Engineering Parallel Algorithms for Community Detection in Massive Networks

Christian L. Staudt and Henning Meyerhenke
Faculty of Informatics, Karlsruhe Institute of Technology (KIT), Germany
{christian.staudt, meyerhenke}@kit.edu

***Abstract*—The amount of graph-structured data has recently experienced an enormous growth in many applications. To transform such data into useful information, fast analytics algorithms and software tools are necessary. One common graph analytics kernel is disjoint community detection (or graph clustering). Despite extensive research on heuristic solvers for this task, only few parallel codes exist, although parallelism will be necessary to scale to the data volume of real-world applications. We address the deficit in computing capability by a flexible and extensible community detection framework with shared-memory parallelism. Within this framework we design and implement efficient parallel community detection heuristics: A parallel label propagation scheme; the first large-scale parallelization of the well-known Louvain method, as well as an extension of the method adding refinement; and an ensemble scheme combining the above. In extensive experiments driven by the algorithm engineering paradigm, we identify the most successful parameters and combinations of these algorithms. We also compare our implementations with state-of-the-art competitors. The processing rate of our fastest algorithm often reaches 50M edges/second. We recommend the parallel Louvain method and our variant with refinement as both qualitatively strong and fast. Our methods are suitable for massive data sets with billions of edges.**[1]

Keywords: Disjoint community detection, graph clustering, parallel Louvain method, parallel algorithm engineering, network analysis

I. Introduction

The data volume produced by electronic devices is growing at an enormous rate. Important classes of such data can be modeled by *complex networks*, which are increasingly used to represent phenomena as varied as the WWW, social relations, and brain topology. The resulting graph data sets can easily reach billions of edges for many relevant applications. Analyzing data of this volume in near real-time challenges the state of the art in terms of hardware, software, and algorithms. A particular challenge is not only the amount of data, but also its structure. Complex networks have topological features which pose computational challenges different from traditional HPC applications: In a *scale-free* network, the presence of a few high-degree nodes (hubs) among many low degree nodes generates load balancing issues. In a *small-world* network, the entire graph can be visited in only a few hops from any source node, which negatively affects cache performance. To enable network analysis methods to scale, we need algorithmic methods that harness parallelism and apply specifically to complex networks.

In this work, we deal with the task of *community detection* (also known as *graph clustering*) in large networks, i. e. the discovery of dense subgraphs. Among manifold applications, community detection has been used to counteract search engine rank manipulation [32], to discover scientific communities in publication databases [34], to identify functional groups of proteins in cancer research [15], and to organize content on social media sites [12]. So far, extensive research on community detection in networks has given rise to a variety of definitions of what constitutes a good community and a variety of methods for finding such communities, many of which are described in surveys by Schaeffer [32] and Fortunato [10]. Among these definitions, the lowest common denominator is that a community is an internally dense node set with sparse connections to the rest of the graph. While it can be argued that communities can overlap, we restrict ourselves to finding disjoint communities, i.e. a partition of the node set which uniquely assigns a node to a community. The quality measure *modularity* [14] formalizes the notion of a good community detection solution by comparing its *coverage* (fraction of edges within communities) to an expected value based on a random edge distribution model which preserves the degree distribution. Modularity is not without flaws (like the *resolution limit* [11], which can be partially overcome by different techniques [2], [18], [23]) nor alternatives [37], but has emerged as a well-accepted measure of community quality. This makes modularity our measure of choice. While optimizing modularity is NP-hard [6], efficient heuristics have been introduced which explicitly increase modularity.

For graphs with millions to billions of edges, only (near) linear-time community detection algorithms are practical. Several fast methods have been developed in recent years. Yet, there is a lack of research in adapting these methods to take advantage of parallelism. A recent attempt at assessing the state of the art in community detection was the *10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering* [1]. *DIMACS* challenges are scientific competitions in which the participants solve problems from a specified test set, with the aim of high solution quality and high speed. Only two of the 15 submitted implementations for modularity optimization relied on parallelism and only very few could handle graphs with billions of edges in reasonable time.

Accordingly, our objective is the development and imple-

[1] A preliminary version of this paper appeared in *Proceedings of the 42nd International Conference on Parallel Processing (ICPP 2013)* [35].

mentation of parallel community detection heuristics which are able to handle massive graphs quickly while also producing a high-quality solution. In the following, the competitors of the *DIMACS* challenge will be used for a comparative experimental study. In the design of such heuristics, we necessarily trade off solution quality against running time. The *DIMACS* challenge also showed that there is no consensus on what running times are acceptable and how desirable an increase in the third decimal place of modularity is. We therefore need to clarify our design goals as follows: In the comparison with other proposed methods, we want to place our algorithms on the Pareto frontier so that they are not dominated, i.e. surpassed in speed *and* quality at the same time. Secondly, we target a usage scenario: Our algorithms should be suitable as part of interactive data analysis workflows, performed by a data analyst operating a multicore workstation. Networks with billions of edges should be processed in minutes rather than hours, and the solution quality should be competitive with the results of well-established sequential methods.

We implement three standalone parallel algorithms: Label propagation [28] is a simple procedure where nodes adopt the community assignment (label) which is most frequent among their neighbors until stable communities emerge. We implement a parallel version of the approach as the PLP algorithm. The Louvain method [5] is a multilevel technique in which nodes are repeatedly moved to the community of a neighbor if modularity can be improved. We are the first to present a parallel implementation of the method for large inputs, named PLM. We also extend the method by adding a refinement phase on every level, which yields the PLMR algorithm. In addition to these basic algorithms, we also implement a two-phase approach that combines them. It is inspired by *ensemble learning*, in which the output of several weak classifiers is combined to form a strong one. In our case, multiple *base algorithms* run in parallel as an ensemble. Their solutions are then combined to form the *core communities*, representing the consensus of all base algorithms. The graph is coarsened according to the core communities, and then assigned to a single *final algorithm*. Within this extensible framework, which we call the *ensemble preprocessing* method (EPP), we apply PLP as base algorithms and PLMR as the final algorithm.

