcal{P}_2$ defined on the same set, $\mathcal{P}_1$ is *finer* than $\mathcal{P}_2$, if for each $C_i \in \mathcal{P}_1$, $C_i \subseteq C_j$ where $C_j \in \mathcal{P}_2$. Thus, intuitively, in $t \times \beta$, after each run of β-depth LP, we enforce that vertices with the same label in G_i will always share the same label in G_{i+1}. However, in contrast, in each iteration of $1 \times t\beta$, vertices with the same label may be assigned different labels in the later iterations, thus easily leading to oscillations. We will provide more evidence in the experiment section.

LEMMA 4. *In MLP, for each i, partitioning $\mathcal{P}'_i$ is finer than $\mathcal{P}'_{i+1}$.*

4.3 The Refinement Step

Let $G_t(V_t, E_t)$ be the final graph produced by the coarsening step. We further partition G_t in the refinement step and adopt the following guidelines: first, each label (partition) is the smallest unit to be distributed; second, we want to create a balanced distribution; third, we want to minimize the total weight of crossing edges among the machines.

We discuss two ways to distribute G_t across k machines. The first is *multiprocessor scheduling* (MS), the second is *weighted graph partitioning* (WGP). We assume that the machines have enough space to hold the clusters to be assigned, i.e., we have a fixed number of machines with finite but large enough space.

4.3.1 *A baseline approach: the MS model*

One direct model to meet the first two requirements is *multiprocessor scheduling* (MS).

DEFINITION 5 (MS). *For a given k and a partitioning $\mathcal{P} = \{C_1, C_2, ..., C_n\}$ over set V, find partitions $\{S_1, S_2, ...S_k\}$ over V such that (1) each S_i is a union of subset in $\mathcal{P}' \subseteq \mathcal{P}$, i.e., $S_i = \cup_{S \in \mathcal{P}'} S$ and (2) the value* $\max\{|S_i||1 \leq i \leq m\}$ *is minimum.*

The first condition ensures that each label is distributed as a whole. The second condition requires us to find the most balanced assignment. Clearly, MS is NP-complete. We use a greedy algorithm to solve this problem. We first sort the subsets in $\mathcal{P}$ by their sizes in descending order, then we assign each subset to the machine with the largest remaining capacity. It is known [21] that this greedy algorithm produces an approximation of $4/3 - 1/(3k)$. The greedy approach is also efficient with a time complexity of $O(|V_t| \log |V_t|)$. Thus, we can handle large G_t. This implies that we can perform MS-based refinement even when the coarsening is conducted for a limited number of levels.

However, the MS model does not follow the third guideline, i.e., minimizing the total number of crossing edges. In general, the MLP+MS approach (i.e., coarsening by multilevel LP and refining by MS) is efficient but may introduce a loss in partitioning quality.

4.3.2 *An improved approach: the WGP model*

To minimize the edge cut size, we propose a *weighted graph partitioning* (WGP) model.

DEFINITION 6 (WGP). *For a given k and a weighted graph $G(V, E)$, find a partitioning $\mathcal{P} = \{C_1, C_2, ..., C_k\}$ of V such that*

- (Weighted balance) *for each C_i, $|W_i| \approx W/k$, where W_i is the sum of vertex weights in C_i and W is the total vertex weight;*
- (Minimizing edge weight) $\sum_{e \in E_C} w(e)$, *where E_C is the set of edges with two ends lying in different parts of $\mathcal{P}$.*

This model is different from its unweighted version in two aspects. The first is that we want to achieve balance in terms of total vertex weight. The second is that the number of cut edges is replaced by the total weight of crossing edges.

We use METIS to solve WGP. Note that METIS in general can only handle graphs with a couple millions of nodes. Hence, in our overall algorithm framework, we need to perform coarsening until the coarsened graph can be handled by METIS. In Algorithm 1, we use parameter α to control the size of the graph that METIS can handle. Of course, any other weighted graph partitioning algorithms can also be applied.

For MLP + WGP, two key issues have impact on the partitioning quality. First, we hope that when a good partitioning on the coarsest graph is projected back, the projected partitioning is also good in the original graph. This holds due to Lemma 4. Second, we hope that the coarsening step can ensure the partitioning quality.

MLP addresses the second issue using the following approach. First, we use Eq 3 to ensure that edges with large weights are coarsened into a single vertex. Second, in MLP when G_i is coarsened to G_{i+1} the following equation holds:

$$w(G_{i+1}) = w(G_i) - w(\mathcal{P}_i) \tag{5}$$

where $w(G_i)$ is the total edge weight of G_i, and $w(\mathcal{P}_i)$ is the total weight of edges that lie within each part of $\mathcal{P}_i$. Thus, the final goal in WGP of minimizing the total crossing edge weight is compatible to the coarsening of large-weight edges. Hence, the coarsening step in MLP naturally leads to a good partitioning.

