A

Scalable and Efficient Flow-Based Community Detection for Large-Scale Graph Analysis

SEUNG-HEE BAE, DANIEL HALPERIN, and JEVIN WEST, University of Washington
MARTIN ROSVALL, Umeå University
BILL HOWE, University of Washington

Community detection is an increasingly popular approach to uncover important structures in large networks. Flow-based community detection methods rely on communication patterns of the network rather than structural properties to determine communities. The Infomap algorithm in particular optimizes a novel objective function called the map equation and has been shown to outperform other approaches in third-party benchmarks. However, Infomap and its variants are inherently sequential, limiting their use for large-scale graphs.

In this paper, we propose a novel algorithm to optimize the map equation called RelaxMap. RelaxMap provides two important improvements over Infomap: parallelization, so that the map equation can be optimized over much larger graphs, and prioritization, so that the most important work occurs first, iterations take less time, and the algorithm converges faster. We implement these techniques using OpenMP on shared-memory multicore systems, and evaluate our approach on a variety of graphs from standard graph clustering benchmarks as well as real graph datasets. Our evaluation shows that both techniques are effective: RelaxMap achieves 70% parallel efficiency on 8 cores, and prioritization improves algorithm performance by an additional 20%–50% in average, depending on the graph properties. Additionally, RelaxMap converges in the similar number of iterations and provides solutions of equivalent quality as the serial Infomap implementation.

Categories and Subject Descriptors: I.5.3 [**Clustering**]: Algorithms; F.1.2 [**Modes of Computation**]: Parallelism and Concurrency

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Community Detection, Graph Analysis, Parallelization, Prioritization

ACM Reference Format:
Seung-Hee Bae, Daniel Halperin, Jevin West, Martin Rosvall, and Bill Howe, 2016. Scalable and efficient flow-based community detection for large-scale graph analysis. *ACM Trans. Knowl. Discov. Data.* V, N, Article A (January 2016), 29 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Community detection in large graphs is emerging as a first-class technique in a number of applications: functional similarity in biological networks [Gavin et al. 2006; Guimera and Amaral 2005], collaboration communities in research networks [Girvan and Newman 2002], and the macro-structure of science through bibliometrics [Rosvall and Bergstrom 2011].

The community detection problem is difficult and is fundamentally different than the graph partitioning problem, which is known to admit a number of scalable approaches. Where graph

This work is sponsored in part by a subcontract from Pacific Northwest National Labs and by the National Science Foundation through the collaborative SI2-S2I2 grants 1216726, 1216754, 1216872, 1216879, 1216884. M. Rosvall was supported by the Swedish Research Council grant 2012-3729.
Author's addresses: S. Bae, the bulk of the research was done at the University of Washington, and he is currently affiliated with Western Michigan University; D. Halperin, the bulk of the research was done at the University of Washington, and he is currently affiliated with Google; J. West, Information School, University of Washington; M. Rosvall, Department of Physics, Umeå University, Sweden; B. Howe, Computer Science and Engineering Department, University of Washington.

DOI: http://dx.doi.org/10.1145/0000000.0000000

partitioning attempts to find an optimal binary split of a network, community detection attempts to simultaneously answer two questions: How many clusters should be used, and which vertices should be assigned to which clusters? While binary splits can be repeated for a given network, graph partitioning methods do not inherently identify the optimal number of splits.

Standard structural approaches to community detection use the intuition that intra-community connections are more common than inter-community connections. The *modularity* measure [Newman and Girvan 2004] scores a partitioning of a graph into communities or modules highly when the number of module-internal edges is higher than would be expected by chance. However, modularity optimization methods suffer from a "resolution limit" that depends on the size and connectivity of the network, and thus modularity cannot detect small clusters [Fortunato and Barthélemy 2007]. Further, Guimerà, Sales-Pardo, and Amaral showed that random graphs have high-modularity subsets, suggesting that structures found with this method may not be reliable structure in practice [Guimerà et al. 2004]. Spectral and min-cut techniques have been shown to be effective at finding structure at all scales, but exhibit a bias such that aggressive optimization of certain community score functions can destroy intuitive notions of cluster quality [Leskovec et al. 2010].

In response to these limitations of structure-based approaches, Rosvall et al. proposed a flow-based formulation of the community detection problem [Rosvall et al. 2009]. A flow in this context is modeled as a random walk through the graph, where each vertex is assigned a weight representing its visit frequency, an alternative formulation of the PageRank algorithm. Using this flow information, the community detection problem is to find a graph partitioning that maximizes the intra-partition flow and minimizes the inter-partition flow. A partitioning is scored by computing the length of a compressed representation of this flow according to the map equation [Rosvall et al. 2009]; better clusterings will have shorter codes. This formulation has been shown to capture some intuitive notions of community. While all two-level community-detection algorithms by construction must have some sort of resolution limit, it is orders of magnitudes less restrictive than for modularity [Kawamoto and Rosvall 2015]. The flagship algorithm and implementation for Rosvall et al.'s formulation [Rosvall et al. 2009] is called "**Infomap**."

It has been repeatedly demonstrated in third-party evaluations that Infomap finds better clusterings than competing community detection algorithms [Aldecoa and Marín 2013; Lancichinetti and Fortunato 2009]. In 2009, Lancichinetti and Fortunato [Lancichinetti and Fortunato 2009] did a comparative analysis with 12 different community detection algorithms, including Infomap, and two different benchmarks, by measuring normalized mutual information (NMI) between the planted partition of the benchmark datasets and the outputs of those algorithms. The study showed that Infomap outperforms other algorithms. In 2013, Aldecoa and Marín [Aldecoa and Marín 2013] did a more comprehensive study with 17 different community detection algorithms and three benchmarks, which consist of a more complicated benchmark, based on Relaxed Caveman (RC) structures, in addition to the two benchmarks in Lancichinetti and Fortunato's work [Lancichinetti and Fortunato 2009]. Infomap is a top-ranked community detection algorithm among 17 different algorithms with respect to the results with the three benchmarks in their study [Aldecoa and Marín 2013].

Although Infomap is a competitive community detection algorithm, it is sequential and cannot scale to handle the graphs with millions and billions of edges that are becoming commonplace. It requires 6 to 8 hours to detect communities of a graph with around 60 million vertices and 150 million edges—a relatively long runtime for a relatively small graph. Parallel algorithms are therefore crucial to enabling the broad application of this clustering technique. Our work represents the first such algorithm.

In this paper, we present a new algorithm, named *RelaxMap*, which optimizes the same flow-based objective as Infomap but provides two strong improvements. First, RelaxMap uses with a parallel search procedure that relaxes serial consistency constraints to improve scalability. The core approach is to relax concurrency assumptions to significantly reduce lock contention: a) decisions to move vertices between modules are made in parallel without locking for checking consistency, but b) the algorithm acquires a global lock before applying each move for assuring the shared module status and the objective function values are updated consistently. We show that these techniques offer

significantly improved performance on modern multicore machines while achieving as good quality scores and similar convergence rates as the sequential algorithm.

Second, RelaxMap employs a prioritization heuristic that prunes from the search space those vertices that are unlikely to significantly contribute to the optimization goal. This *prioritized* searching mechanism improves efficiency by a factor of $1.2\times$–$1.5\times$, and still provides quality outputs as good as the original sequential algorithm in our experiments. Prioritization applies to flow-based clustering algorithms, including the serial Infomap algorithm; prioritization can be used with and without parallelism, and its improvements are cumulative. The algorithm that combines parallelism and prioritization, which we call *prioritized RelaxMap*, offers quality as good as RelaxMap and more efficient runtime performance than RelaxMap alone.

We offer the following contributions:

— We describe a novel parallel algorithm for graph clustering called *RelaxMap* that parallelizes the optimization of flow-compression for community detection. To the best of our knowledge, this is the first parallel algorithm for flow-based community detection.
— We propose a prioritization strategy that avoids handling vertices that do not significantly improve the objective function. This strategy can improve performance by a factor of 1.5 for all parallel and serial algorithms considered.
— We present an experimental evaluation of the proposed algorithms against benchmark graphs [Lancichinetti et al. 2008], demonstrating that RelaxMap and the prioritized RelaxMap find solutions that are measurably similar to the "planted partitions" on which the benchmarks are based, but delivers significantly better performance.
— Using a variety of real graphs (where the true clusters are unknown), we show that the proposed parallel algorithms can achieve quality scores similar to the sequential algorithm (up to the variance associated with different random seeds), while achieving 70% parallel efficiency and significant performance improvement.

This paper is an extended version of an earlier publication on RelaxMap [Bae et al. 2013]. In this paper, we provide more thorough evaluation of RelaxMap and additionally propose an efficient prioritization scheme to enhances both RelaxMap and other techniques. Section 2 summarizes the flow-based community detection algorithm and the objective function called "the map equation." We describe the proposed parallelization and prioritization methods in Section 3 and Section 4, respectively. In Section 5 we evaluate both techniques, independently and when combined, on benchmark and real graph datasets. Finally, we discuss related work in Section 6 before presenting conclusions and future work in Section 7.

2. THE MAP EQUATION

In this section we briefly summarize the principles behind the map equation, a flow-based, information-theoretic objective function for graph community detection. We then describe the core Infomap algorithm for optimizing the map equation over possible clusterings. Following the literature, we will refer to a cluster as a *module* in this paper. Infomap is designed for cluster-weighted directed networks, which arise in many important applications, and has been shown to outperform other methods in objective third party benchmarks [Aldecoa and Marín 2013; Lancichinetti and Fortunato 2009]. Infomap has many generalizations, such as clustering into hierarchically nested modules [Rosvall and Bergstrom 2011] and operation on higher-order Markov dynamics [Rosvall et al. 2014], but here we focus on two-level clustering and first-order Markov dynamics. In the next section, we will show how RelaxMap can optimize the map equation in a parallel fashion.

The term "map" in *map equation* is an analogy to road maps, where the regions can be defined by traffic density: intra-region traffic is more dense than the inter-region traffic. Optimizing the map equation over possible module assignments produces modules with this same property: a random walker through the graph will tend to spend more time within a module than moving between modules. Flow is represented as a directed edge weights, which in many applications can be interpreted directly

ALGORITHM 1: Pseudo code for the serial Infomap algorithm [Rosvall et al. 2009]

input: Network $G = (V, E)$, where V = set of N vertices, E = set of edges. Minimum quality improvement threshold τ

Run PageRank to calculate *vertex visit rate* for each vertex.
$\mathsf{M} = \{m_i = \{v_i\} \mid v_i \in V\}$
$L = L(\mathsf{M})$ in (2)

repeat
  $L_{prev} = L$
  R = random sequence of integers 1 to N
  for i = 0; i < N; i++ **do**
    m_{new} = bestNewModule(M, $v_{R[i]}$);
    Move $v_{R[i]}$ to m_{new} module, and update M and L.
  end for
until $L_{prev} - L < \tau$

return M

as traffic flow (e.g., airplanes flying between airports [Menon et al. 2004]), or it may be interpreted as information flow (i.e., a citation graph in the scientific literature [West et al. 2010])

The map equation expresses this objective via information theory. Any regularity in data can be used for compression, such that the compression ratio one achieves can be interpreted as a measure of one's ability to find regularity in the data. The map equation takes advantage of this *minimum description principle* by measuring the description length of a random walker (or of real flow) on a network with a *modular codebook* structure [Rosvall et al. 2009]. Each one of $|\mathsf{M}|$ modular codebooks describes movements between vertices assigned to the corresponding module.