With our shared-memory parallel implementation of community detection by label propagation (PLP), we provide an extremely fast basic algorithm that scales well with the number of processors (considering the heterogeneous structure of the input). The processing rate of PLP reaches 50M edges per second for large graphs, making it suitable for massive data sets. With PLM, we present the first parallel implementation of the Louvain community detection method for massive inputs, and demonstrate that it is both fast and qualitatively strong. We show that solution quality can be further improved by extending the method with a refinement phase on every level of the hierarchy, yielding the PLMR algorithm. The EPP ensemble algorithm can yield a good quality-speed tradeoff on some instances when an even lower time to solution is desired. In comparative experiments, our implementations perform well in comparison to other state-of-the-art algorithms (Sec. V-E and V-F): Three of our algorithms are on the Pareto frontier.

Our community detection software framework, written in C++, is flexible, extensible, and supports rapid iteration between design, implementation and testing required for algorithm engineering [25]. In this work, we focus on specific configurations of algorithms, but future combinations can be quickly evaluated. We distribute our community detection code as a component of *NetworKit* [36], our open-source network analysis package, which is under continuous development.

II. Related Work

This section gives a short overview over related efforts. For a comprehensive overview of community detection in networks, we refer the interested reader to aforementioned surveys [32], [10]. Recent developments and results are also covered by the *10th DIMACS Implementation Challenge* [1].

Among efficient heuristics for community detection we can distinguish between those based on community agglomeration and those based on local node moves. Agglomerative algorithms successively merge pairs of communities so that an improvement with respect to community quality is achieved. In contrast, local movers search for quality gains which can be achieved by moving a node to the community of a neighbor.

A globally greedy agglomerative method known as CNM [8] runs in $O(md \log n)$ for graphs with n nodes and m edges, where d is the depth of the dendrogram of mergers and typically $d \sim \log n$. Among the few parallel implementations competing in the *DIMACS* challenge, Fagginger Auer and Bisseling [9] submitted an agglomerative algorithm with an implementation for both the GPU (using *NVIDIA CUDA*) and the CPU (using *Intel TBB*). The algorithm weights all edges with the difference in modularity resulting from a contraction of the edge, then computes a heavy matching M and contracts according to M. This process continues recursively with a hierarchy of successively smaller graphs. The matching procedure can adapt to star-like structures in the graph to avoid insufficient parallelism due to small matchings. In the challenge, the CPU implementation competed as CLU_TBB and proved exceptionally fast. Independently, Riedy et al. [30] developed a similar method, which follows the same principle but does not provide the adaptation to star-like structures. An improved implementation, labeled CEL in the following, corresponds to the description in [29].

Community detection by label propagation belongs to the class of local move heuristics. It has originally been described by Raghavan et al. [28]. Several variants of the algorithm exist, one of them (under the name *peer pressure clustering*) is due to Gilbert et al. [13]. The latter use the algorithm as a prototype application within a parallel toolbox that uses numerical algorithms for combinatorial problems. Unfortunately, Gilbert et al. report running times only for a different algorithm, which solves a very specific benchmark problem and is not applicable in our context. A variant of label propagation by Soman and Narang [33] for multicore and GPU architectures exists, which seeks to improve quality by re-weighting the graph.

A locally greedy multilevel-algorithm known as the *Louvain method* [5] combines local node moves with a bottom-up multilevel approach. Bhowmick and Srinivasan [3] presented a previous parallel version of the algorithm. According to their experimental results, our implementation is about four orders of magnitude faster. Noack and Rotta [31] evaluate similar sequential multilevel algorithms, which combine agglomeration with refinement.

Ovelgönne and Geyer-Schulz [27] apply the *ensemble learning* paradigm to community detection. They develop what they call the *Core Groups Graph Clusterer* scheme, which we adapt as the *Ensemble Preprocessing* (EPP) algorithm. They also introduce an iterated scheme in which the core communities are again assigned to an ensemble, creating a hierarchy of solutions/coarsened graphs until quality does not improve any more. Within this framework, they employ *Randomized Greedy* (RG), a variant of the aforementioned CNM algorithm. It avoids a loss in solution quality that arises from highly unbalanced community sizes. The resulting CGGC algorithm emerged as the winner of the Pareto part of the DIMACS challenge, which related quality to speed according to specific rules. Recently Ovelgönne [26] presented a distributed implementation (based on the big data framework *Hadoop*) of an ensemble preprocessing scheme using label propagation as a base algorithm. This implementation processes a 3.3 billion edge web graph in a few hours on a 50 machine Hadoop cluster [26, p. 73]. (Our *OpenMP*-based implementation of the similar EPP algorithm requires only 4 *minutes* on a shared-memory machine with 16 physical cores.)

From an algorithmic perspective, disjoint community detection is related to graph partitioning (GP). Although the problems are different in important aspects (unbalanced vs balanced blocks, unknown vs known number of blocks, different objectives), algorithms such as the Louvain method or PLMR bear conceptual resemblance to multilevel graph partitioners. Exploiting parallelism has been studied extensively for GP. Several established tools are discussed in recent surveys [4], [7], most of them for machines with distributed memory. Often employed techniques are parallel matchings for coarsening and parallel variants of Fiduccia-Mattheyses for local improvement. These techniques are at best partially helpful in our scenario since vanilla matching-based coarsening is ineffective on complex networks and distributed-memory parallelism is not necessary for us. Related to our work is a recent study on multithreaded GP by LaSalle and Karypis [20], who explore the design space of multithreaded GP algorithms. Their results provide interesting insights, but are not completely transferable to our scenario. Very recently they presented *Nerstrand* [21], a fast parallel community detection algorithm based on modularity maximization and the multilevel paradigm, using different aggregation schemes. Our work on PLP in this paper has also inspired a new parallel multilevel algorithm for partitioning massive complex networks [24].

We observe that most efficient disjoint community detection heuristics make use of agglomeration or local node moves, possibly in combination with multilevel or ensemble techniques. Both basic approaches can be adapted for parallelism, but this is currently the exception rather than the norm in our scenario. In this work we compare our own algorithms with the best currently available, sequential and parallel alike.