5. DISK-BASED MLP

In this section, we show that, even if we do not have a large number of machines to hold an entire billion-node graph in the memory, we can still partition the graph without much performance penalty.

5.1 Rationale

Real life graphs are becoming bigger and denser. The Facebook social network currently has about 1 billion nodes and over 100 billion links. The size of the graph topology is at least 1T. If each machine has 32G memory, we need at least 30 machines to hold the graph topology in memory. For the web graph, hundreds of machines are needed. To make things worse, real life graphs are becoming denser [22]. Let $\bar{d}$ denote the average degree of a graph. A graph consumes $\Theta(\bar{d}\cdot|V|)$ memory space. It has been shown that for real life graphs, the numbers of edges and nodes at time t, denoted by e_t and n_t, respectively, satisfy $e_t \propto n_t^\delta$ where $1 < \delta < 2$. This shows that real life graphs are getting denser over time. Clearly, for web-scale graphs, *memory is a bottleneck*. Small organizations are not able to work on large graphs because they cannot afford to deploy a large memory cloud. Hence, the challenge is: Can we run our algorithms on a big graph using limited memory?

5.2 Overview of our approach

Recall that we keep graphs memory resident because online graph queries incur random accesses on a graph, and disk-based random accesses are very slow. But for queries that only access a portion of the graph instead of the entire graph, we only need to keep that portion memory resident. In MLP, the labels of the vertices are updated independently. Thus, if we are able to schedule MLP sequentially, then we can pipeline the execution on different portions of the graph, and at any point of time only one portion of the graph need to be memory resident. This enables us to partition graphs of any size.

In Trinity, each machine manages a sub-graph. We divide a subgraph into a set of disjoint *blocks* so that each block is small enough to fit in the memory of a single machine. A block contains a set of vertices and their adjacent lists. During computation, we load the blocks into memory one by one. For each block, we need to ensure we can carry out the following computation when no other blocks is in memory: each vertex in the block sends its label to all of its neighbors, receives messages from its neighbors, and finally updates its label. This allows us to pipeline the process to improve the usage of the CPU, disk I/O, and network communication.

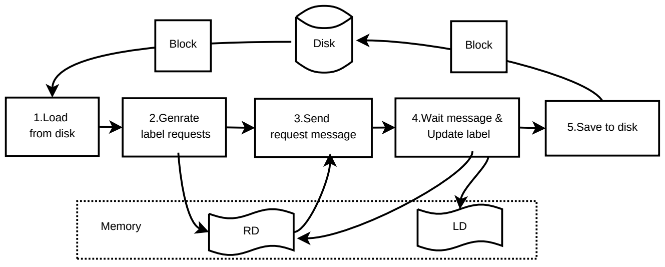


Figure 4: Process of pipeline execution

5.3 Pipelined MLP (pMLP)

We now present the details of how to perform LP in a pipelined manner.

Data Structures. Assume we are currently performing the k-th iteration of LP on machine i. Let V_i be the vertices that reside on machine i, and at any point, only a single block, which contains vertices $S \subseteq V_i$, is in the memory.

Machine i is both a consumer and a producer. As a consumer, it requests remote machines to send labels of S's neighboring vertices. As a producer, it needs to provide labels of V_i to other machines upon request. Since machine i only keeps S, which is a subset of V_i, in memory, in order to provide V_i's labels, it needs to perform disk I/O. To avoid disk I/O, we keep the labels of V_i memory resident. Note that keeping the entire V_i's adjacent lists memory resident takes $O(\bar{d} \cdot |V_i|)$ space, where $\bar{d}$ is the average degree ($\bar{d} > 100$ in Facebook), while keeping the labels only takes $O(|V_i|)$ memory space. In fact, in each machine, we keep the following three dictionaries in memory, as shown in Table 1.

	Content	Usage
LD1	(*vertex, label*)	V_i's labels in iteration $k-1$
LD2	(*vertex, label*)	V_i's labels in iteration k
RD	(*vertex, machine/label*)	Neighbors' machine ID and labels

Table 1: In-memory dictionaries

The dictionaries support constant-time lookup and update (by indexing on vertex). We use dictionary LD1 to maintain V_i's labels generated in the last iteration, and LD2 to maintain V_i's labels generated in the current iteration. At the beginning of a new iteration, the content of LD2 is copied over to LD1. Dictionary RD is used as an input/output buffer. A machine generates a request in the form of (*vertex*, *machine*), meaning it needs to know the label of the *vertex* that resides on the given *machine*. The requests are buffered and sent in batch, and the responses come back in the form of (*vertex*, *label*). RD buffers and groups these incoming and outgoing messages.