Say there is a module m among $|\mathsf{M}|$ modules. Let α be a member of module m, $\alpha \in m$, and let the *vertex visit rate* of vertex α be p_α. The vertex visit rates can be computed by finding PageRanks of all vertices and dividing each PageRank by the sum of PageRanks of all vertices. For instance, the vertex visit rate of vertex α, p_α, is equal to $pg(\alpha)/\sum_{v\in V} pg(v)$, where $pg(\alpha)$ and $pg(v)$ are the PageRanks of vertices α and v. Also, if the probability for the random flow to exit module m is defined as $q_{m\curvearrowright}$, then the overall probability for the flow in module m, denoted $p^m_{\circlearrowleft}$, is equal to $\sum_{\alpha\in m} p_\alpha + q_{m\curvearrowright}$, the sum of the vertex visit rates of members of the module m and the exit probability of module m. With the probability distribution of a codebook of module m, denoted $\mathscr{P}^m$, the average description length of a module codebook for module m, $H(\mathscr{P}^m)$, can be calculated as follows:

$$H(\mathscr{P}^m) = -\frac{q_{m\curvearrowright}}{p^m_{\circlearrowleft}} \log\left(\frac{q_{m\curvearrowright}}{p^m_{\circlearrowleft}}\right) - \sum_{\alpha\in m} \frac{p_\alpha}{p^m_{\circlearrowleft}} \log\left(\frac{p_\alpha}{p^m_{\circlearrowleft}}\right) \tag{1}$$

according to Shannon's source coding theorem [Shannon 1948a,b].

Moreover, a single *index codebook* describes movements between the module codebooks. With $\mathscr{Q}$ for the probability distribution of module entering rates $q_{m\curvearrowright}$, the average description length is given by the entropy $H(\mathscr{Q})$. The frequency of use of the index codebook is $q_{\curvearrowright} = \sum_{m\in\mathsf{M}} q_{m\curvearrowright}$. Also, $p^m_{\circlearrowleft}$ represents how many times the module codebook of module m is used. Taken together, the map equation, Eq. (2), measures the average description length $L(\mathsf{M})$ given modular assignments M.

$$L(\mathsf{M}) = q_{\curvearrowright} H(\mathscr{Q}) + \sum_{m\in\mathsf{M}} p^m_{\circlearrowleft} H(\mathscr{P}^m). \tag{2}$$

Minimizing the map equation over all possible module assignments gives the optimal modular structure for describing movements on the network, and therefore reveals important structures with respect to the dynamics on the network [Rosvall et al. 2009].

Algorithm 1 illustrates the *core algorithm* of the fast stochastic and recursive search algorithm implemented in Infomap [Rosvall et al. 2009]. The core algorithm proceeds in two phases:

ALGORITHM 2: Pseudo code for a naïve lock-free algorithm

```
1: for (in Parallel) i = 0; i < N; i++ do
2:     M_new[R[i]] = bestNewModule(M, v_R[i]);
3: end for
4: M = M_new, L = L(M_new)
```

— Phase 1: (line 2) The visit probability (rank) of each vertex is computed in terms of the network flow.
— Phase 2: (lines 5- 12) The space of possible modularizations is greedily searched. The initial modules are singletons — one module per vertex. For each vertex v, the call **bestNewModule**(M, v) (line 9) considers each possible move of v to a neighboring module and greedily selects the one that reduces $L(\mathsf{M})$ by the greatest amount. This function computes the new exit probability that would result from each move, and uses the value to compute the change in L. The move that generates the greatest improvement is selected, and the target module s returned. The algorithm stops when the change in the *minimum description length (MDL)* score in each iteration is less than a minimum quality improvement threshold, $L_{prev} - L < \tau$.

In the search procedure for the best new module of a vertex v (line 9), the algorithm calculates the total *in-flow* and total *out-flow* between the vertex v and its neighbor modules (i.e., the set of modules to which any of its neighbors belong). Finally, the improvement in the MDL for each candidate move can be calculated from the measured total in-/out-flow information. The algorithm assigns the vertex v to whichever new module minimizes the MDL.

After the core algorithm converges to a solution, Infomap algorithm generates a new network based on the current module assignment by aggregating vertices assigned in each module as a vertex. Then, it revisits **Phase 2** with the aggregated network. This aggregation method iterates until no meaningful quality improvement is achieved. This aggregation step was used by the Louvain method [Blondel et al. 2008] for modularity based community detection implementation, and the Infomap algorithm uses this technique for finding communities by minimizing the map equation.

In addition to the core algorithm, Rosvall *et al.* applied two heuristics for improving the mapping accuracy to the core algorithm: (1) *Sub-module movements* and (2) *Single-node movements* heuristics [Rosvall et al. 2009]. For the sub-module movements heuristic, we extract sub-modules from each module by running Infomap algorithm within a network built by members of each module only, and then combine vertices assigned in each sub-module as a vertex, which is similar to the aggregation step mentioned above. We can think of both heuristics as the adjustment of the unit of each movement in the Phase 2 of the core algorithm. In fact, the algorithm repeats the following in Phase 2: adjusting the movement unit (e.g. single-node, sub-module, or aggregated-node) and iteratively searching new module for each movement unit, so-called "Super-Step," until the algorithm meets the stop condition.

We also implement the aggregation step and the Sub-module movements and Single-node movements heuristics in the proposed parallel algorithms, but the core algorithm dominates the elapsed time so we focus the core algorithm in our paper.

Details of the map equation and the Infomap community detection algorithm are available in the original paper [Rosvall et al. 2009] and a dynamic visualization of the technique is available online[1].

3. PARALLELIZING FLOW-BASED COMMUNITY DETECTION

To parallelize Algorithm 1, we observe that *Phase 1* of that algorithm is the same as PageRank [Brin and Page 1998] for which there are many parallel and distributed implementations [Ekanayake et al. 2010; Gleich and Zhukov 2005; Low et al. 2012; Zaharia et al. 2012]. The remainder of this section describes how to parallelize *Phase 2*.

[1] http://www.mapequation.org

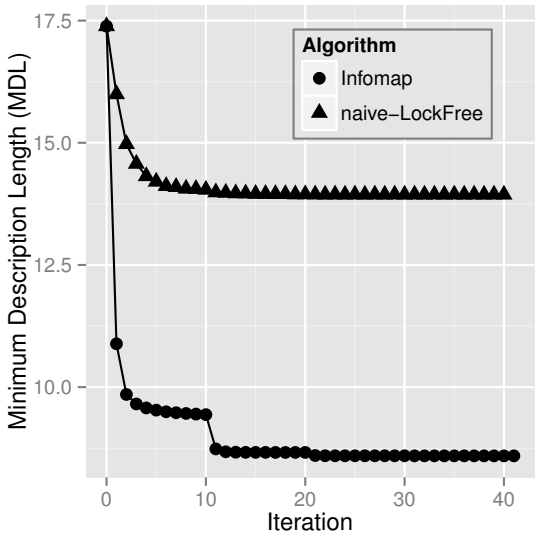


Fig. 1: The Minimum Description Length (MDL) with respect to the iteration number of the naïve lock-free parallel scheme with *web-Stanford* dataset in Table II.

To compute the exact same process as the sequential algorithm in parallel, each thread attempting to move a vertex must acquire a lock on the module to which that vertex belongs as well as locks on all neighboring vertices (including itself) and all neighboring modules. This direct approach to parallelization requires a prohibitive amount of synchronization between threads, and our initial experiments showed that this approach would offer no siginficant benefit over the serial algorithm. As a result, we do not discuss this "lock-intensive" approach further.

Another approach to parallelization is to simply ignore the locks altogether, as explored in several recent papers on optimization (c.f., [Niu et al. 2011]). The reasoning is that the improvement in scalability offsets the loss of quality improvement on each step, such that the overall convergence rate is faster. However, this approach will not necessarily converge at all, depending on the problem. We evaluate this *naïve* approach as a baseline method.

3.1. The Parallel Algorithm

We can design a naïve lock-free scheme by checking each vertex independently in parallel and simply ignoring any interaction between neighboring vertices. The naïve lock-free algorithm runs steps 8 through 11 of Algorithm 1 for each vertex in an independent thread, identifying a new module for the vertex being considered based only on the information from the previous iteration M. In contrast, the serial algorithm considers only one vertex at a time, such that each move has access to the results of all prior moves. This naïve parallel algorithm is summarized in Algorithm 2. In this scheme, each candidate move is run independently since the previous module assignment for all vertices is already fixed, so the algorithm does not need to use any synchronization to make decisions for several vertices in parallel. This simple parallel algorithm can be implemented using an additional module assignment array, denoted M_{new}, to store each movement decision at the current iteration, and updating the module array and corresponding module values, M, based on the stored movement decisions, after all the vertices finish their movement decisions at current iteration.

This approach will achieve perfect parallelism for the most expensive part of this application, from line 8 to line 11 in Algorithm 1, which has complexity $\mathcal{O}(\mathbf{E})$. However, since the update for each movement at line 10 in Algorithm 1 is postponed until the naïve algorithm completes all parallel

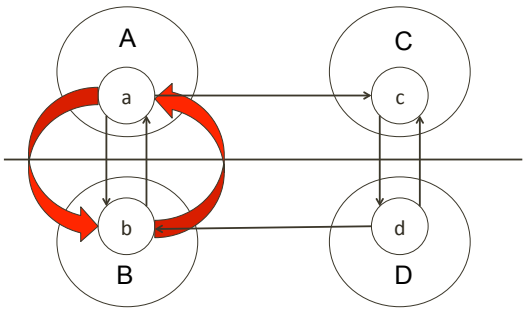


Fig. 2: An example of the cyclical movement in the naïve lock-free parallel algorithm; 4-vertex graphs running in 2-way parallelism. Assume one thread examines vertices **a** and **c** and a different thread examines vertices **b** and **d**. On the first iteration, vertex **a** will be moved to module **B** and vertex **b** will be moved to module **A**, which will have no effect on the module structure.

moves, it requires an additional procedure to update status values for each module and the new MDL at the end of each iteration as shown at line 4 in Algorithm 2.

The naïve method described is not competitive with the sequential method, however, as seen in Figure 1. In Figure 1, the naïve algorithm produces clusters with much worse quality than the sequential Infomap with a real-world datasets, *web-Stanford*, in Table II. To see why, consider the simple 4-vertex network in Figure 2. In the initial stage, each vertex is assigned to its own module, which we write as $a \in A, \ldots, d \in D$, where $a, \ldots, d$ represent vertex IDs and $A, \ldots, D$ are module IDs. One thread moves a to B, while an independent thread moves b to A, based on the (stale) network flow and the module information from the previous iteration. The two moves offset each other, causing the algorithm to make cyclical movements with no net improvement in quality, and converge prematurely. Figure 1 shows an example execution where the naive parallel algorithm converges prematurely, a behavior we saw frequently.

To improve on the naïve method, we propose an algorithm **RelaxMap** based on the assumption that real networks are typically sparse in practice. In a sparse network, the movement of a single vertex will typically only affect a small subset of the graph. As a result, if we consider a small number of random vertices concurrently, they are unlikely to influence each other, and the problems with the naïve method will be minimized.

In RelaxMap, each of p threads examines a vertex independently, then acquires a lock to apply the winning move and update the module information. When considering the p vertices, the move decisions are made with stale module information from the previous parallel round. In addition, the RelaxMap parallel algorithm avoids the cyclical-movement problem, shown in Figure 2. To see why, consider the case where one thread p_1 examines a and c and a different thread p_2 examines b and d as in Figure 2. Say two threads worked on a and d concurrently, deciding to move a to B and d to C. Then, as the two threads begin examining c and b respectively, they have access to the current module assignments: $a, b \in B$ and $c, d \in C$. So thread p_2 would decide to keep b in B since it knows a is also in B, and no cyclical movement occurs.

There is still some possibility of encountering a problem: the two threads could examine a and b at first concurrently, moving a to B and b to A, but the probability of this case will be very low if the number of vertices is large. And, even if some cyclical movements happen during one iteration, they will be fixed at later iterations with high probability if the MDL decreases enough to continue.

To make this specific, let us assume among N vertices, there are two strongly connected vertices. If we run p-way parallel RelaxMap ($p \ll N$ in general), the probability that those two vertices are examined concurrently is $(p-1)/N \approx p/N$. If the algorithm stops after t iterations, then the probability that those two vertices executed at the same time through all the t iterations is $(p/N)^t$ because the algorithm searches new modules for vertices in random order at each iteration. The two

ALGORITHM 3: The core-algorithm of the RelaxMap

1: **for** (**in Parallel**) i = 0; i < N; i++ **do**
2: m_{new} = bestNewModule(**M**, $v_{R[i]}$);
3: acquire a lock for the updates.
4: Move $v_{R[i]}$ to m_{new} module, and update M and L.
5: release lock.
6: **end for**

vertices will not be run concurrently at least once among t iterations in $1-(p/N)^t \approx 1$ probability. Therefore, we argue that the cyclical movement problem will not be an issue in practice.