III. Algorithms

In this section we formulate and describe our parallel variants of existing sequential community detection algorithms, as well as ensemble techniques which combine them. Implementation details are also discussed. We use the following notation: A graph, the abstraction of a network data set, is denoted as $G = (V, E)$ with a node set V of size n and an edge set E of size m. In the following, edges $\{u, v\}$ are undirected and have weights $\omega : E \to \mathbb{R}^+$. The weight of a set of nodes is denoted as $\omega(E') := \sum_{\{u,v\} \in E'} \omega(u, v)$. A community detection solution $\zeta = \{C_1, \ldots, C_k\}$ is a partition of the node set V into disjoint subsets called communities. Equivalently, such a solution can be understood as a mapping where $\zeta(v)$ returns the community containing node v. For our implementation, the nodes have consecutive integer identifiers $id(v)$ and edges are pairs of node identifiers. A solution is represented as an array indexed by integer node identifiers and containing integer community identifiers.

A. Parallel Label Propagation (PLP)

a) Algorithm: Community detection by label propagation, as originally introduced by Raghavan et al. [28], extracts communities from a labelling $V \to \mathbb{N}$ of the node set. Initially, each node is assigned a unique label, and then multiple iterations over the node set are performed: In each iteration, every node adopts the most frequent label in its neighborhood (breaking ties arbitrarily). Densely connected groups of nodes thus agree on a common label, and eventually a globally stable consensus is reached, which usually corresponds to a good solution for the network. Label propagation therefore finds communities in nearly linear time: Each iteration takes $O(m)$ time, and the algorithm has been empirically shown to reach a stable solution in only a few iterations, though not mathematically proven to do so. The number of iterations seems to depend more on the graph structure than the size. More theoretical analysis is done by Kothapalli et al. [16]. The algorithm can be described as a locally greedy *coverage* maximizer, i.e. it tries to maximize the fraction of edges which are placed within communities rather than across. With its purely local update rule, it tends to get stuck in local optima of *coverage* which implicitly are good solutions with respect to modularity: A label is likely to propagate through and cover a dense community, but unlikely to spread beyond bottlenecks. The local update rule and the absence of global variables make label propagation well-suited for a parallel implementation.

Algorithm 1 denotes PLP, our parallel variant of label propagation. We adapt the algorithm in a straightforward way to make it applicable to weighted graphs. Instead of the most frequent label, the *dominant label* in the neighborhood is chosen, i.e. the label l that maximizes $\sum_{u \in N(v): \zeta(u) = l} \omega(v, u)$. We continue the iteration until the number of nodes which changed their labels falls below a threshold θ.

Algorithm 1: PLP: Parallel Label Propagation

Input: graph $G = (V, E)$
Result: communities $\zeta : V \to \mathbb{N}$

```
1  parallel for v ∈ V
2      ζ(v) ← id(v)
3  updated ← n
4  V_active ← V
5  while updated > θ do
6      updated ← 0
7      parallel for v ∈ {u ∈ V_active : deg(u) > 0}
8          l* ← arg max_l { Σ_{u∈N(v):ζ(u)=l} ω(v,u) }
9          if ζ(v) ≠ l* then
10             ζ(v) ← l*
11             updated ← updated + 1
12             V_active ← V_active ∪ N(v)
13         else
14             V_active ← V_active \ {v}
15 return ζ
```

b) Implementation: We make a few modifications to the original algorithm. In the original description [28], nodes are traversed in random order. Since the cost of explicitly randomizing the node order in parallel is not insignificant, we make this optional and rely on some randomization through parallelism otherwise. We also observe that forgoing randomization has a negligible effect on quality. We avoid unnecessary computation by distinguishing between active and inactive nodes. It is unnecessary to recompute the label weights for a node whose neighborhood labels have not changed in the previous iteration. Nodes which already have the heaviest label become inactive (Algorithm 1, line 14), and are only reactivated if a neighboring node is updated (line 12). We restrict iteration to the set of active nodes. Iterations are repeated until the number of nodes updated falls below a threshold value. The motivation for setting threshold values other than zero is that on some graph instances, the majority of iterations are spent on updating only a very small fraction of high-degree nodes (see Figure 12 in the supplementary material for an example). Since preliminary experiments have shown that time can be saved and quality is not significantly degraded by simply omitting these iterations, we set an update threshold of $\theta = n \cdot 10^{-5}$. Note that we do not use the termination criterion specified in [27] as it does not lead to convergence on some inputs. The original criterion is to stop when all nodes have the label of the relative majority in their neighborhood [28].

Label propagation can be parallelized easily by dividing the range of nodes among multiple threads which operate on a common label array. This parallelization is not free of race conditions, since by the time the neighborhood of a node u is evaluated in iteration i to set $\zeta_i(u)$, a neighbor v might still have the previous iteration's label $\zeta_{i-1}(v)$ or already $\zeta_i(v)$. The outcome thus depends on the order of threads. However, these race conditions are acceptable and even beneficial in an ensemble setting since they introduce random variations and increase base solution diversity. This also corresponds to *asynchronous updating*, which has been found to avoid oscillation of labels on bipartite structures [28]. When dealing with scale-free networks whose degree distribution follows a power law, assigning node ranges of equal size to each thread will lead to load imbalance as computational cost depends on the node degree. Instead of statically dividing the iteration among the threads, guided scheduling (with `#pragma omp parallel for schedule(guided)`) assigns node ranges of decreasing size from a queue to available threads. This way it can help to overcome load balancing issues, since threads processing large neighborhoods will receive fewer vertices in later phases of the dynamical assignment process. This introduces some overhead, but we observed that guided scheduling is generally superior to static parallelization for PLP and similar methods.

B. Parallel Louvain Method (PLM)

Algorithm: The *Louvain method* for community detection was first presented by Blondel et al. [5]. It can be classified as a locally greedy, bottom-up multilevel algorithm and uses modularity as the objective function. In each pass, nodes are repeatedly moved to neighboring communities such that the locally maximal increase in modularity is achieved, until the communities are stable. Algorithm 2 denotes this move phase. Then, the graph is coarsened according to the solution (by contracting each community into a supernode) and the procedure continues recursively, forming communities of communities. Finally, the communities in the coarsest graph determine those in the input graph by direct prolongation.