Basic Pipelining Procedure. During the pipelined execution, each machine performs the five steps illustrated in Figure 4. In the first step, we load the next block from disk. In the second step, for each vertex u in the block, we find the labels of its neighbors. For each remote neighbor *v*, we generate a label request message in the form of (*v, machine*), where *machine* is the machine where vertex *v* resides. We then add each message into RD (using dictionary RD as a sending buffer). In the third step, after all label request messages are generated, we group them by *machine* and send each group as a single message to the remote machine. In the fourth step, we wait until all message requests are processed by the remote machines and all the responses come back in the form of (*v*, *label*) in RD, where *label* is the label for *v*. Using the label information of neighboring nodes, we update the labels of V_i in LD2. Finally, the block is disposed from memory, and we enter the next iteration. Besides updating the labels of V_i, each machine also receives remote requests for the labels of V_i. The requests are served by dictionary look ups in LD1.

We pipeline the above procedure to improve the effectiveness of system resources, including CPU, disk controller, and network bandwidth. Each of the above steps is carried out by a single process. Through pipelining, the CPU usage and the network usage are significantly improved. Furthermore, all communication mechanisms, such as sending, receiving, deadlock detection and synchronization, are provided as the off-the-shelf components of Trinity. These off-the-shelf components are general enough for our purpose (no customization of algorithms or data structures are needed).

Label size estimation. After one β-depth LP, we need to count the size of each label to construct a coarser graph. A naive solution needs to count the sizes of labels on each machine, then accumulate the label size on a master computer. This solution needs overall $O(k|L|)$ communications and poses $O(k|L|)$ computation overheads on the master computer, where L is the set of labels. In the first several iterations of MLP, $|L|$ may be quite close to $|V|$. In many cases, it is larger than one-tenth of $|V|$. To reduce the communication cost and computation overheads on the master computer, we sample vertices in each machine with probability q and estimate the label size as n'/q, where n' is the number of vertices with the label in the samples. In this manner, the communication complexity is $O(|V|q)$. In general, we can use quite a small q to ensure that $O(|V|q)$ is smaller than $O(k|L|)$ with bounded estimation error.

Coarser graph generation. Consider the procedure to generate G_i by G_{i-1} and $\mathcal{P}_{i-1}$. We first use hash functions

$$h(l(v)) = l(v) \mod k \tag{6}$$

to distribute vertices $v \in G_{i-1}$ with label $l(v)$ and their adjacent lists onto k machines. This hash function ensures that vertices with the same label will be assigned on the same machine and the graph creation job is evenly distributed over k machines. Then, on each machine, we use the next two steps to create G_i. First, for each label x we create a new vertex u of G_i for all vertices in G_{i-1} with the label. Meanwhile, we create the mapping between u and $V(u)$ (the vertices in G_0 that u represents). Next, we create the new adjacent list for u. We collect the new vertex id in G_i for all neighbors of $\{v|v \in G_{i-1}, l(v) = x\}$ in G_{i-1}. This can be achieved by message passing. Each of these new vertices represents a neighbor of u in G_i. Simultaneously, we calculate the weight of edges between u and each of of its new neighbors. Once G_i is created, G_{i-1} is dropped off.

5.4 Analysis

We give the analysis of complexity of time, space, and communication for pMLP, then discuss the influence of block size.

Complexity analysis. The time complexity is obviously $O((t\beta + c)|E|/k)$ for t runs of β-depth LP, where c accounts for the rounds of message passing for coarser graph creation at the end of each β-depth LP. Within each iteration of LP, overall $O(2|V_i| + |S|\bar{d})$ memory space is needed on each machine, where $S \subseteq V_i$ is a block of V_i. $2|V_i|$ accounts for the space usage of LD1 and LD2. $|S|\bar{d}$ accounts for the space usage of RD. In general, we can set $|S| \sim \frac{|V_i|}{\bar{d}}$ to limit the size of RD.