The main difference between the naïve lock-free algorithm and the RelaxMap is the following: RelaxMap considers batches of p vertices concurrently, while the naïve method considers batches of N vertices concurrently. Since $p \ll N$, the probability that two or more of the p vertices are interdependent is low, and the algorithm can converge almost as quickly as the serial version.

The RelaxMap algorithm is described in Algorithm 3. One specific feature of the RelaxMap (and the Infomap) algorithm is that the MDL value, L, and the statistics of modules (i.e. exit-probability $q_{m\curvearrowright}$ and sum of the visit-rates, $\sum_{\alpha\in m} p_\alpha$, for each module m) are always correct with respect to the current module assignment for the consistent and efficient calculation of L and correct algorithmic procedure. For achieving the correctness of those values, we use a global lock in lines 3 and 5 of Algorithm 3. Since we use a global lock for updating module information of a vertex movement in RelaxMap, the updating step would be a bottle neck for the parallel performance of RelaxMap if the moving vertex were a very high degree vertex. Even though using locking mechanism for consistency might be expensive in some high-degree vertices cases, it will reduce the overall number of iterations for the convergence to the final solutions, which will finally reduce the run-time of the algorithm. We tested RelaxMap without acquiring the global lock in lines 3 and 5 of Algorithm 3 to see how much it affects the algorithm. This approach was ineffective, for two reasons. First, the inconsistency caused the algorithm to converge slower than when the consistency is achieved, in some cases slower than the sequential algorithm. Second, the incorrect module information results in incorrect counting for active module numbers, which causes race conditions and subsequent memory faults.

4. PRIORITIZED FLOW-BASED COMMUNITY DETECTION

In the last section, we showed how to scale the clustering process by considering module movements for many nodes in parallel. Now, we will show how to further optimize the search process by prioritizing which nodes we consider. As a motivating example, consider Figure 3, which shows the number of vertices the algorithm chose to move in each of the first 10 iterations for the Live Journal dataset. In the first iteration, more than 80% of vertices (more than 4 million vertices among 4,847,571 vertices as shown Table II) are moved to new modules— that most nodes move is expected, since each vertex starts in its own module. However, the graph also shows that in each subsequent iteration, many fewer vertices move; noting that the y-axis is logarithmic, the number of moved vertices decreases nearly exponentially. Despite this property, both Infomap Algorithm 1 and RelaxMap (as described thus far), consider every vertex in each iteration. Searching all the vertices in each iteration will not be a significant runtime cost for small datasets, but for large datasets, such as millions of vertices, it will be a significant runtime cost without much gain per each iteration after a few iterations are done. Intuitively, an optimization algorithm should be able to exploit this property to gain efficiency: if an algorithm could identify the nodes that are likely to move between modules, it might do much less work.

Phase 2 of the flow-based community detection algorithm in Algorithm 1 is an iteration mechanism of the following two-steps:

— Find new module for a vertex where the objective function is maximally optimized. (Line 9)
— Apply new module for the vertex and update module status. (Line 10)

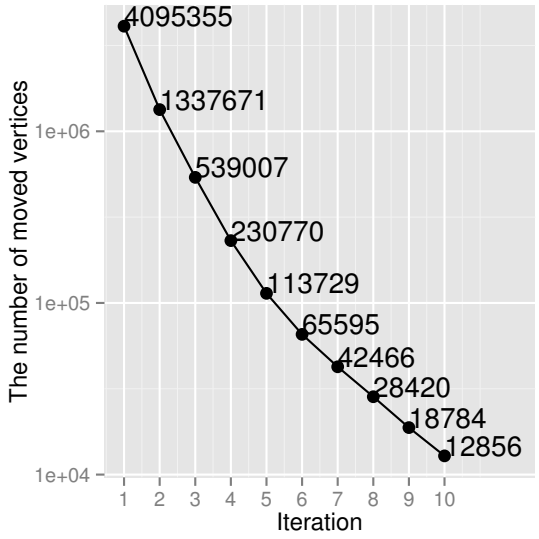


Fig. 3: The number of moved vertices among 5 million vertices in each iteration of the first 10 iterations when we run Algorithm 1 with the *soc-LiveJournal1* dataset in Table II. Note that the y-axis of the graph is log-scale and the number of moved vertices drops down almost exponentially.

After finishing the above two-steps for all vertices, we need to check whether the algorithm is converged to a solution or not. It will iterate for a new search step, unless it has converged to a solution, which is determined by measuring quality improvement ($L_{prev} - L < \tau$). The original Infomap algorithm in Algorithm 1 assumes that each iteration is independent, so all of the vertices are examined again for the next iteration.

However, if we look into the details of the searching mechanism in Algorithm 1, we can find that the new community for a vertex v will be affected by its neighbor vertices' community-assignments and the changes of the neighbor communities. On the other hand, the changes of long-distanced vertices from a vertex will not affect finding new community for the vertex at all. In other words, if a vertex v changed its community, then it affects the assignment of the community for only a subset of vertices.

Figure 4 illustrates the possible priority relation resulted from a vertex movement. When a vertex v moved from old module M_a to new module M_b, each vertex could belong to one of three different groups:

— The direct neighbors of the vertex v. The *orange* vertices n_1, n_2, and n_3 in Figure 4 are direct neighbors of v. We refer to this group as **Neighbors**.
— The members of the source and target module that are not direct neighbors of v. The *pink* vertices $m1$ and $m2$ in Figure 4 are members of M_a or M_b but are not neighbors of v. We refer to this group as **Mod-Members**.
— The neighbors of the vertices in Mod-Members that are not in the source or target module. The *green* vertices $mn1$, $mn2$, and $mn3$ in Figure 4 are neighbors of vertices in M_a or M_b, but are not themselves in M_a or M_b. We refer to this group as **Mod-Neighbors**.

The *yellow* vertices c_1, c_2, and c_3 in Figure 4 are none of the above categories. In Figure 4, it is clear that the movement of vertex v from M_a to M_b will not make any notable change related to the calculation of the map equation, Eq. (2), from c_1, c_2, and c_3 vertices. Vertices in the other three

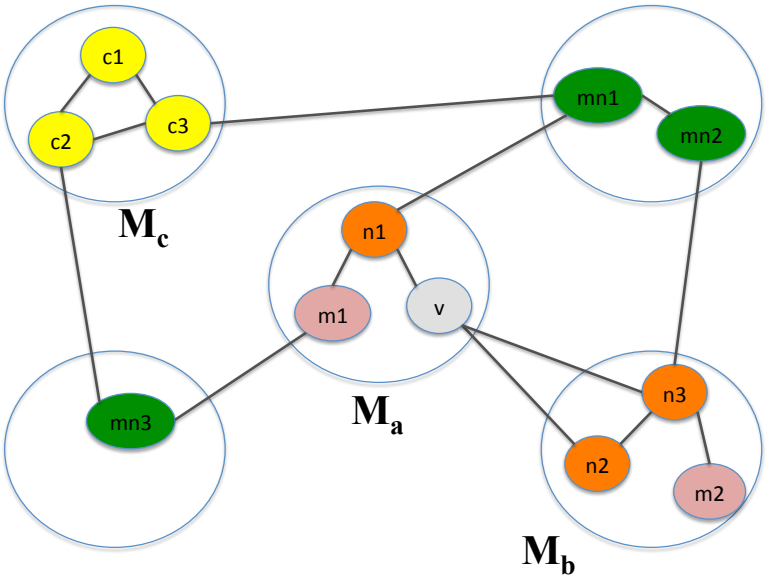


Fig. 4: An example of a movement of a vertex v from old module M_a to new module M_b and possible prioritized vertices. The different colors represent different prioritized groups: the direct neighbors of v in **orange**, the member of old module M_a and new module M_b in **pink**, and the neighbors of the vertices in M_a and M_b in **green**. The vertices that are unrelated to a vertex v are in **yellow**.

groups, i.e. Neighbors, Mod-Members, and Mod-Neighbors, are likely affected for searching new modules for them.

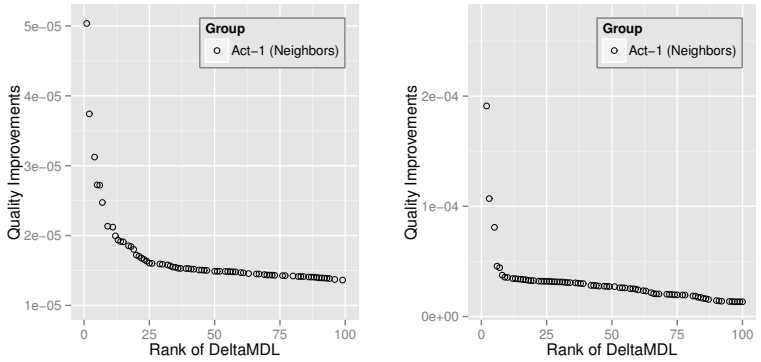
Among those three different active groups, we can give the priority based on the possibility of impact for quality improvement. Possible priority for getting maximally optimized movement in next iteration from a movement of a vertex v would be following:

— **Neighbors**: The direct neighbors might be mostly affected from the movement, since they are directly connected the moved vertex v. Each flow connected via v directly affects the flow between vertices in this group and their neighbor modules are changed due to this move.
— **Mod-Members**: Among members of the old module, some of them might be in the old module due to the moved vertex v, so they should be investigated the possible moves in the next iteration. Similarly, some of the members in the new module might be better to be separate with it, so they should be tested as well.
— **Mod-Neighbors**: Some neighbors of the members of the old module would be in the old module if the moved vertex v were not there. On the other hands, some neighbors of the members of the new module would be in the new module if it were there. Thus, we need to check them whether there will be some improvement.

Based on the above rationale, we proposed a prioritized searching mechanism for finding possible movements of vertices in each iteration, in that, we investigate only the vertices in some top-priority groups, which activated at the previous iteration.

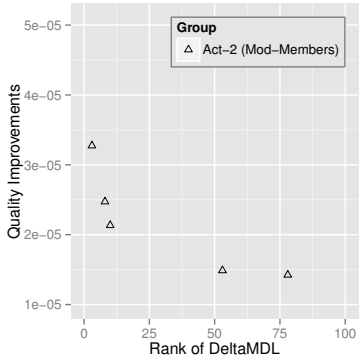
4.1. Effect of Different Prioritization Schemes

We examined the effectiveness of the three different active groups of vertices, in 9 of the first 10 iterations, except the first iteration. Figure 5 illustrates the MDL improvement for top 100 moved vertices sorted by its rank. Each point is labeled by its membership in one of the active groups. The data was collected from one specific representative iteration in the experiments with *web-BerkStan*

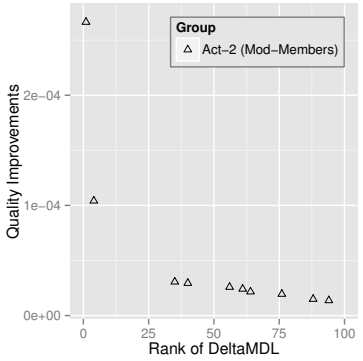


(a) web-BerkStan (Neighbors)

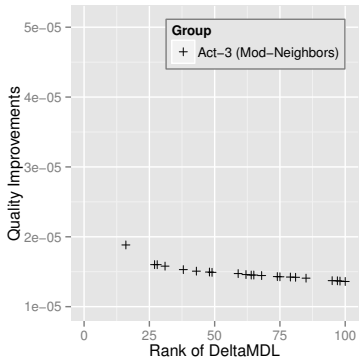
(b) web-Stanford (Neigbhors)



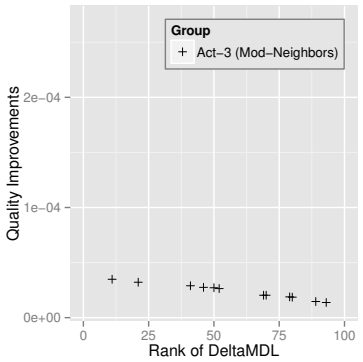
(c) web-BerkStan (Mod-Members)



(d) web-Stanford (Mod-Members)



(e) web-BerkStan (Mod-Neighbors)



(f) web-Stanford (Mod-Neighbors)

Fig. 5: The MDL improvement ($\Delta(MDL)$) corresponding to the rank of the top 100 ranks. Each active group is expressed in different shape. Most of the top 100-ranked improvements ($\Delta(MDL)$) occur in the **Neighbors** activity group (circles). This illustrates that we could achieve most of the quality improvements by investigating the **Neighbors** activity group only.