Computation of the objective function modularity is a central part of the algorithm. Let $\omega(u, C) := \sum_{\{u,v\}:v\in C} \omega(u, v)$ be the weight of all edges from u to nodes in community C, and define the *volume* of a node and a community as $vol(u) := \sum_{\{u,v\}:v\in N(u)} \omega(u, v) + 2 \cdot \omega(u, u)$ and $vol(C) := \sum_{u\in C} vol(u)$, respectively. The modularity of a solution is defined as

$$mod(\zeta, G) := \sum_{C\in\zeta} \left(\frac{\omega(C)}{\omega(E)} - \frac{vol(C)^2}{4\omega(E)^2} \right) \tag{III.1}$$

Note that the change in modularity resulting from a node move can be calculated by scanning only the local neighborhood of a node, because the difference in modularity when moving node $u \in C$ to community D is:

$$\Delta mod(u,\ C \to D) = \frac{\omega(u, D \setminus \{u\}) - \omega(u, C \setminus \{u\})}{\omega(E)} + \frac{(vol(C \setminus \{u\}) - vol(D \setminus \{u\})) \cdot vol(u)}{2 \cdot \omega(E)^2}$$

We introduce a shared-memory parallelization of the Louvain method (PLM, Algorithm 3) in which node moves are evaluated and performed in parallel instead of sequentially. This approach may work on stale data so that a monotonous modularity increase is no longer guaranteed. Suppose that during the evaluation of a possible move of node u other threads might have performed moves that affect the Δmod

scores of u. In some cases this can lead to a move of u that actually decreases modularity. Still, such undesirable decisions can also be corrected in a following iteration, which is why the solution quality is not necessarily worse. Working only on independent sets of vertices in parallel does not provide a solution since the sets would have to be very small, limiting parallelism and/or leading to the undesirable effect of a very deep coarsening hierarchy. Concerns about termination turned out to be theoretical for our set of benchmark graphs, all of which can be successfully processed with PLM. The community size resolution produced by PLM can be varied through a parameter γ in the range $[0, 2m]$, 0 yielding a single community, 1 being standard modularity and $2m$ producing singletons. Tuning this parameter is a possible practical remedy [18] against modularity's resolution limit.

Algorithm 2: move: Local node moves for modularity gain

Input: graph $G = (V, E)$, communities $\zeta : V \to \mathbb{N}$
Result: communities $\zeta : V \to \mathbb{N}$

repeat
  parallel for $u \in V$
    $\delta \leftarrow \max_{v \in N(u)} \{\Delta mod(u, \zeta(u) \to \zeta(v))\}$
    $C \leftarrow \zeta(\arg\max_{v \in N(u)} \{\Delta mod(u, \zeta(u) \to \zeta(v))\})$
    if $\delta > 0$ **then**
      $\zeta(u) \leftarrow C$
until ζ *stable*
return ζ

Algorithm 3: PLM: Parallel Louvain Method

Input: graph $G = (V, E)$
Result: communities $\zeta : V \to \mathbb{N}$

$\zeta \leftarrow \zeta_{\text{singleton}}(G)$
$\zeta \leftarrow \text{move}(G, \zeta)$
if ζ *changed* **then**
  $[G', \pi] \leftarrow \text{coarsen}(G, \zeta)$
  $\zeta' \leftarrow \text{PLM}(G')$
  $\zeta \leftarrow \text{prolong}(\zeta', G, G', \pi)$
return ζ

Implementation: The main idea of PLM (Algorithm 3) is to parallelize both the node move phase and the coarsening phase of the Louvain method. Since the computation of the Δmod scores is the most frequent operation, it needs to be very fast. We store and update some interim values, which is not apparent from the high-level pseudocode in Algorithm 3. An earlier implementation associated with each node a map in which the edge weight to neighboring communities was stored and updated when node moves occurred. A lock for each vertex v protected all read and write accesses to v's map since `std::map` is not thread-safe. Meant to avoid redundant computation, we later discovered that this introduces too much overhead (map operations, locks). Recomputing the weight to neighbor communities each time a node is evaluated turned out to be faster. The current implementation only stores and updates the volume of each community. An additional optimization to the PLM implementation eliminated the overhead associated with using an `std::map` to store for each node the weights of edges leading to neighboring communities. the mechanism was replaced by one `std::vector` for each of the p threads, leading to an acceleration of a factor of 2 on average, at the cost of a memory overhead of $O(p \cdot n)$. The former version (referred to as PLM*) can still be used optionally under tighter memory constraints.

Graph coarsening according to communities is performed in a straightforward way such that the nodes of a community in G are aggregated to a single node in G'. An edge between two nodes in G' receives as weight the sum of weights of inter-community edges in G, while self-loops preserve the weight of intra-community edges. A mapping π of nodes in the fine graph to nodes in the coarse graph is also returned. In earlier versions of PLM, the graph coarsening phase proved to be a major sequential bottleneck. We address this problem with a parallel coarsening scheme: Each thread first scans a portion of the edges in G and constructs a coarse graph G'_t of its own. These partial graphs are then combined into G' by processing each node of G' in parallel and merging the adjacencies stored in each G'_t.

C. Parallel Louvain Method with Refinement (PLMR)

Following up on the work by Noack and Rotta on multilevel techniques and refinement heuristics [31], we extend the Louvain method by an additional move phase after each prolongation. This makes it possible to re-evaluate node assignments in view of the changes that happened on the next coarser level, giving additional opportunities for modularity improvement at the cost of additional iterations over the node set in each level of the hierarchy. We denote the method and implementation as PLMR for *Parallel Louvain Method with Refinement*. We present a recursive implementation in Algorithm 4 which uses the same concepts as PLM.