Next, we present the communication complexity analysis. Note that we use dictionaries to avoid duplicate label requests. Lemma 5 shows that when V_i can be entirely loaded into memory as a block (i.e. $|S| = |V_i|$), the total number of label requests under partitioning $\mathcal{P}$, is exactly $cv(\mathcal{P})$. When allowing duplicate messages, the total number of messages is obviously $ec(\mathcal{P})$. Hence, by using a dictionary, in the best case we reduce the communication complexity from $\Theta(ec(\mathcal{P}))$ to $\Theta(cv(\mathcal{P}))$. As shown in Section 3, $cv(\mathcal{P})$ is smaller than $ec(\mathcal{P})$.

LEMMA 5. *Given a partitioning $\mathcal{P}$ over graph G on k machines, the total number of messages is $\Theta(cv(\mathcal{P}))$ when $|S| = |V_i|$.*

When $|S|$ is smaller than $|V_i|$, a constant factor $\zeta = \frac{|V_i|}{|S|}$ will be introduced into the above result because a remote vertex label may be requested approximately ζ times. However, whatever ζ is, $ec(\mathcal{P})$ is always an upper bound of the total number of messages. Thus, a more accurate estimation of the communication complexity is $\Theta(\min\{\zeta cv(\mathcal{P}), ec(\mathcal{P})\})$.

Block size. The block size $|S|$ determines the memory usage and communication overheads. The larger $|S|$ is, the more memory will be consumed and the fewer total communications will be produced. $|S|$ has also another indirect influence on the overall performance. When $|S|$ is quite large, too many messages will be sent to other machines, which in turn will overload these machines and degrade their performance. Hence, $|S|$ is critical for the balance between the message generating speed and consuming speed, and the trade-off between memory usage and communication volume. In general, the optimal block size depends on CPU speed, parallelism efficiency, network speed, and memory size. It is hard to quantify the optimal block size. We will explore this by experiments.

6. EXPERIMENTS

We present experiment results in this section. For performance, we measure *peak memory usage* and *running time*. For partitioning quality, we measure *edge cut size* (*ec*), *total communication volume* (*cv*), and imbalance, where imbalance is measured by the maximal imbalance or the percentage by which the size of the largest part exceeds the average partition size. We evaluate two versions of our solution: LP+MS and MLP+METIS.

We further compare our solutions to two typical versions of METIS as well as the baseline random partition approach. One is recursive bisection with RM, denoted by rb+rm. The other is k-way partitioning with sorted HEM, denoted by kway+shem. Sorted HEM (SHEM) is identical to HEM except that vertices are matched in the descending order of degree so that it will match small-degree vertices as well. In general, SHEM shrinks a graph faster than HEM. For comparison with random partitioning, we generate 10 random partitionings, then report the average measures.

6.1 Sequential MLP

We implement a sequential version of MLP in C (the same as METIS), and then compare it to the sequential version of METIS. We run all experiments on a PC with Intel Xeon at 2.67GHz, 48G memory running 64-bit Windows server 2008. We compare MLP and its competitors on both real life graphs and synthetic graphs.

6.1.1 Experiments on real life graphs

Datasets. We use three large graphs: LiveJournal, WikiTalk, and Patents [4]. LiveJournal is an online social network, in which nodes represent users and edges represent the friendship relationship. In the WikiTalk graph, nodes represent Wikipedia users, and a directed edge from node i to node j represents that user i at edited a talk page of user j at least once. Patents is the citation network of patents in the U.S.

[4]Download from http://snap.stanford.edu/index.html

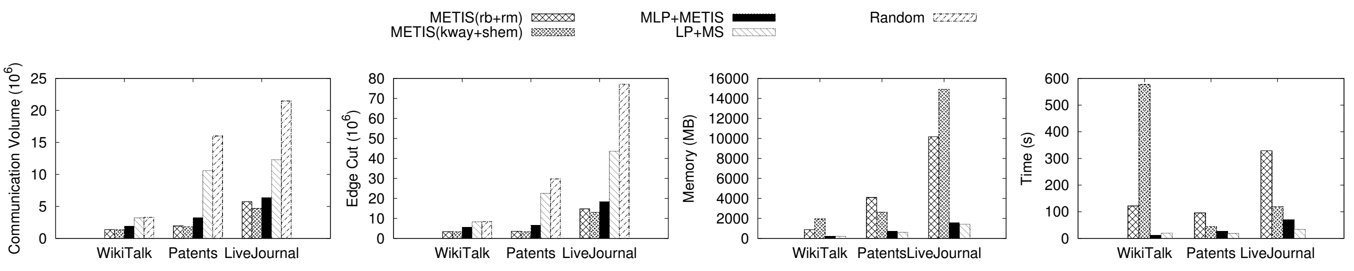


Figure 5: Quality and performance on million-node real graphs

Network	#Vertex	# Edge	size (M)
WikiTalk	2394385	4659565	53.8
Patents	3774768	16518947	154.8
LiveJournal	4846609	42851237	363.9