Table I: When a vertex is moved from one module to another, its direct neighbors contribute more than 95% of the quality improvement in the next round. Members of the same module and members of neighboring modules also contribute, but to a much lesser extent. Therefore, prioritizing the search to the direct neighbors is effective.

Dataset	Neighbors (%)	Mod-Members (%)	Mod-Neighbors (%)	Others (%)
web-BerkStan	**95.53**	1.81	2.62	0.04
web-NotreDame	**94.07**	3.03	2.86	0.04
web-Stanford	**96.03**	1.33	2.56	0.08

ALGORITHM 4: Pseudo code for the *Phase 2* by prioritization method

```
1: Â = V;
2: repeat
3:    L_prev = L
4:    A = Â;
5:    Â = ∅;
6:    R = randomSequentialOrder(A);
7:    for i = 0; i < R.size(); i++ do
8:       m_old = M[v_R[i]];
9:       m_new = bestNewModule(M, v_R[i]);
10:      if m_new != m_old then
11:         Move v_R[i] to m_new module, and update M and L.
12:         Add all neighbors of v_R[i] to Â.
13:      end if
14:   end for
15: until L_prev − L < τ
```

and *web-Stanford* datasets. The plot indicates that the vertices in the **Neighbors** group (in Figure 5-(a),(b)) — the vertices that are direct neighbors of vertices that moved in the previous iteration — are highly ranked, and therefore contributing most significantly to improving the quality score. The vertices in the groups **Mod-Members** and **Mod-Neighbors** are not as highly ranked.

We performed a similar analysis with the *web-NotreDame* dataset, and the result was similar to Figure 5. These results suggest that most of the important moves involve vertices in the **Neighbors** group, and that prioritizing this group can dramatically reduce the work without sacrificing significant quality.

Table I shows the portion of the quality improvement by each group in percentage among all movements of original method, in 9 of 10 iterations (except the first iteration), with respect to three different real-world datasets. $\Delta L(M)$ is used as a measure of the quality improvement. The **Neighbors** group covers about 95% of the overall quality improvement, and the other active groups covers around 5%. The improvement by the vertices in none of the active groups were negligible (less than 0.1%).

As shown in Table I, **Neighbors** group is dominant for the quality improvement. We may still cover the remaining 5% quality improvement by activating Mod-Members and Mod-Neighbors groups, but this strategy involves significantly more computation time for relatively little improvement in quality. Also, even if we designed to activate only the vertices in the Neighbors group for the proposed prioritized scheme, it would be still possible that some of the vertices in the Mod-Members or Mod-Neighbors group can be activated as the vertices in the Neighbors group in later iterations, which would result in making up the 5% quality improvement. Based on this analysis, we chose to prioritize only **Neighbors** group in our proposed method. In Section 5.2, we can see that the prioritization method results in no measurable quality loss for every tested dataset and environment.

Table II: Network datasets used for evaluating parallel RelaxMap and PriorMap algorithms.

Dataset	Number of vertices	Number of edges	Avg. degree	Max degree
directNet-1k	1,000	19,849	39.70	69
directNet-5k	5,000	98,313	39.33	69
directNet-10k	10,000	196,414	39.29	69
directNet-50k	50,000	1,013,314	40.53	220
directNet-100k	100,000	2,040,074	40.80	220
directNet-200k	200,000	4,070,634	40.70	220
directNet-400k	400,000	8,152,828	40.76	220
directNet-800k	800,000	16,303,395	40.76	220
directNet-1600k	1,600,000	32,657,712	40.82	220
directNet-3M	3,000,000	61,128,636	40.75	220
web-BerkStan	685,230	7,600,595	22.18	84,290
web-NotreDame	325,729	1,497,134	9.19	10,721
web-Stanford	281,903	2,312,497	16.41	38,626
uk-2002	18,484,117	298,113,762	32.26	194,956
soc-LiveJournal1	4,847,571	68,993,773	28.47	22,887
soc-Pokec	1,632,803	30,622,564	37.51	20,518
Twitter	41,652,230	1,468,365,182	70.51	3,081,112
wiki-Talk	2,394,385	5,021,410	4.19	100,032

Algorithm 4 shows the pseudo code of the proposed prioritization method of the community detection algorithm. A and $\widehat{A}$ represent the sets of the current activated vertices and active vertices for next iteration, correspondingly. M shows the current module assignment and the module status information.

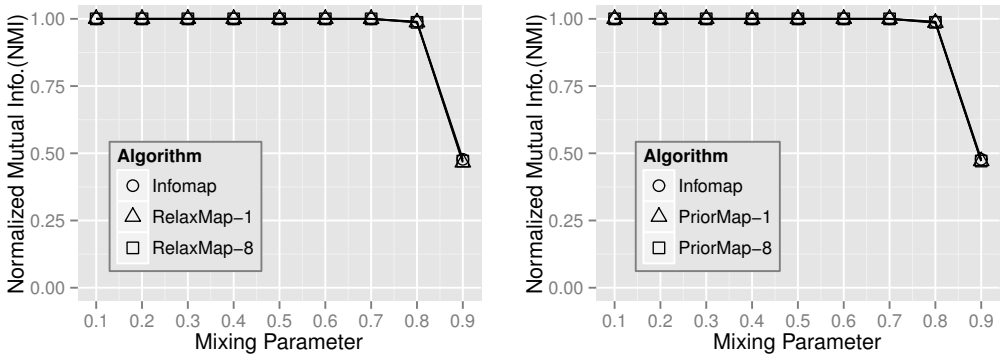
In addition, we combine the prioritization method into the proposed parallel algorithm Section 3, called *prioritized RelaxMap* (hereafter, **PriorMap** in short). Our experimental results in Section 5 shows that **PriorMap** is much more efficient than the regular independent searching mechanism, and still provides compatible outputs in terms of the quality.

5. EXPERIMENTAL ANALYSIS

We study *RelaxMap* and *PriorMap* experimentally to answer three questions: First, *do RelaxMap and PriorMap produce clusters that match the quality of state-of-the-art flow-based clustering (Infomap), despite RelaxMap's relaxed consistency constraints?* We evaluate both methods on benchmark graphs for which the known clusters ("planted partitions") are available, then consider a set of real graph datasets (Section 5.2). Second, *do RelaxMap and PriorMap significantly improve performance over the serial algorithm?* We evaluate performance over the same graph datasets using three different machines and up to 60 cores (Section 5.3). Third, *is PriorMap more efficient than RelaxMap without output quality loss?* We evaluate performance over the same graph datasets by PriorMap and RelaxMap and compare the average overall runtime (Section 5.4). Our results in this section answer these questions in the affirmative: communities identified by the proposed RelaxMap and PriorMap match the quality of the state-of-the-art, RelaxMap achieves 70% parallel efficiency (8-way parallelism) in the machines tested, and PriorMap is 1.2 to 1.5 times more efficient than RelaxMap in most cases.

5.1. Experimental Setup

Algorithms. We compare **RelaxMap** and **PriorMap** against **Infomap**, the state-of-the-art serial algorithm to optimize the map equation [Rosvall et al. 2009]. We used the open-source implemen-



(a) RelaxMap (b) PriorMap

Fig. 6: The average normalized mutual information (NMI) as a function of the mixing parameter comparing **Infomap** with (a) **RelaxMap** or (b) **PriorMap** for the *directNet-5k* benchmark dataset, showing high similarity between **RelaxMap**, **PriorMap** and the serial **Infomap** algorithm. These overlapping plots describe that the proposed RelaxMap and PriorMap produce output with the same quality as Infomap. The NMI is a measure of similarity between each result of the algorithm and the planted partitioning of the corresponding benchmark graph. The NMI ranges from 0 to 1, with 1 representing an exact match between the planted partition and the computed cluster. The average NMI scores are almost same for **Infomap**, **RelaxMap**, and **PriorMap** using the *directNet-5k* dataset with single-thread running and 8-way parallel running results. Since the range of NMI scores are less than 0.02 with all of the tested cases, we ignore the error-bar of each experiment for visual clarity. We see the same effect at other scales using the *directNet-1k* and *directNet-10k* datasets.

tation of Infomap from Rosvall *et al.* (www.mapequation.org). We implemented RelaxMap and PriorMap[2] using OpenMP [OpenMP Architecture Review Board 2008] for shared memory parallel environments, and ran both parallel algorithms with up to p concurrent threads, where p is the number of cores on the test machine.

Datasets. We used three different benchmark graphs from a standard clustering benchmark network generator, directNet [Lancichinetti et al. 2008] to evaluate clustering when a planted partition of a graph is available to evaluate against. We follow the parameters given by Lancichinetti *et al.* [Lancichinetti and Fortunato 2009, Section VI-A] to generate graphs of 1000, 5000, and 10000 vertices with "small" communities between 10 and 50 vertices each with different mixing parameters. We also generate larger benchmark graphs of 50k, 100k, 200k, 400k, and up to 3 million (directNet-3M) vertices with communities between 20 and 1000 vertices with 0.5 mixing parameter in order to evaluate weak-scale parallel performance of the proposed algorithms. The benchmark graphs are generated based on the *planted l-partition* model [Condon and Karp 2001], so we interpret this planted partition of each generated graph as the true community structure of the graph. We also used six real network datasets from the Stanford Network Analysis Project (SNAP)[3] [Leskovec and Krevl 2014], *uk-2002* dataset from Laboratory for Web Algorithmics[4] [Boldi and Vigna 2004], and *twitter* dataset from the twitter dataset website[5] [Kwak et al. 2010]. You can find a summary of salient properties of the datasets in Table II, and detailed information on the corresponding websites.

[2]You can find source code for our algorithms at: https://github.com/uwescience/RelaxMap
[3]http://snap.stanford.edu/data/index.html
[4]http://law.di.unimi.it/datasets.php
[5]http://an.kaist.ac.kr/traces/WWW2010.html

Table III: The final quality scores for **Infomap**, **RelaxMap** (8 threads), and **PriorMap** (8 threads) using the six real-world datasets in Table II. Each average, minimum, and maximum MDL score from 10 different runs with different random seeds. All the values are round-off at 10^{-3}.