Algorithm 4: PLMR: Parallel Louvain Method with Refinement

Input: graph $G = (V, E)$
Result: communities $\zeta : V \to \mathbb{N}$

$\zeta \leftarrow \zeta_{\text{singleton}}(G)$
$\zeta \leftarrow \text{move}(\zeta, G)$
if ζ *changed* **then**
  $[G', \pi] \leftarrow \text{coarsen}(G, \zeta)$
  $\zeta' \leftarrow \text{PLMR}(G')$
  $\zeta \leftarrow \text{prolong}(\zeta', G, G', \pi)$
  $\zeta \leftarrow \text{move}(\zeta, G)$
return ζ

D. Ensemble Preprocessing (EPP)

In machine learning, *ensemble learning* is a strategy in which multiple *base classifiers* or *weak classifiers* are combined to form a strong classifier. Classification in this context can be understood as deciding whether a pair of nodes should belong to the same community. We follow this general idea, which has been applied successfully to graph clustering before [27]. Subsequently, we describe an ensemble techniques

EPP. We also briefly describe algorithms for combining multiple base solutions.

Algorithm 5: EPP: *Ensemble Preprocessing*

Input: graph $G = (V, E)$, ensemble size b
Result: communities $\zeta : V \to \mathbb{N}$
parallel for $i \in [1, b]$
  $\zeta_i \leftarrow \mathsf{Base}_i(G)$
$\bar{\zeta} \leftarrow \mathsf{combine}(\zeta_1, \ldots, \zeta_b)$
$G', \pi \leftarrow \mathsf{coarsen}(G, \bar{\zeta})$
$\zeta' \leftarrow \mathsf{Final}(G')$
$\zeta \leftarrow \mathsf{prolong}(\zeta', G, G', \pi)$
return ζ

In a preprocessing step, assign G to an ensemble of base algorithms. The graph is then coarsened according to the *core communities* $\bar{\zeta}$, which represent the consensus of the base algorithms. Coarsening reduces the problem size considerably, and implicitly identifies the contested and the unambiguous parts of the graph. After the preprocessing phase, the coarsened graph G' is assigned to the final algorithm, whose result is applied to the input graph by prolongation. Our implementation of the ensemble technique EPP is agnostic to the base and final algorithms and can be instantiated with a variety of such algorithms. We instantiate the scheme with PLP as a base algorithm and PLMR as the final algorithm. Thus we achieve massive nested parallelism with several parallel PLP instances running concurrently in the first phase, and proceed in the second phase with the more expensive but qualitatively superior PLMR. This constitutes the EPP algorithm (Algorithm 5). We write EPP(b, Base, Final) to indicate the size of the ensemble b and the types of base and final algorithm.

Implementation: A consensus of $b > 1$ base algorithms is formed by combining the base solutions ζ_i in the following way: Only if a pair of nodes is classified as belonging to the same community in every ζ_i, then it is assigned to the same community in the core communities $\bar{\zeta}$. Formally, for all node pairs $u, v \in V$:

$$\forall i \in [1, b]\ \zeta_i(u) = \zeta_i(v) \quad \iff \quad \bar{\zeta}(u) = \bar{\zeta}(v). \tag{III.2}$$

We introduce a highly parallel combination algorithm based on *hashing*. With a suitable hash function $h(\zeta_1(v), \ldots, \zeta_b(v))$, the community identifiers of the base solutions are mapped to a new identifier $\bar{\zeta}(v)$ in the core communities. Except for unlikely hash collisions, a pair of nodes will be assigned to the same community only if the criterion above is satisfied. We use a relatively simple function called djb2 due to Bernstein,[2] which appears sufficient for our purposes. The use of a b-way hash function is fast due to a high degree of parallelism.

IV. Implementation and Experimental Setup

A. Framework and Settings

The language of choice for all implementations is C++ according to the C++11 standard, allowing us to use object-oriented and functional programming concepts while also compiling to native code. We implemented all algorithms on top of a general-purpose adjacency array graph data structure. Basically, it represents the adjacencies of each node by storing them in an `std::vector`, allowing for efficient insertions and deletions of nodes and edges. A high-level interface encapsulates the data structure and enables a clear and concise notation of graph algorithms. In particular, our interface conveniently supports parallel programming through parallel node and edge iteration methods which receive a function and apply it to all elements in parallel. Parallelism is achieved in the form of loop parallelization with *OpenMP*, using the `parallel for` directive with `schedule(guided)` where appropriate for improved load balancing.

We publish our source code under a permissive free software license to encourage reproduction, reuse and contribution by the community. Implementations of all community detection algorithms mentioned are part of *NetworKit* [36], our growing toolkit for network analysis.[3] The software combines fast parallel algorithms written in C++ with an interactive Python interface for flexible and interactive data analysis workflows.

For representative experiments we average quality and speed values over multiple runs in order to compensate for fluctuations. Table I provides information on the multicore platform used for all experiments.

	phipute1.iti.kit.edu
compiler	gcc 4.8.1
CPU	2 x 8 Cores: Intel(R) Xeon(R) E5-2680 0 @ 2.70GHz, 32 threads
RAM	256 GB
OS	SUSE 13.1-64

Table I: Platform for experiments

B. Networks

We perform experiments on a variety of graphs from different categories of real-world and synthetic data sets. Our focus is on real-world complex networks, but to add variety some non-complex and synthetic instances are included as well. The test set includes web graphs (`uk-2002`, `eu-2005`, `in-2004`, `web-BerkStan`), internet topology networks (`as-22july06`, `as-Skitter`, `caidaRouterLevel`), social networks (`soc-LiveJournal`, `fb-Texas84`, `com-youtube`, `wiki-Talk`, `soc-pokec`, `com-orkut`), scientific coauthorship networks (`coAuthorsCiteseer`, `coPapersDBLP`), a connectome graph (`con-fiber_big`), a street network (`europe-osm`) and synthetic graphs (`G_n_pin_pout`, `kron_g500-simple-logn20`, `hyperbolic-268M`). Therefore, we cover a range of graph-structural properties. Real-world complex networks are heterogeneous data sets, which makes it impossible to pick an ideal or generic instance from which to generalize. Our main test set is chosen such that it can be handled by competing codes as well. It contains 20 networks from different domains. With this test set we aim for generalizable results. Note