Table 2: Basic information of real graphs

Partitioning quality. The first two plots of Figure 5 show the quality of partitioning for real life graphs. The quality is measured by cv (communication volume) and ec (edge cut). The results were obtained by running MLP+METIS with 3×5 coarsening strategy and LP+MS with one 15-depth LP. As we can see, our solutions and METIS produced partitions of significantly better quality than random partitioning. The comparisons show that MLP+METIS's quality is comparable to that of METIS. LP+MS's quality is weaker than METIS but still better than random partitioning. The imbalance results are omitted since for both our solutions and its competitors on all tested networks, the maximal imbalance is less than 3%, which is minor and can be ignored.

Performance. The last two plots in Figure 5 show the performance results of various partitioning algorithms. The results were obtained by running MLP+METIS with 3×5 coarsening strategy and LP+MS with one 15-depth LP. The results imply that a minor sacrifice of quality brings significant performance improvement. In general, MLP+METIS and LP+MS consume significantly less memory than METIS. We can see that MLP+METIS or LP+MS consistently runs faster than METIS on different real networks. In some cases (e.g., memory usage on LiveJournal, running time on WikiTalk), the performance of our solutions is one order of magnitude better than METIS.

Convergence. In this experiment, we compare the speed of convergence of different coarsening strategies. We show the results on LiveJournal in Figure 6. Results on other graphs are similar and are omitted here to save space. From the results, we can see that SHEM converges fastest since it pays special attention to matching small-degree vertices. The speed of convergence of two of our strategies is close to that of SHEM. We also compared different $a \times b$ strategies of our solution. We found that 3×5 performs the best. In 3×5, after each 5-depth LP there is a clear drop in the number of labels. This observation indicates that by creating a new graph and then running LP on the new graph we reduce the graph size further. We also found that by 3×5 coarsening, the coarsest graph is almost two orders of magnitude smaller than the original graph. This is really important when partitioning a large graph with billion nodes.

6.1.2 Experiments on synthetic graphs

Synthetic graph model. We generated a collection of synthetic graphs with embedded communities to test the effectiveness and efficiency of MLP. We used the graph model [47] that generates synthetic graphs with varying degree of "clearness" of community structure. There are three parameters α, β, μ in the model. Both degree and community size follow power-law distribution with exponents α and β, respectively. The parameter μ controls the proportion of neighbors of a vertex that reside in other communities. By tuning μ, we vary the clearness of the community structure. We generated five graphs with 1 million vertices and approximately 10.6 million edges by setting $\alpha = 2, \beta = 3$. The parameter μ varied from 0.1 to 0.5. As μ becomes smaller, the boundaries of the communities in the graph become clearer.

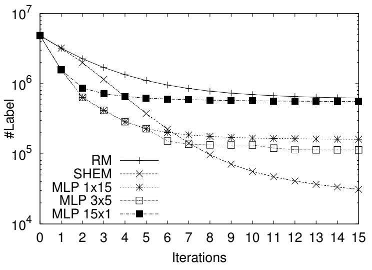


Figure 6: Convergence of coarsening on LiveJournal

Partition quality and performance. Table 3 measures the quality and the performance of METIS and our partitioning method on synthetic graphs. We ran MLP+METIS under the 1×10 coarsening strategy. We found that MLP uses much less memory and time than the two versions of METIS. The quality of MLP, measured by ec (edge cut) and cv (communication volumn), is not only comparable but in some cases better than METIS. The partitioning quality of MLP is consistently better than METIS(kway+schem) for different μ. It is also clear that when boundaries between communities are clearer (smaller μ), MLP's quality is closer to that of METIS(rb+rm). The imbalance of all solutions can be ignored since the maximal imbalance is at most 3%.

In summary, the results on the synthetic graphs with embedded communities sufficiently show that *MLP can effectively leverage the community structure of graphs to generate a good partitioning with less memory and time*. In contrast, METIS, which is based on the maximal matching method, is not community-aware when it coarsens a graph, thus heavily relying on costly refinement in the uncoarsening phase to ensure the solution quality. As a result, METIS incurs more time and space costs.

6.2 pMLP

We next examine the scalability and effectiveness of pMLP. The following results were all obtained by running pMLP on Trinity.

Speedup. To study the efficiency of pMLP, we ran pMLP under the 3×5 coarsening strategy on the LiveJournal graph using 2 to 8 machines. The results are shown in Figure 7. We can see that the running time almost linearly decreases with the number of machines. We achieved almost 6 speedup on 8 machines, indicating that our parallel solution is effective.