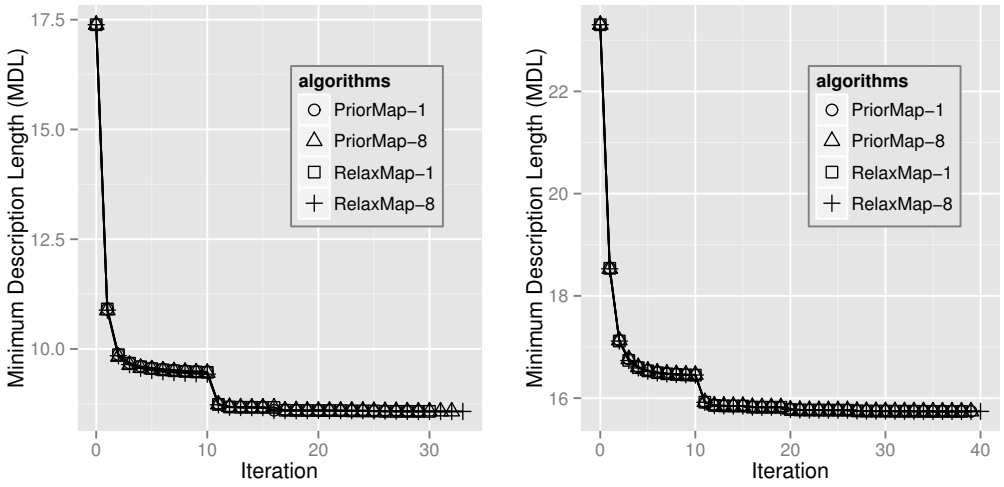
Dataset	Algorithm	Average	Minimum	Maximum
web-BerkStan	Infomap	9.298	9.297	9.300
	PriorMap-8	9.297	9.295	9.298
	RelaxMap-8	9.297	9.294	9.301
web-NotreDame	Infomap	12.397	12.395	12.402
	PriorMap-8	12.397	12.395	12.398
	RelaxMap-8	12.397	12.395	12.399
web-Stanford	Infomap	8.577	8.576	8.579
	PriorMap-8	8.577	8.575	8.579
	RelaxMap-8	8.576	8.574	8.577
wiki-Talk	Infomap	20.869	20.818	20.923
	PriorMap-8	20.802	20.784	20.819
	RelaxMap-8	20.841	20.790	20.889
soc-Pokec	Infomap	16.894	16.892	16.897
	PriorMap-8	16.899	16.898	16.900
	RelaxMap-8	16.899	16.898	16.901
soc-LiveJournal1	Infomap	15.742	15.740	15.744
	PriorMap-8	15.741	15.740	15.742
	RelaxMap-8	15.741	15.741	15.742

Test machines. We use three different multicore computers to perform our parallel, shared-memory experiments with RelaxMap. One 8-core machine, *Machine-I*, has two Intel Xeon E5420 quad-core processors (2.50 GHz, 12 MB L2 Cache) and 16 GB of main memory. The 12-core machine, *Machine-II*, has two six-core Intel Xeon E5-2430L processors (2.00 GHz, 15 MB Intel Smart Cache) and 64 GB of main memory. The 60-core machine, *Machine-III*, has four 15-core Intel Xeon E7-4890v2 processors (2.80 GHz, 37.5 MB Intel Smart Cache) and 1 TB of main memory.
Experimental Parameters. Except where noted otherwise, all results are based on the average of 10 experimental runs with different random seeds. The threshold value τ for the stop condition of iteration is $1e-3$, and the maximum iteration number for each movement-unit (i.e. single-node, aggregated node, or sub-module) is 10.

5.2. Clustering Quality Analysis

In this section, we demonstrate that RelaxMap and PriorMap find clusterings of similar quality to the original Infomap [Rosvall et al. 2009] algorithm.

We begin with a standard network benchmark, in which benchmark graphs with known communities are constructed randomly according to a mixing parameter that describes how likely inter-community edges are among all edges in the benchmark graph [Lancichinetti and Fortunato 2009]. Given this planted partition, the standard score for a clustering is its normalized mutual information (NMI) [Danon et al. 2005], which equals 1 if it produces the exact same communities as the planted partition, and 0 if the all sets are pairwise disjoint. Infomap was determined to be the best-performing algorithm in an objective third-party benchmark study [Lancichinetti and Fortunato 2009]; we want to evaluate whether the relaxed consistency model and reduced search space can achieve similar results.



(a) web-Stanford (b) soc-LiveJournal1

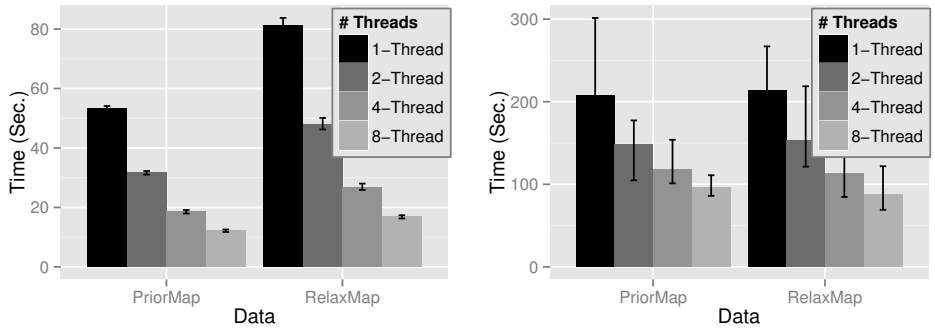
Fig. 7: The MDL values with respect to each iteration for the **PriorMap** method and the regular full-search algorithm (**RelaxMap**) with (a) *web-Stanford* and (b) *soc-LiveJournal1* datasets. These plots clearly show that the convergence-rate of PriorMap is similar to that of RelaxMap in each iteration with both sequential runs and consistency-relaxed parallel runs.

Figure 6-(a) shows the average of the NMI of clusterings produced by Infomap, a serial version of RelaxMap that runs with only one thread (**RelaxMap-1**), and RelaxMap running with 8 threads (**RelaxMap-8**), over 100 runs for graphs generated with varying mixing parameter. The experimental result for PriorMap, which compares Infomap, PriorMap-1, and PriorMap-8, is shown in Figure 6-(b). The parallel and serial algorithms achieve the same quality: the proposed algorithms RelaxMap and PriorMap find clusters with the same NMI as the clusters found by Infomap itself. All are able to identify the planted partitions with mixing parameter below 0.8. Results are shown for graphs with 5000 nodes, but other sizes showed similar results. We conclude that clusterings identified by RelaxMap and PriorMap in parallel are no worse than those found by the sequential Infomap algorithm.

We also evaluated real-world datasets as well as the benchmark datasets. Since there exists no planted partition for the real-world datasets, we cannot assess the accuracy of the result with an objective metric as we did with NMI for the benchmark networks. However, since the objective of the algorithm is to minimize the map equation, which has been shown to give good results on benchmark networks, we compare the shortest description lengths obtained on the real networks.

Table III compares final output qualities of the 8-way parallel results of RelaxMap (**RelaxMap-8**) and PriorMap (**PriorMap-8**) algorithms and the Infomap algorithm in terms of the final MDL code length for six real-world datasets in Table II. In Table III, we show the average, minimum, and maximum of MDL code length from 10 runs of the experiment. As illustrated in Table III, the quality achieved by RelaxMap and PriorMap are within the error bounds of the Infomap algorithm for most datasets.

In Table III, the results for *soc-Pokec* dataset are worse than those of other datasets. Although the absolute quality difference on average is small (about 0.005), it shows that the RelaxMap algorithm results does not capture the same MDL as the serial algorithm. As we mentioned in our previous work [Bae et al. 2013], both single-threaded tests of RelaxMap and 8-way parallel RelaxMap results from the RelaxMap algorithm are very similar to each other, so the consistency relaxation feature of



(a) web-BerkStan (b) wiki-Talk

Fig. 8: The elapsed time of the **RelaxMap** and **PriorMap** parallel algorithms on *Machine-I* with (a) *web-BerkStan* and (b) *wiki-Talk* from Table II using 1 through 8 threads. With (a) *web-BerkStan*, the runtimes of both algorithms with 8-thread are about 5 times faster than sequential runtimes, and PriorMap is much faster than RelaxMap with the same number of threads. Other real-world datasets, e.g. *web-Stanford*, *web-NotreDame*, *soc-Pokec*, and *soc-LiveJournal1*, also show similar performance patterns as web-BerkStan in (a). On the other hand, with (b) wiki-Talk dataset, PriorMap does not achieve any benefit from the prioritization compared to RelaxMap.

our parallel algorithms is not the cause of the quality difference. Because the number of sub-modules differ between the two methods, as does the convergence behavior once sub-modules are computed, subtle implementation differences in how sub-modules are generated explain the difference in quality.

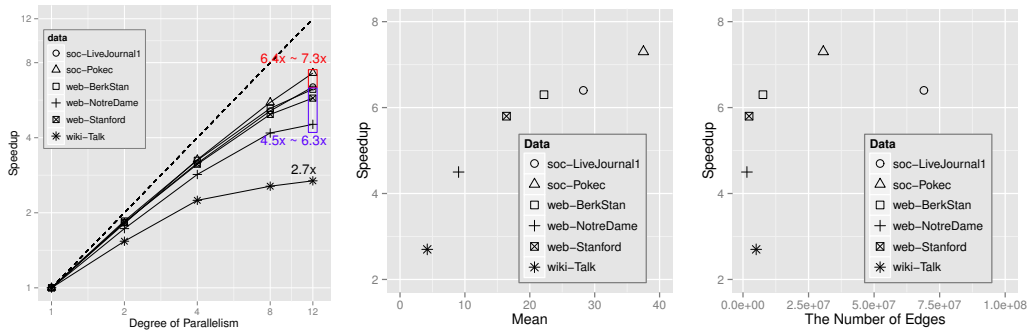
The convergence patterns of the sequential and 8-way parallel runs for both the PriorMap and the RelaxMap algorithms are shown in Figure 7 to check whether the parallelism affects the convergence patterns of both algorithm. Although we show the convergence pattern of RelaxMap and PriorMap algorithms with only two datasets in Figure 7, the same patterns happen to the experiments with all the real-world test datasets in Table II. From this experimental results, we can infer two things: (1) the prioritization method can achieve the same quality improvement as the non-prioritized method in each iteration, and (2) the parallel PriorMap, which combines the relaxed consistency of RelaxMap with the priortization method, also exhibits the same quality improvement pattern as the sequential non-prioritized method.

5.3. Parallel Performance Analysis

In Section 5.2, we discussed the output quality of the proposed parallel algorithms compared to the sequential Infomap algorithm, and showed that the proposed parallel RelaxMap and PriorMap achieve similar quality to Infomap. In this section, we would like to analyze parallel performance of the RelaxMap and PriorMap algorithms.

In our previous work [Bae et al. 2013], we compared RelaxMap-1 to Infomap with respect to the runtime, and it showed that the runtime of RelaxMap-1 is comparable to the runtime of Infomap, the state-of-the-art serial algorithm. Furthermore, PriorMap-1 is always faster than Infomap/RelaxMap-1 (up to 1.5 times faster) except with the wiki-Talk dataset because of the efficient "prioritization" approach. Instead of comparing to the runtime of Infomap, in this paper we therefore measure the parallel performance, such as the *speedup* and *efficiency*, of the proposed algorithms based on the sequential runtime of the corresponding algorithm of each dataset.

Figure 8 refers to the average, minimum, and maximum elapsed times of 1-way through 8-way parallel **RelaxMap** and **PriorMap** algorithms with two real-world test datasets, *web-BerkStan* and *wiki-Talk*, on *Machine-I*. These times correspond to the experiments of Table III. We can



(a) Speedup (b) Speedup vs. Avg. Degree (c) Speedup vs. Number of Edges

Fig. 9: (a) Parallel speedup of **RelaxMap** on *Machine-II* with six real-world datasets in Table II. **RelaxMap** achieves about 6.4 to 7.3 times speedup with social network datasets (in the red rectangle), 4.5 to 6.3 times speedup with web-graph datasets (in the blue rectangle), and 2.7 times speedup with Wiki-Talk dataset in 12-way parallelism. (b) Relation between Speedup of RelaxMap with 12-way parallelism and the average of the degree of vertices. (c) Relation between Speedup of Relaxmap with 12-way parallelism and the number of edges of datasets.

clearly see that both RelaxMap and PriorMap show performance gains as the degree of parallelism increases with those datasets. Although we only show runtimes for web-BerkStan and wiki-Talk in Figure 8, experiments using 8 threads (RelaxMap-8 and PriorMap-8) complete 4 to 5 times faster than experiments using 1 thread (RelaxMap-1 and PriorMap-1) for all web-graphs and social network graphs in Table III as similar as Figure 8-(a). Also, Figure 8 shows that the elapsed times of the prioritized method are about 1.2 to 1.5 times faster than the corresponding times of the non-prioritized method, except for the wiki-Talk dataset as we discuss in Section 5.4, which implies that the proposed prioritization benefits actual runtime without loss of output quality.