[2] hash functions: http://www.cse.yorku.ca/~oz/hash.html

[3] *NetworKit*: https://networkit.iti.kit.edu/

graph	n	m	maxdeg	comp	lcc
as-22july06	22963	48436	2390	1	0.3493
G_n_pin_pout	100000	501198	25	6	0.0040
caidaRouterLevel	192244	609066	1071	308	0.2016
coAuthorsCiteseer	227320	814134	1372	1	0.7629
fb-Texas84	36371	1590655	6312	4	0.1985
com-youtube	1157828	2987624	28754	22939	0.1725
wiki-Talk	2394385	4659565	100029	2555	0.1991
web-BerkStan	685231	6649470	84230	677	0.6343
as-Skitter	1696415	11095298	35455	756	0.2930
in-2004	1382908	13591473	21869	134	0.7013
coPapersDBLP	540486	15245729	3299	1	0.8111
eu-2005	862664	16138468	68963	1	0.6509
soc-pokec	1632804	22301964	14854	2	0.1223
soc-LiveJournal	4847571	43369619	20334	1876	0.3667
kron_g500-simple...	1048576	44619402	131503	253380	0.2096
con-fiber_big	591428	46374120	5166	727	0.6024
europe-osm	50912018	54054660	13	1	0.0012
com-orkut	3072627	117185083	33313	187	0.1735
uk-2002	18520486	261787258	194955	38359	0.6892
hyperbolic-268M	6710886	268851810	71585	1	0.7895
uk-2007-05	105896555	3301876564	975419	756936	0.743

Table II: Overview of graphs used in experiments

that the achievable modularity for a network depends on its size and inherent community structure, which may or may not be distinctive, and varies widely among the instances. The majority of test networks are taken from the collection compiled for the *10th DIMACS Implementation Challenge*[4] as well as the *Stanford Large Network Dataset Collection*[5] and are freely available on the web. They are undirected, unweighted graphs. Table II gives an overview over graph sizes as well as some structural features: A high maximum node degree (maxdeg) indicates possible load balancing issues. The number of connected components (comp) points to isolated single nodes or small groups of nodes. A high average local clustering coefficient (lcc) is an indicator for the presence of dense subgraphs. We evaluate solution quality and running time for all of our own algorithms as well as several relevant competitors on this set. For those algorithms that can process in reasonable time the largest real-world graph available to us, a web graph of the .uk domain with $m \approx 3.3 \cdot 10^9$, we add further experiments (see Section V-H). To measure strong scaling, we run our parallel algorithms on this web graph.

V. Experiments and Results

In this section we report on a representative subset of our experimental results for our different parallel algorithms, as well as competing codes. Figures 7 and 8 (as well as Figures 16 and 17 in the supplementary material) show running time and quality differences broken down by the networks of our test set. The bars of the charts are in ascending order of graph size. We have selected a diverse test set and show results for each network. The Pareto evaluation (Section V-F) then aims to condense this into a single performance score.

A. Parallel Label Propagation (PLP)

PLP is extremely fast and able to handle the large graphs easily. The "weak classifier" PLP is nonetheless able to detect an inherent community structure and produce a solution with reasonable modularity values, although it cannot distinguish communities in a Kronecker graph, which has a very weak community structure. To demonstrate strong scaling behavior, we apply PLP to the large `uk-2007-05` web graph and increase the number of threads from 1 to 32 (Figure 1). (Weak scaling results on PLP and PLM are shown in Figure 10.) A speedup of about factor 8 is achieved when scaling from 1 to 32 threads. Note that we have only 16 physical cores and the step from 16 to 32 threads implies hyperthreading, so that a lower speedup is expected. Our results indicate that PLP can benefit from increased parallelism. Figure 13 in the supplementary material breaks running times down by iteration, showing that the vast majority of time is spent in the first couple of iterations.

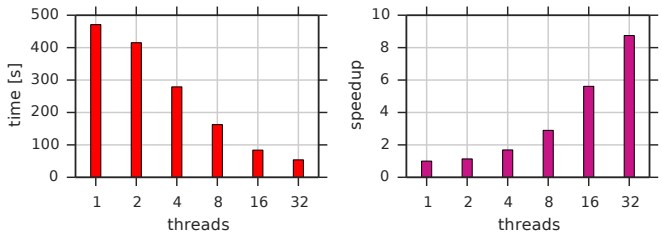


Figure 1: PLP strong scaling on the `uk-2007-05` web graph

B. Parallel Louvain Method (PLM)

For PLM we observe only small deviations in quality between single-threaded and multi-threaded runs, supporting the argument that the algorithm is able to correct undesirable decisions due to stale data. PLM detects communities with relatively high modularity in the majority of networks. Even large instances are processed in no more than a few minutes. Figure 2 shows the scaling behavior of PLM. Since both the node move phase and the coarsening phase have been parallelized, PLM profits from increased parallelism as well, achieving a speedup of factor 9 for 32 threads. In comparison to PLP (Figure 7b), we observe that PLP can solve instances in only half the time required by PLM, but at a significant loss of modularity. As discussed in Sec. VI, the communities detected by the two algorithms can be markedly different. Because the Louvain method for community detection is well-known and accepted, we choose the performance of PLM as our baseline (Figure 7a) and present quality and running time of other algorithms relative to PLM.