μ	cv ($\times 10^6$)			ec ($\times 10^6$)			Memory (MB)			Time(s)			Imbalance Ratio		
	M1	M2	MLP	M1	M2	MLP	M1	M2	MLP	M1	M2	MLP	M1	M2	MLP
0.1	0.64	0.89	0.64	0.40	0.66	0.40	994.74	591.47	352.08	12.31	4.37	5.117	1.00	1.00	1.03
0.2	1.61	2.01	1.59	1.06	1.62	1.05	1162.18	1007.22	354.98	19.67	9.58	7.33	1.00	1.02	1.03
0.3	2.17	2.64	2.20	1.60	2.43	1.61	1541.54	1359.48	406.07	21.47	12.15	8.46	1.00	1.03	1.03
0.4	2.80	3.29	3.01	2.23	3.18	2.39	1863.83	1745.54	516.50	26.35	18.61	9.67	1.00	1.03	1.03
0.5	3.31	3.83	3.83	2.85	3.96	3.53	2192.56	2091.88	646.47	29.97	27.32	12.22	1.00	1.00	1.03

Table 3: Quality and performance on synthetic networks. M1 is METIS(rb+rm), M2 is METIS(kway+shem).

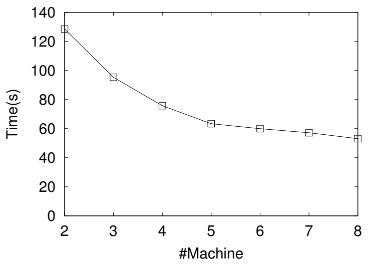
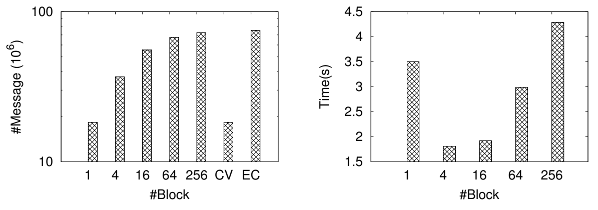


Figure 7: Parallelism efficiency of pMLP

Block size. In this experiment, we examined the influence of block size (for a set of vertices V_i on a machine, block size determines the number of blocks). The experimental settings were the same as used in previous experiments, and the only difference is that we ran it on 8 machines with a varying numbers of blocks. As shown in Figure 8(a), the number of messages increases with the number of blocks. We also show ec and cv of the partitioning. We can see that, when there is only one block, the system produces a minimal number of messages (which is close to cv). On the other hand, when the number of blocks is very large, many redundant messages are generated and the total number of messages is close the theoretical worst case of ec. The running time under different block numbers is given in Figure 8(b). As expected, an optimal number of blocks exists. In our experiment, 4 blocks led to the least time.



(a) # Message (b) Running time

Figure 8: Performance of partitioning algorithms.

Scalability. We generated synthetic graphs with the numbers of vertices ranging from 64M to 512M and an average degree of 26 using the RMAT [46] graph generator (a widely-used model to generate graphs with power-law degree distribution). The RMAT model has four parameters a, b, c, d. We set the parameters 0.45, 0.15, 0.15, and 0.25, respectively. We ran MLP+METIS under the 5×5 coarsening strategy. All other experiment settings are the same as that used in the previous experiment. We used 8 machines, each of which had 48G of memory. The results are shown in Figure 9. As we can see, the running time almost increases linearly with the growth of the graph size.

We want to highlight that on the largest graph, which has 512M nodes and 6.5G edges, it only takes about 4 hours and 8×48G total memory for pMLP to finish the partitioning job. This is the largest graph that has ever been partitioned to the best of our knowledge. For this graph, after 5×5 coarsening, the coarsest graph has only about 1.4M nodes, which is a significant reduction. This reduction is the key reason why pMLP can scale up to web-scale graphs. For all the tested large graphs, the maximal imbalance is less than 5%. The RMAT graphs do not necessarily have community structures.

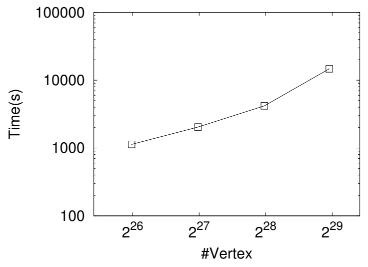


Figure 9: Scalability to billion-node graphs

Hence, it is less possible to find a partitioning that is significantly better than a random partitioning. In spite of this, the ec and cv of the partitioning produced by pMLP is only 90% of that of random partitioning. For web-scale real life graphs that contain communities, the partitioning results will be even better.