In addition to the parallel runtime analysis, we would like to show the parallel *speedup* of **RelaxMap**. Parallel speedup is a measurement of how much the parallel execution is faster than the corresponding sequential execution, which can be calculated as the sequential runtime, T_{seq}, divided by the parallel runtime, $T(p)$, with p-way parallel execution: $s = T_{seq}/T(p)$. Figure 9-(a) shows the parallel speedup of **RelaxMap** on *Machine-II* with real-world datasets in Table II. The dotted line means the perfect linear speedup. RelaxMap achieves fastest (6.4× to 7.3×) speedup with social network datasets (in the red rectangle in Figure 9-(a)), 4.5× to 6.3× speedup with web-graph datasets (in the blue rectangle in Figure 9-(a)), and least (2.7×) speedup with Wiki-Talk dataset.

Based on the parallel speedup results in Figure 9-(a), which shows a weak correlation between the types of graphs and speedup, we conclude that other properties of the networks have affected speedup more than the size of the networks. We assume that graphs of the same type have some similar properties. From Table II, social network graphs have higher average degree than web-graphs and wiki-Talk dataset, and we found that the density (or the average degree) of the graphs are more related to the parallel speedup of the proposed algorithm than the number of edges of the graphs, as shown in Figure 9-(b) and -(c). The denser (higher average of degree) graphs tend to achieve better performance than the sparser (lower average of degree) graphs. Note that *wiki-Talk* is much larger than *web-Stanford* with respect to the number of edges (or size of the graphs), but the algorithm achieves better speedup with *web-Stanford* than with *wiki-Talk*.

In order to analyze the relation between the parallel speedup of the algorithm and the average vertex degree of graphs, take two different graphs, say $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, where $|V_1|$ and $|V_2|$ are the number of vertices which $|V_1| = |V_2|$, $|E_1|$ and $|E_2|$ are the number of edges which $c * |E_1| = |E_2|$, and $c > 1.0$. Since the parallel workloads of RelaxMap are assigned to threads with respect to the

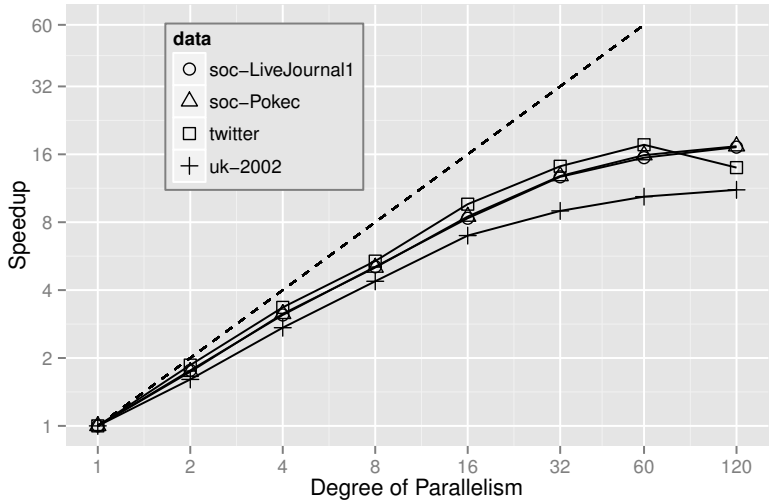


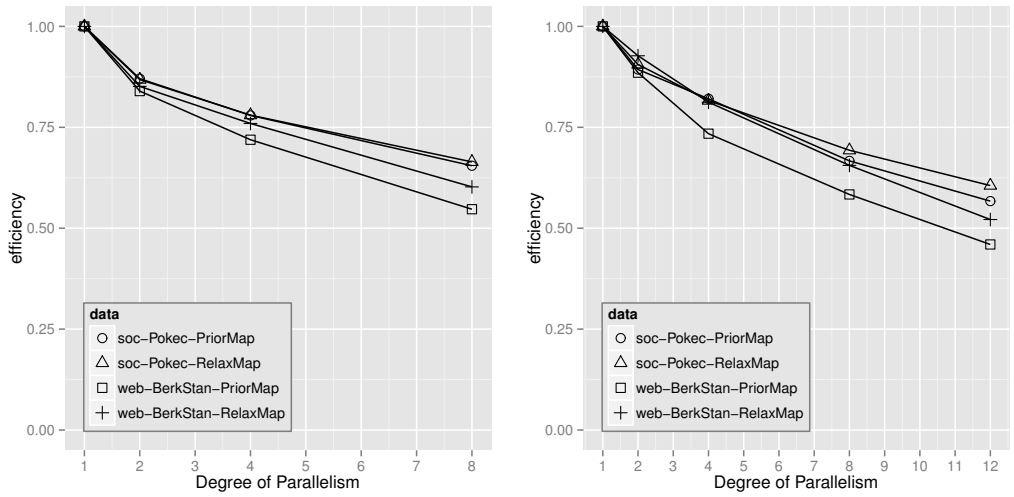
Fig. 10: Parallel speedup of **RelaxMap** on *Machine-III* with large real-world datasets in Table II. **RelaxMap** achieves up to 17.6 times speedup within 60-core machine without loss of output quality.

number of vertices in Algorithm 3, the non-parallel portion of RelaxMap is approximately $O(|V|)$ and the parallel portion is asymptotically $O(|E|)$ in complexity. Thus, the overall runtime of sequential run of the algorithm is roughly $T_s = O(|V|) + O(|E|)$ and that of parallel run with p threads is $T_p = O(|V|) + O(|E|)/p$. Based on these simplified assumptions, we compare the parallel speedup of G_1 and G_2, named s_1 and s_2 correspondingly. $s_1 = T_s^1/T_p^1$ and $s_2 = T_s^2/T_p^2$, where $p > 1$, so we compare s_1 and s_2 by calculating $s_1 - s_2$ as follows (where $O(|V_1|) = O(|V_2|) = \alpha$, $O(|E_1|) = \beta$, and $O(|E_2|) \approx c\beta$):

$$\begin{aligned} s_1 - s_2 &= \frac{O(|V_1|) + O(|E_1|)}{O(|V_1|) + O(|E_1|)/p} - \frac{O(|V_2|) + O(|E_2|)}{O(|V_2|) + O(|E_2|)/p} \\ &= \frac{\alpha + \beta}{\alpha + \beta/p} - \frac{\alpha + c\beta}{\alpha + c\beta/p} \\ &= \frac{p(1-p)(c-1)\alpha\beta}{(p\alpha + \beta)(p\alpha + c\beta)} < 0. \end{aligned}$$

This asymptotic analysis confirms that the proposed algorithms will perform higher speedup with the graphs with higher average degree of vertices than with the graphs with lower average degree. Note that other properties, such as the skewness of edge distribution, also affect the speedup as well as the average degree. For instance, if two graphs have the same number of vertices and edges, then one with lower skewness would have higher speedup, since the algorithms achieve better load balance with it than the other graph.

We also performed parallel performance experiments on Machine-III, which has 4 CPUs of 15 cores with hyper-threading technology and 1 TB of main memory. Figure 10 shows the parallel speedup of **RelaxMap** on *Machine-III* with large real world datasets, including *twitter* dataset which has billions of edges. Dotted line in Figure 10 represents the perfect speedup. Due to the long running time, we run 3 times with twitter dataset for each degree of parallelism and each run was limited to 4 "Super-Steps."



(a) Machine-I (b) Machine-II

Fig. 11: Parallel efficiency of the **PriorMap** and **RelaxMap** algorithms on *Machine-I* and *Machine-II* with *soc-Pokec* and *web-BerkStan* datasets in Table II. As seen in this figure, each PriorMap efficiency result is comparable to the corresponding RelaxMap efficiency result.

As shown in Figure 10, RelaxMap achieves speedup gradually as the number of threads increases. Since the parallel overhead increases as the degree of parallelism increases, the gap between the perfect speedup and the actual speedup is also increased as the degree of parallelism increases. In Figure 10, the algorithm achieved up to 17.6 times speedup when the degree of parallelism is 60, without loss of output quality. Although the speedups on *soc-Pokec* and *soc-LiveJournal1* are almost identical each other, that is coincidental; the runtimes are very different on each other and the speedups on them are different in Figure 9-(a). As we discussed above, the parallel speedup of RelaxMap is more related to the average degree and the skewness than the number of edges, so that *uk-2002* dataset shows the least speedup though it is the second largest dataset among tested datasets in Figure 10. Since the CPUs in Machine-III support hyper-threading technology, we also tested RelaxMap with 120 threads on Machine-III. The experimental result shows that the hyper-threading technology benefits to the parallel performance of the algorithm in general, but not in every case as shown in twitter case in Figure 10.

Here, we investigate why the RelaxMap algorithm does not get benefits of hyper-threading with the twitter dataset. Since we doubled the number of threads from 60 threads to 120 threads, the lock-contention chance would also double asymptotically. To make matters worse, hyper-threading does not increase the actual number of physical cores, but two threads share a physical core. Thus, roughly each thread uses only a half of each physical core; the module information updating time would increase twice or more since the module information updating part is inside of the locking mechanism in Algorithm 3. In our experiments, when the algorithm used hyper-threading on *Machine-III* explicitly by launching 120 threads, it increased the lock-contention for updating the module information, which is represented in line 3-5 of Algorithm 3, especially at the first iteration, when almost all vertices are getting into the instant updating part. This lock-contention with hyper-threading would be worse if the data is larger and/or more skewed, and the twitter dataset is the largest and the most skewed among the tested datasets in Figure 10. We found that the runtimes inside of the locking mechanism in Algorithm 3 with 120 threads increased compared to the corresponding runtimes with 60 threads with all datasets in Figure 10, and the time increased more than twice with twitter so

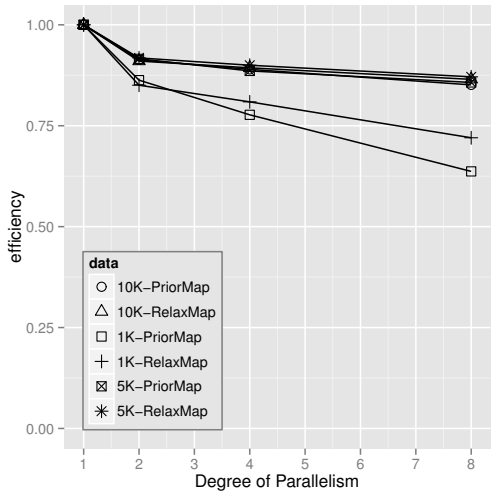


Fig. 12: The parallel efficiency of the **PriorMap** and **RelaxMap** algorithms on *Machine-I* with benchmark datasets in Table II. Even though the graph size is relatively small ($|V| \leq 10,000$), both algorithms show very high efficiency. Both algorithms achieve 85% efficiency in 8-way parallelism with 5k and 10k benchmark datasets.

that hyper-threading made poor performance with the twitter dataset in Figure 10. While benefits of hyper-threading in other parts of the proposed algorithm could catch up the performance loss with other datasets so that RelaxMap achieved a slight improvement of the overall speedup, the benefits could not cover the performance loss for the twitter case.

The Parallel efficiency ε of an algorithm compares the parallel runtime to the best possible runtime assuming perfect scalability. Thus, we calculate the parallel efficiency of the proposed algorithms to analyze the scalability of them. Eq. (3) is the equation of the parallel efficiency:

$$\varepsilon = \frac{T_{seq}}{pT(p)} \tag{3}$$

where p is the number of parallel units, $T(p)$ is the time with p parallel units, and T_{seq} is the time of the sequential version.

Figure 11 illustrates the parallel efficiency of the RelaxMap and PriorMap algorithms with real-world datasets in Table II on *Machine-I* (a) and *Machine-II* (b), respectively. In Figure 11, both algorithms show 50% to 70% parallel efficiency for 8-way parallelism on both Machine-I and Machine-II with soc-Pokec and web-BerkStan datasets. This parallel efficiency offers a significant improvement in runtime over the state-of-the-art. In addition, the PriorMap algorithm is about 1.2 to 1.5 times more efficient in sequential experiments than the RelaxMap algorithm, and it still achieves similar parallel efficiency to the RelaxMap algorithm as shown in Figure 11.