C. Parallel Louvain Method with Refinement (PLMR)

As shown by Figure 7c, adding a refinement phase generally leads to a (sometimes significant) improvement in modularity. This improvement is paid for by a small increase in running time. The results indicate that our proposed extension of the original Louvain method by a refinement phase can efficiently increase solution quality. We also evaluate the scaling behavior of each phase of the PLMR algorithm. In Figure 3 a yellow bar indicates the running time on the finest graph while the

[4] *DIMACS* collection: http://www.cc.gatech.edu/dimacs10/downloads.shtml
[5] *Stanford* collection: http://snap.stanford.edu/data/index.html

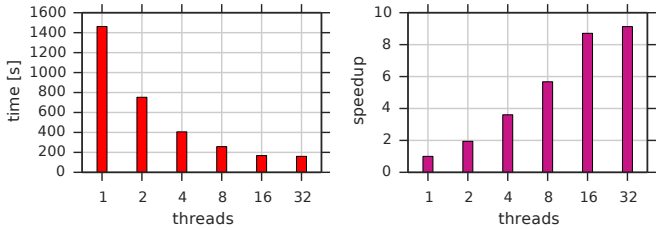


Figure 2: PLM strong scaling on the uk-2007-05 web graph

red bar stops at the total running time of the phase. Time spent on the finest graph clearly dominates all running times. Our experiments show that the move and refinement phases scale well with the number of threads, while the coarsening phase only partially profits from parallelization. The results on this graph are representative for the trend of the scaling behavior for the algorithm's phases: Figure 4 shows speedup factors for each of the phases, aggregated over the test set of 20 graphs.

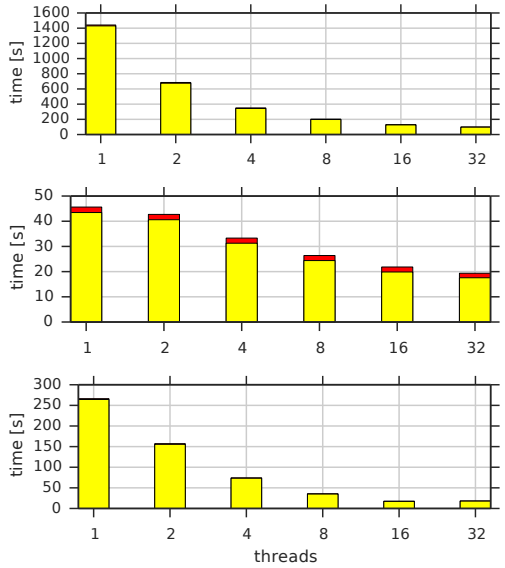


Figure 3: PLMR strong scaling of the move, coarsening and refinement phases (top to bottom) on uk-2007-05

D. Ensemble Preprocessing (EPP)

Figure 15 in the supplementary material demonstrates the effectiveness of the ensemble approach. Results were generated by an EPP instance with a 4-piece PLP ensemble and PLMR as final algorithm in comparison to a single PLP instance. We observe that the approach of EPP pays off in the form of improved modularity on most instances, exploiting differences in the base solutions and spending extra time on classifying contested nodes. For larger networks, this comes at a cost of about 5 times the running time of PLP alone. It also becomes clear that for small networks the approach does not pay off as running time becomes dominated by the overhead of the ensemble scheme. In comparison to PLM (Figure 7d), the ensemble approach can be slightly faster on some networks, but quality is slightly worse in most cases. We conclude that

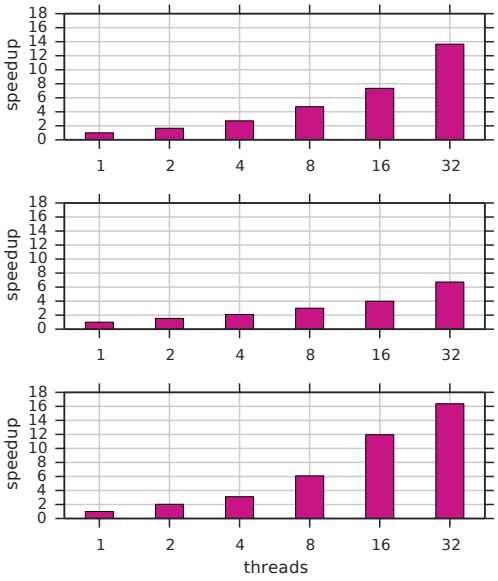


Figure 4: PLMR strong scaling of the move, coarsening and refinement phases (top to bottom) – speedup factors aggregated

the ensemble technique EPP is effective in improving on the quality of a single algorithm. While somewhat lower in modularity, the communities detected are similar (see Sec. VI) to those of the Louvain method. In practice, our acceleration of the PLM algorithm have made the ensemble approach less relevant.

E. Comparison with State-of-the-Art Competitors

In this section we present results for an experimental comparison with several relevant competing community detection codes. These are mainly those which excelled in the *DIMACS* challenge either by solution quality or time to solution: The agglomerative algorithms CLU_TBB[6] and RG, as well as CGGC and CGGCi[7], ensemble algorithms based on RG. We also include the widely used original sequential Louvain[8] implementation, as well as the agglomerative algorithm CEL. In contrast to the *DIMACS* challenge, we run all codes on the same multicore machine (Tab. I) and measure time to solution for sequential and parallel ones alike.

a) Louvain: Although not submitted to the *DIMACS* competition, the original sequential implementation of the Louvain method is still relatively fast (Figure 8a). The marginally different modularity values in comparison to PLM may be caused by subtle differences in the implementation. For example, Louvain explicitly randomizes the order in which nodes are visited, while we rely on implicit randomization through parallelism. For the smallest graphs, running time values are missing because the implementation reported a running time of zero. Louvain eventually falls behind the parallel algorithm for large graphs, confirming that the overhead and

[6]CLU_TBB http://www.staff.science.uu.nl/~faggi101/
[7]RG etc: http://www.umiacs.umd.edu/~mov/
[8]Louvain https://sites.google.com/site/findcommunities/

complexity introduced by parallelism is eventually justified when we target massive datasets.

b) CLU_TBB and CEL: CLU_TBB, one of the few parallel entries in the *DIMACS* competition, is a very fast implementation of agglomerative modularity maximization, solving the larger instances more quickly than PLM (Figure 8b). Qualitatively however, PLM is clearly superior on most networks. Both in terms of modularity and running time, CLU_TBB occupies a middle ground between PLP and PLM, and is qualitatively very similar to our ensemble algorithm EPP. CEL, as another fast parallel program, produced consistently and significantly worse modularity than PLM, failed to produce a solution on some graphs, and is not as fast as PLP.

c) RG, CGGC and CGGCi: Ovelgönne and Geyer-Schulz entered the DIMACS challenge with an ensemble approach conceptually similar to what we have developed in this paper. Their base algorithm is the sequential agglomerative RG, and two ensemble variants exist: CGGC implements anensemble technique very similar to EPP, while CGGCi iterates the approach. The RG algorithm achieves a high solution quality, surpassing PLM by a small margin on most networks (Figure 8c). Quality is again slightly improved by the ensemble approach CGGC and its iterated version CGGCi (Figure 8d, and 16a in the SM), with the latter surpassing any other heuristic known to us. However, all three are very expensive in terms of computation time, often taking orders of magnitude longer than PLM. We consider running times of several hours for many of our networks no longer viable for the scenario we target, namely interactive data analysis on a parallel workstation.