7. RELATED WORK

We survey related work on graph partitioning from four angles.

Large graph partitioning. The problem of graph partitioning is an NP-complete problem [23]. Earlier efforts focused on designing effective sub-optimal algorithms. Typical algorithms of this class include local search based solutions (such as KL [5] and FM [6]), which swap heuristically selected pairs of nodes, simulated annealing [24], and genetic algorithm [25] based solutions, etc. To scale up to graphs with million of nodes, multi-level partitioning solutions, such as Metis [8], Chaco [7], and Scotch [10], have been proposed. These algorithms are further parallelized to handle even larger graphs, and examples of these parallelized solutions include ParMetis [11] and Pt-Scotch [12]. To summarize, existing solutions can partition graphs with up to tens of millions nodes. The method presented in this paper is the first that is able to partition billion-node graphs on a general-purpose distributed memory system.

Distributed graph computing platform. Our solution is built on top of Trinity [17], but the algorithm is portable to other distributed graph computing schemes that support efficient vertex-centric programming and message passing. In recent years, a variety of distributed graph computing platforms have emerged, including P-BGL [26], Pregel [15], InfiniteGraph [28], HyperGraphDB [29], and many others are under development [30]. PBGL is a C++ library for parallel distributed computation on graphs and support message passing by MPI. Pregel is provided as a MapReduce-like programming framework to support vertex-centric programming model. InfiniteGeraph and HyperGraphDB are graph database engines that support storage, query, and update of a graph database. Most of them by default use the random partitioning to partition the data. Many of them, such as PBGL and Pregel, provide the flexibility to allow users to specify the partitioning. However, none of them provides an efficient algorithm or solution to partition a large graph. An ongoing project GPS [31] has been aware of partitioning on web-scale graphs on a general-purpose distributed platform, but the project is still in an early stage, and neither documentation nor source code is available.

Label propagation. Raghavan et al. [32] first proposed using label propagation (LP) to detect communities in real networks. Later, Barber et al [33] reformulated the naive LP approach to an equivalent optimization problem. Leung et al. [34] found that LP leads to 'monster' communities that contain more than half of the vertices in the graph, and use a parameter to penalize a label by distance. LP was further improved to detect communities in bipartite networks [35], to detect overlapping communities [36], and to identify core, as well as whisker, communities [37]. LP also offers applications for compressing a large social network [38]. However, to the best of our knowledge, LP is not used for partitioning. The label updating rules proposed in this paper has rarely been studied in previous works.

Algorithms on large graphs. In recent years, a great deal of research has been dedicated to managing and mining large graphs. Typical problems that have attracted wide research interest include: label-constraint reachability queries [39], top-K substructure pattern mining [40], approximate subgraph search [41], best cluster finding [42], subgraph query optimization [43], etc. However, most of these algorithms or solutions are not designed for a general-propose distributed graph platform. The results reported in these works were primarily obtained on a single machine, and the largest graph ever reported has only millions of nodes. Quite recently, some papers have proposed some scalable solutions for spectral analysis [44], pattern inference [28], and max flow [45] on billion-node graphs. However, these solutions generally are built upon MapReduce instead of a general-purpose distributed memory system. And partitioning algorithms for a billion-node graph is not yet available.

8. CONCLUSION

In this paper, we propose a novel approach for partitioning billion-node graphs on a general-purpose distributed memory system. Our approach, which is called MLP, uses multilevel label propagation to iteratively coarsen a graph until the coarsened graph is small enough, and then uses a high-quality off-the-shelf partitioning algorithms to generate the final partitioning on the coarsened graph. Due to the semantic-awareness and efficiency of LP, MLP is more efficient, easily parallelized, and effective than existing approaches. Extensive experiments on large real graphs and synthetic graphs verify the efficiency and effectiveness of MLP. MLP successfully finds good partitionings on billion-node graphs with acceptable memory and time, which shows that MLP can scale up to billion-node graphs.

9. REFERENCES

[1] http://www.worldwidewebsize.com/.

[2] http://www.facebook.com/press/info.php?statistics.

[3] http://www.w3.org/.

[4] D. R. Zerbino et al., "Velvet: algorithms for de novo short read assembly using de bruijn graphs." *Genome Research*, vol. 18, no. 5, pp. 821–9, 2008.

[5] B. Kernighan et al., "An efficient heuristic procedure for partitioning graphs," *Bell Systems Journal*, vol. 49, pp. 291–307, 1972.