The parallel efficiency of the proposed PriorMap and RelaxMap algorithms on *Machine-I* with benchmark datasets in Table II is shown in Figure 12. Since the benchmark datasets are much smaller datasets compared to real-world datasets as in Table II, we expected that the parallel efficiency of the proposed algorithms using small benchmark datasets to be lower than with real-world datasets. However, the parallel efficiency of both RelaxMap and PriorMap with 5k and 10k benchmark datasets is about 85% in 8-way parallelism in Figure 12, which is higher than the real-world datasets. Also, we found that the PriorMap is about 1.2 to 1.3 times faster than the RelaxMap in our experimental results of 100 runs, even with small benchmark datasets.

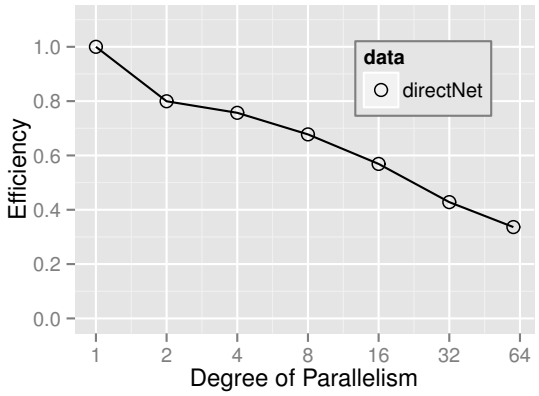


Fig. 13: The weak-scale parallel efficiency of **RelaxMap** on *Machine-III* with large benchmark datasets, from 50k to 3M vertices in Table II. As the degree of parallelism increases, we vary the dataset size proportionally. RelaxMap shows about 35% weak-scale efficiency with 60-way parallelism.

The maximum degrees of the benchmark datasets are much smaller than those of real-world datasets. As a consequence, the edge degree distributions of the benchmark datasets show low variances and those of the real-world datasets are high variances. Since the work to find a new community for each vertex (a.k.a. bestNewModule() function in Algorithm 3 and Algorithm 4) is proportional to the degree of the vertex, the work for each vertex in the benchmark datasets is more uniform than than in real-world datasets, which may result in better load-balancing when scheduling work among processors and correspondingly better parallel efficiency.

To quantify scalability, we compute the weak-scale parallel efficiency. Weak-scale parallel efficiency is measured by comparing the average runtime of p-way parallel execution of $p \times w$ workloads, t_{pw}, with the average runtime of sequential execution of w workloads, t_w. If $t_{pw} = t_w$, then the algorithm achieve 1.0 weak-scale parallel efficiency. The definition of weak-scale parallel efficiency ε_w is in Eq. (4).

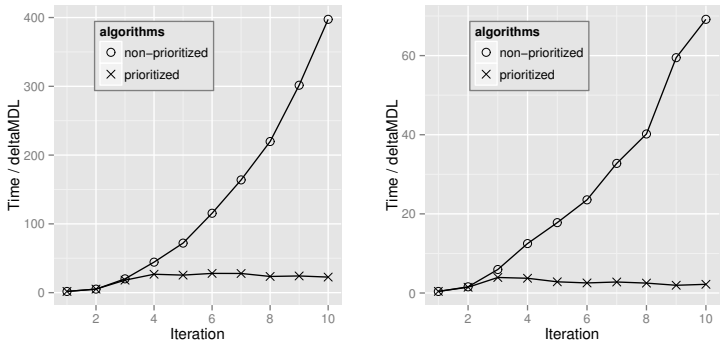
$$\varepsilon_w = \frac{t_w}{t_{pw}}. \tag{4}$$

In Figure 13, we show weak-scale parallel efficiency with the benchmark graphs of 50k, 100k, 200k, 400k, and up to 3M vertices in Table II on *Machine-III*. Note that, although the workload is roughly proportional to the number of edges, it is sensitive to the non-deterministic features of the algorithm. Also, it is not expected to achieve perfect scaling since there is a non-trivial sequential portion of the parallel algorithm, and the algorithm runtime depends on the degree distribution of the input graph. RelaxMap achieved about 35% weak-scale efficiency on a 60-times larger graph with 60-way parallelism in Figure 13, which is good weak-scale performance for a complex parallel algorithm like RelaxMap, as we tested on big benchmark datasets, from 50k to 3M vertices in Table II with sequential, 2-way, and up to 60-way parallelism, correspondingly.

5.4. Prioritization Performance Analysis

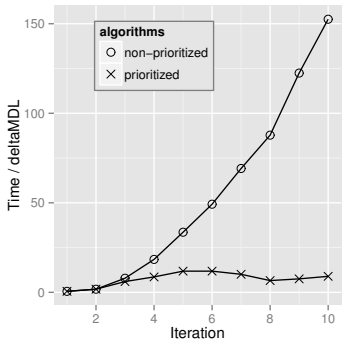
We have motivated our choice of prioritization strategy in Section 4.1; we now measure the performance improvement realized by applying it.

Figure 14 shows the effectiveness of the prioritization method by comparing the time cost per unit MDL improvement, $t/\Delta MDL$, for the first 10 iterations. Since each graph shows the time for unit MDL improvement, the lower value means that an algorithm uses time more effectively to improve

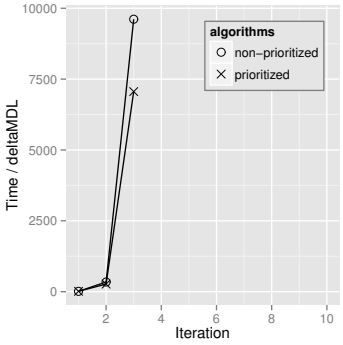


(a) web-BerkStan

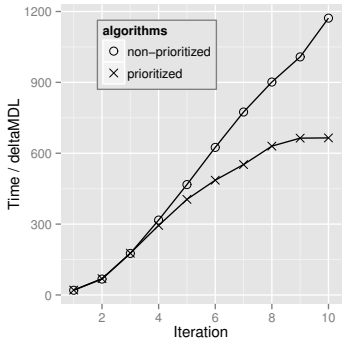
(b) web-NotreDame



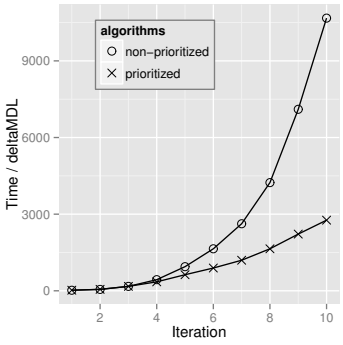
(c) web-Stanford



(d) wiki-Talk



(e) soc-Pokec



(f) soc-LiveJournal1

Fig. 14: The effectiveness of the prioritization method and non-prioritized method with six real-world datasets in Table II. In each plot, the x-axis represents the iteration number in the first super-step and the y-axis is the time per unit MDL improvement ($t/\Delta(MDL)$). Although the runtime would depend on the complexity of each dataset, the prioritization method is much more effective than the non-prioritized method on all of the tested datasets, except wiki-Talk dataset.

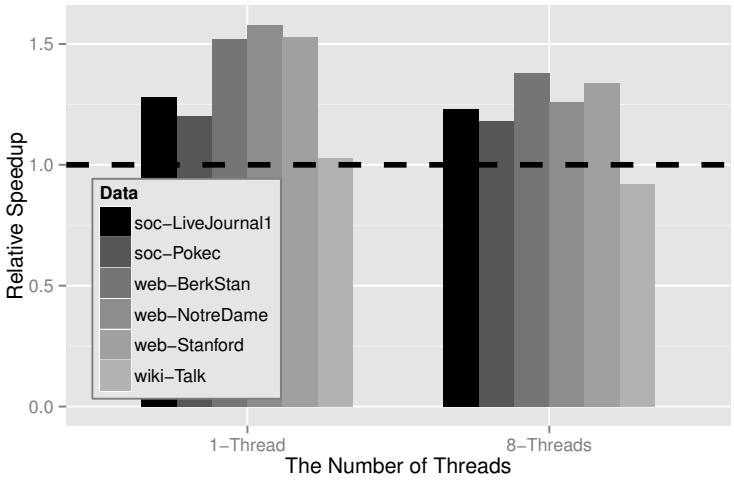


Fig. 15: The relative speedup of prioritized method vs. non-prioritized method with 1-thread runs and 8-thread parallel runs in terms of real-world datasets in Table II on *Machine-I*. All of the cases show that prioritized method is about 1.2 to 1.5 times faster than the non-prioritized method, except with wiki-Talk dataset. Since both algorithms converge in a few iterations for each super-step with wiki-Talk dataset, the prioritized method will not benefit from pruning the search space.

the output quality. For all datasets, the proposed prioritized method (**prioritized**) runs much more efficiently than the original full-search method (**non-prioritized**), except in the case of the *wiki-Talk* dataset in Figure 14-(d). Specifically, the prioritized method maintains a relatively low time cost per unit of MDL improvement throughout the computation in contrast to the non-prioritized method.

For the first few iterations, both the prioritized method and the non-prioritized method provide similar effectiveness in Figure 14. This is natural, since a lot of vertices are moved during this period, so most of vertices are neighbors of a moved vertex, and therfore activated by the algorithm. However, in later iterations, most of the vertices "settle down" in their final communities. Thus, in later iterations, the prioritized method will activate far fewer vertices than the non-prioritized algorithm which results in using much less time but still covers similar quality improvements.

For *wiki-Talk* dataset, the prioritized method shows similar effectiveness per iteration in contrast to the other datasets, in Figure 14-(d). The reason is that the algorithms converged in 3 iterations with *wiki-Talk* dataset for the first super-step, so the prioritization does not significantly benefit the elapsed time with the dataset.

In Section 5.2, we have shown the overall convergence rate of the priortized and the non-prioritized methods in Figure 7. In fact, our experiments show that the convergence rate of the prioritized method is the same as the non-prioritized full-search method but performs significantly less work.

Figure 15 illustrates the relative speedup of the prioritized method across several datasets. The horizontal dashed line represents a speedup of 1.0. The prioritized method is about 1.2 to 1.5 times faster than non-prioritized method with one thread. With 8-way parallel experiments, the relative speedup slightly reduced from 1.5 to around 1.35 for small datasets but remains the same as the sequential runs for the experiments with larger datasets (soc-LiveJournal1 and soc-Pokec).

In Figure 15, the prioritized method does not show better efficiency compared to non-prioritized method for *wiki-Talk* dataset, in contrast to the other experiments. About 94% of the vertices in the wiki-Talk dataset are "dangling:" they have no outgoing edges. With lots of dangling vertices, the algorithm converges in just a few iterations for each super-step; there are not enough opportunities for the algorithm to significantly prune the search space. With 8-way parallelism, the prioritized method

in fact slows down the runtime; we found that the prioritized method requires more super-steps than the non-prioritized method, for wiki-Talk dataset. However, the prioritized method appears to produce more stable outputs than non-prioritized runs, as shown in Table III.

6. RELATED WORK

Several parallel community detection algorithms have been proposed. A well-known metric for community detection problem is modularity [Newman and Girvan 2004], and the modularity maximization method [Clauset et al. 2004] is one of the most-used algorithms for determining community structure. Riedy *et al.* [Riedy et al. 2011, 2012] worked on parallel modularity maximization algorithm under shared-memory many-core environments. Although this work achieves benefits from the parallelism and handles graphs with billions of edges, it still has the "resolution limit" problem since it is a modularity optimization method, and multiple third party benchmarks have shown that Infomap delivers better quality results. Also, Fagginger Auer and Bisseling [Fagginger Auer and Bisseling 2013] proposed a graph coarsening method for a modularity based graph clustering and fine-grained shared memory parallel algorithm for multi-core CPUs and GPUs systems. Although the algorithm scales well for large graphs, a parallel local refinement method is missing from the algorithm due to the difficulty of parallelization.