[6] C. M. Fiduccia et al., "A linear-time heuristic for improving network partitions," in *DAC '82*.

[7] B. Hendrickson et al., "The chaco user's guid, version 2.0," *Technical Report SAND94-2692, Sandia National Laboratories*.

[8] G. Karypis et al., "Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0," Tech. Rep., 1995.

[9] G. Karypis et al., "Analysis of multilevel graph partitioning," in *Supercomputing '95*.

[10] F. Pellegrini et al., "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *HPCN Europe '96*.

[11] G. Karypis et al., "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *J. Parallel Distrib. Comput.*, vol. 48, pp. 71–95, 1998.

[12] C. Chevalier et al., "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, pp. 318–331, 2008.

[13] A. Lumsdaine et al., "Challenges in parallel graph processing," *Parallel Processing Letters*, pp. 5–20, 2007.

[14] K. Munagala et al., "I/O-complexity of graph algorithms," in *SODA '99*.

[15] G. Malewicz et al., "Pregel: a system for large-scale graph processing," in *SIGMOD '10*.

[16] A. Abou-Rjeili et al., "Multilevel algorithms for partitioning power-law graphs," in *IPDPS '06*.

[17] B.Shao et al., "The trinity graph engine," *Microsoft Technique Report, MSR-TR-2012-30*.

[18] P. Erdős et al., "Graphs with prescribed degrees of vertices (hungarian)." *Mat. Lapok*, vol. 11, pp. 264–274, 1960.

[19] A.-L. Barabasi et al., "Emergence of scaling in random networks," vol. 286, pp. 509–512, 1999.

[20] A. Z. Broder et al., "Min-wise independent permutations," *Journal of Computer and System Sciences*, vol. 60, pp. 327–336, 1998.

[21] R. L. Graham et al., "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, 1969.

[22] J. Leskovec et al., "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *KDD '05*.

[23] M. R. Garey et al., "Some simplified np-complete problems," in *STOC '74*.

[24] D. S. Johnson et al., "Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning," *Oper. Res.*, vol. 37, pp. 865–892, 1989.

[25] T. N. Bui et al., "Genetic algorithm and graph partitioning," *Computers, IEEE Transactions on*, vol. 45, no. 7, pp. 841 –855, 1996.

[26] D. Gregor et al., "The parallel bgl: A generic library for distributed graph computations," in *POOSC '05*.

[27] http://neo4j.org/.

[28] http://www.infinitegraph.com/.

[29] B. Iordanov, "Hypergraphdb: a generalized graph database," in *WAIM'10*.

[30] http://en.wikipedia.org/wiki/Graph_database.

[31] http://infolab.stanford.edu/gps/.

[32] U. N. Raghavan et al., "Near linear time algorithm to detect community structures in large-scale networks," *Phys.Rev.E*, vol. 76, p. 036106, 2007.

[33] M. J. Barber et al., "Detecting network communities by propagating labels under constraints," *Phys.Rev.E*, vol. 80, p. 026129, 2009.

[34] I. X. Y. Leung et al., "Towards real-time community detection in large networks," *Phys.Rev.E*, vol. 79, p. 066107, 2009.

[35] X. Liu et al., "How does label propagation algorithm work in bipartite networks?" in *WI-IAT '09*.

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

[37] L. Subelj et al., "Unfolding communities in large complex networks: Combining defensive and offensive label propagation for core extraction," *Phys.Rev.E*, vol. 83, p. 036103, 2011.

[38] P. Boldi et al., "Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks," in *WWW '11*.

[39] R. Jin et al., "Computing label-constraint reachability in graph databases," in *SIGMOD '10*.

[40] F. Zhu et al., "Mining top-k large structural patterns in a massive network," in *VLDB '11*.

[41] A. Khan et al., "Neighborhood based fast graph search in large networks," in *SIGMOD '11*.

[42] K. Macropol et al., "Scalable discovery of best clusters on large graphs," in *VLDB '10*.

[43] P. Zhao et al., "On graph query optimization in large networks," in *VLDB '10*.

[44] U. Kang et al., "Spectral analysis for billion-scale graphs: discoveries and implementation," in *PAKDD'11*.

[45] F. Halim et al., "A mapreduce-based maximum-flow algorithm for large small-world network graphs," in *ICDCS '11*.

[46] D. Chakrabarti et al.,"R-MAT: A recursive model for graph mining," in *SDM' 04*.

[47] A. Lancichinetti et al., "Benchmark graphs for testing community detection algorithms," *Phys. Rev. E*, vol. 78, no. 4, pp. 046110 (2008).