The Louvain method [Blondel et al. 2008] is well-known modularity maximization community detection algorithm, which utilizes aggregation step efficiently. Recently, Bhowmick and Srinivasan proposed a shared-memory parallel template for the Louvain method [Bhowmick and Srinivasan 2013]. The parallel Louvain method is similar methodology to the proposed RelaxMap algorithm, which searches new communities in parallel but updates the search results under critical section to guarantee the correctness of the algorithm. The parallel Louvain method achieves similar modularity score as in sequential with the benefit of parallel execution. However, the parallel Louvain method is a modularity-based algorithm, which is a structure-based community detection algorithm, but the RelaxMap algorithm optimizes the map equation, which is a network flow-based community detection method. Also, Staudt and Meyerhenke [Staudt and Meyerhenke 2015] proposed several shared-memory parallel community detection algorithms for undirected graphs: Parallel Label Propagation (PLP), Parallel Louvain Method (PLM), Parallel Louvain Method with Refinement (PLMR), and Ensemble Preprocessing (EPP). PLP is based on the label propagation method, and the other algorithms optimize modularity for finding community structure. The implementations of these algorithms are published as a component of *NetworKit* [Staudt et al. 2014].

Zhang *et al.* [Zhang et al. 2009] proposed another metric for community detection called *propinquity* and provided an assoicated parallel algorithm based on this metric. By utilizing an incremental design for the propinquity calculation, which avoids unnecessary recalculation of the propinquity values per each iteration, they achieved better efficiency in parallel than without applying incremental design.

Since the above algorithms use a different objective function for evaluating communities, we did not perform any experimental evaluation between the proposed algorithms and the algorithms in this paper.

Niu *et al.* suggested an interesting *lock-free* scheme for parallel stochastic gradient descent (SGD) algorithms, called *HogWild!* [Niu et al. 2011]. In their paper, Niu *et al.* proved that the HogWild! approach will converge in optimal ratio, which is similar to its original algorithm, given a sparse dataset, although the HogWild! approach allows overwrites on decision variables for the SGD optimization due to not using lock mechanism in shared memory parallelism. Since most of the real network graphs are usually sparse, in this paper, we proposed a parallel flow-based community detection algorithm in terms of the *map equation* [Rosvall et al. 2009] metric, by applying lock-free scheme as similar as HogWild! for SGD algorithms.

Zhang *et al.* [Zhang et al. 2011] proposed a distributed framework for the fast convergence of iterative computation, called *PrIter*. In contrast to working all data in each iteration, PrIter supports prioritized iterations which enable it to achieve much faster convergence. They experimented PrIter with several iterative graph algorithms, such as single source shortest path, pagerank, and connected

components, and the PrIter shows significant performance gains on those algorithms by the prioritized iteration scheme.

A fast modularity-based community detection algorithm was presented by Shiokawa *et al.* [Shiokawa et al. 2013]. For avoiding high computation cost on the modularity-based community detection algorithm, they used incremental aggregation, incremental pruning, and reordering of vertex selections. Based on those key ideas, Shiokawa *et al.* [Shiokawa et al. 2013] can partition of large graphs, such as a graph with a few billion edges. This results implies that searching all vertices in each iteration is inefficient in modularity-based community detection.

In addition, Liu and Murata [Liu and Murata 2010] proposed an algorithm that avoids the local optima problem of a modularity-specialized label propagation algorithm. The authors briefly mentioned that the algorithm could be sped up by only updating the labels of vertices whose neighbors were updated before, though they did not provide any detailed performance evaluation on the prioritized method.

Most of community detection algorithms search for a new community for each vertex based on the vertex's neighbors' status. Thus, in many cases, a community detection algorithm may not need to search for a new module for a vertex, if none of its neighbors has been updated since it was assigned to the current community. Based on this simple idea, we proposed a prioritized method for the flow-based community detection with respect to the map equation in this paper to improve the algorithm's efficiency.

7. CONCLUSION AND FUTURE WORK

We proposed a new parallel flow-based community detection algorithm called *RelaxMap*. Due to the original algorithm's sequential and dependent nature, it is difficult to directly implement an efficient parallel algorithm with identical behavior. The proposed algorithm relaxes the consistency model to allow several vertex moves to be considerd in parallel before synchronizing. We assume, and experimentally verify, that conflicts do not arise frequently due to graphs typically being sparse. Empirically, we show the convergence rate of the RelaxMap algorithm is as fast as the original sequential algorithm.

In addition to the fast convergence rate, our proposed parallel algorithm achieves similar quality to the sequential algorithm, and achieves acceptable parallel efficiency in multiple experimental environments. With 8-way parallelism, we achieved about 50% - 70% parallel efficiency with real-world datasets and about 85% parallel efficiency with benchmark datasets. When we tested the algorithm with 60-way parallelism, the proposed algorithm achieved 17.6 times speedup without loss of output quality.

We also introduced an efficient flow-based community detection algorithm, which utilizes *prioritization* method, and applied the prioritization to the RelaxMap parallel algorithm. The proposed algorithm achieves better efficiency by reducing unnecessary investigation. The priortization method is up to 1.5 times faster than the non-prioritization method without loss of the quality of the outputs, and still achieves highly efficient parallelism. Since the relaxation and prioritization methods are general ideas, we believe the proposed methods are applicable to other community detection algorithms for improving their efficiency and scalability.

For future work, we would like to extend the proposed prioritized RelaxMap parallel algorithm to parallelizing hierarichical community detection algorithms and community detection algorithms for dynamic networks.

ACKNOWLEDGMENTS

We would like to thank Andrea Lancichinetti for the synthetic graph generator. The authors would also like to thank the anonymous reviewers for their insightful comments.

REFERENCES

Rodrigo Aldecoa and Ignacio Marín. 2013. Exploring the limits of community detection strategies in

complex networks. *Scientific reports* 3, 2216 (2013).
Seung-Hee Bae, Daniel Halperin, Jevin West, Martin Rosvall, and Bill Howe. 2013. Scalable Flow-Based Community Detection for Large-Scale Network Analysis. In *Proceedings of 2013 IEEE 13th International Conference on Data Mining Workshops (ICDMW)*. IEEE, 303–310.
Sanjukta Bhowmick and Sriram Srinivasan. 2013. A Template for Parallelizing the Louvain Method for Modularity Maximization. In *Dynamics On and Of Complex Networks, Volume 2*, Animesh Mukherjee, Monojit Choudhury, Fernando Peruani, Niloy Ganguly, and Bivas Mitra (Eds.). Springer New York, 111–124. DOI:http://dx.doi.org/10.1007/978-1-4614-6729-8_6
Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008.
Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.
Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer networks and ISDN systems* 30, 1 (1998), 107–117.
Aaron Clauset, Mark EJ Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical review E* 70, 6 (2004), 066111.
Anne Condon and Richard M Karp. 2001. Algorithms for graph partitioning on the planted partition model. *Random Structures and Algorithms* 18, 2 (2001), 116–140.
Leon Danon, Albert Diaz-Guilera, Jordi Duch, and Alex Arenas. 2005. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment* 2005, 09 (2005), P09008.
Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. 2010. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 810–818.
Bas Fagginger Auer and Rob H Bisseling. 2013. Graph coarsening and clustering on the GPU. In *Graph Partitioning and Graph Clustering:10th DIMACS Implementation Challenge Workshop*. Vol. 588. American Mathematical Soc.
Santo Fortunato and Marc Barthélemy. 2007. Resolution limit in community detection. *Proceedings of the National Academy of Sciences* 104, 1 (02 Jan. 2007), 36–41. DOI:http://dx.doi.org/10.1073/pnas.0605965104
Anne-Claude Gavin, Patrick Aloy, Paola Grandi, Roland Krause, Markus Boesche, Martina Marzioch, Christina Rau, Lars Juhl Jensen, Sonja Bastuck, Birgit Dümpelfeld, and others. 2006. Proteome survey reveals modularity of the yeast cell machinery. *Nature* 440, 7084 (2006), 631–636.
Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826.
David F. Gleich and Leonid Zhukov. 2005. Scalable Computing with Power-Law Graphs: Experience with Parallel PageRank. In *SuperComputing 2005*. Poster.
Roger Guimera and Luis A Nunes Amaral. 2005. Functional cartography of complex metabolic networks. *Nature* 433, 7028 (2005), 895–900.
Roger Guimerà, Marta Sales-Pardo, and L. A. N. Amaral. 2004. Modularity from fluctuations in random graphs and complex networks. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)* 70, 2 (2004). DOI:http://dx.doi.org/10.1103/physreve.70.025101
Tatsuro Kawamoto and Martin Rosvall. 2015. Estimating the resolution limit of the map equation in community detection. *Phys. Rev. E* 91 (Jan 2015), 012809. Issue 1. DOI:http://dx.doi.org/10.1103/PhysRevE.91.012809
Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web*. ACM, New York, NY, USA, 591–600. DOI:http://dx.doi.org/10.1145/1772690.1772751
Andrea Lancichinetti and Santo Fortunato. 2009. Community detection algorithms: A comparative

analysis. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)* 80, 056117 (2009).
Andrea Lancichinetti, Santo Fortunato, and Filippo Radicchi. 2008. Benchmark graphs for testing community detection algorithms. *Physical Review E* 78, 4 (2008), 046110.
Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (Jun. 2014).
Jure Leskovec, Kevin J. Lang, and Michael Mahoney. 2010. Empirical comparison of algorithms for network community detection. In *Proceedings of the 19th international conference on World wide web (WWW '10)*. ACM, New York, NY, USA, 631–640. DOI:http://dx.doi.org/10.1145/1772690.1772755
Xin Liu and Tsuyoshi Murata. 2010. Advanced modularity-specialized label propagation algorithm for detecting communities in networks. *Physica A: Statistical Mechanics and its Applications* 389, 7 (2010), 1493–1500.
Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727. http://dl.acm.org/citation.cfm?id=2212351.2212354
Padmanabhan K Menon, Gregory D Sweriduk, and Karl D Bilimoria. 2004. New approach for modeling, analysis, and control of air traffic flow. *Journal of guidance, control, and dynamics* 27, 5 (2004), 737–744.
Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.
Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. 2011. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. http://books.nips.cc/papers/files/nips24/NIPS2011_0485.pdf
OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf. (May 2008).
E Jason Riedy, Henning Meyerhenke, David Ediger, and David A Bader. 2011. Parallel community detection for massive graphs. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics-Volume Part I*. Springer-Verlag, 286–296.
Jason Riedy, David A Bader, and Henning Meyerhenke. 2012. Scalable Multi-threaded Community Detection in Social Networks. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 1619–1628.
Martin Rosvall, Daniel Axelsson, and Carl T Bergstrom. 2009. The map equation. *The European Physical Journal Special Topics* 178, 1 (2009), 13–23.
Martin Rosvall and Carl T Bergstrom. 2011. Multilevel compression of random walks on networks reveals hierarchical organization in large integrated systems. *PloS one* 6, 4 (2011), e18209–e18209.
Martin Rosvall, Alcides V Esquivel, Andrea Lancichinetti, Jevin D West, and Renaud Lambiotte. 2014. Memory in network flows and its effects on spreading dynamics and community detection. *Nature communications* 5 (2014).
Claude Elwood Shannon. 1948a. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (July 1948), 379–423. Issue 3.
Claude Elwood Shannon. 1948b. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (October 1948), 623–656. Issue 4.
Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2013. Fast Algorithm for Modularity-based Graph Clustering. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. 1170–1176.
C. Staudt and H. Meyerhenke. 2015. Engineering Parallel Algorithms for Community Detection in Massive Networks. *Parallel and Distributed Systems, IEEE Transactions on* PP, 99 (2015), 1–1. DOI:http://dx.doi.org/10.1109/TPDS.2015.2390633
Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2014. Networkit: An interactive tool suite for high-performance network analysis. *arXiv preprint arXiv:1403.3005* (2014).
Jevin D West, Theodore C Bergstrom, and Carl T Bergstrom. 2010. The Eigenfactor Metrics^{TM}: A

network approach to assessing scholarly journals. *College & Research Libraries* 71, 3 (2010), 236–244.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2011. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 13.

Yuzhou Zhang, Jianyong Wang, Yi Wang, and Lizhu Zhou. 2009. Parallel community detection on large networks with propinquity dynamics. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 997–1006.

Received December 2014; revised July 2015; accepted June 2016