F. Pareto Evaluation

We have so far presented results broken down by data set to stress that observed effects may vary strongly from one network to another, a sign of the heterogeneity of real-world complex networks. Additionally, we want to give a condensed picture of the results. For this purpose we use the previous experimental data to compute a score for running time and solution quality. The time score is the geometric mean of running time ratios over our test set of networks with the running time of PLM as the baseline, while the modularity score is the arithmetic mean of absolute modularity differences. Figure 5 shows the resulting points. It becomes clear that all algorithms except CEL and EPP are placed on or close to the Pareto frontier. PLP is unrivaled in terms of time to solution, but solution quality is suboptimal. In the middle ground between label propagation and Louvain method, the parallel CLU_TBB achieves about the same modularity but beats the ensemble approach in terms of speed. PLM and PLMR emerge as qualitatively strong and fast candidates, closest to the lower right corner. (Their more memory-efficient implementation PLM* is about a factor of 2 slower.) It is also evident that our extended version PLMR can improve solution quality for a small computational extra charge. We recommend both PLM and PLMR as the default algorithms for parallel community detection in large networks. The original sequential implementation of the Louvain method is thus no longer on the Pareto frontier since it cannot benefit from multicore systems. RG and its ensemble combinations have the best modularity scores by a narrow margin, while they are by far the most computationally expensive ones, which places them outside of the application scenario we target.

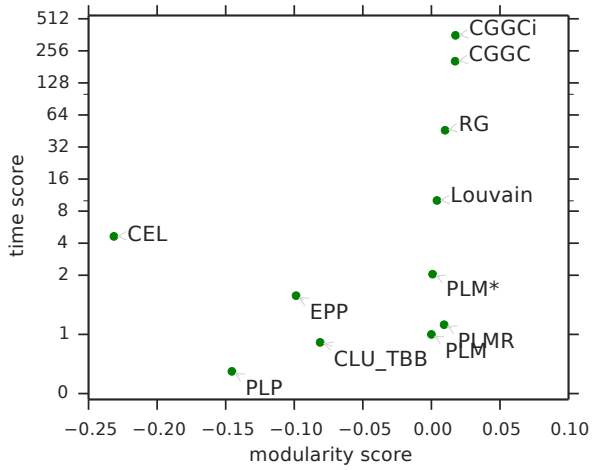


Figure 5: Pareto evaluation of community detection algorithms

G. LFR Benchmark

The LFR benchmark [19] is an established method for evaluating community detection algorithms: A generator produces graphs that resemble real complex networks and contain dense communities which are the more sparsely connected the lower the mixing parameter μ. Algorithm performance is measured as the accuracy in recognizing the ground truth communities supplied by the generator, in view of increasing difficulty (μ). Although there are real-world networks that come with supposed ground truth communities (e. g. interest-based groups of online social networks in the SNAP collection), we consider only a synthetic ground truth reliable enough for our purposes. In Figure 6 we plot the agreement (graph-structural Rand index, where 1 is complete agreement) between detected and ground truth communities for our algorithms, and show that the PLM method is able to detect the ground truth even with strong noise ($\mu = 0.8$), while PLP (and hence EPP) is somewhat less robust.

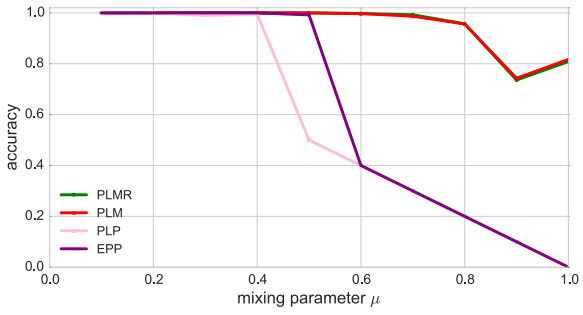
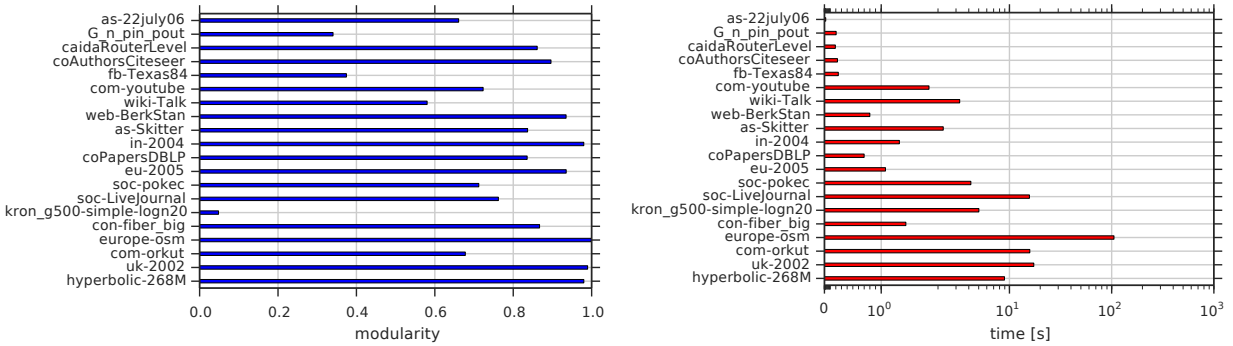
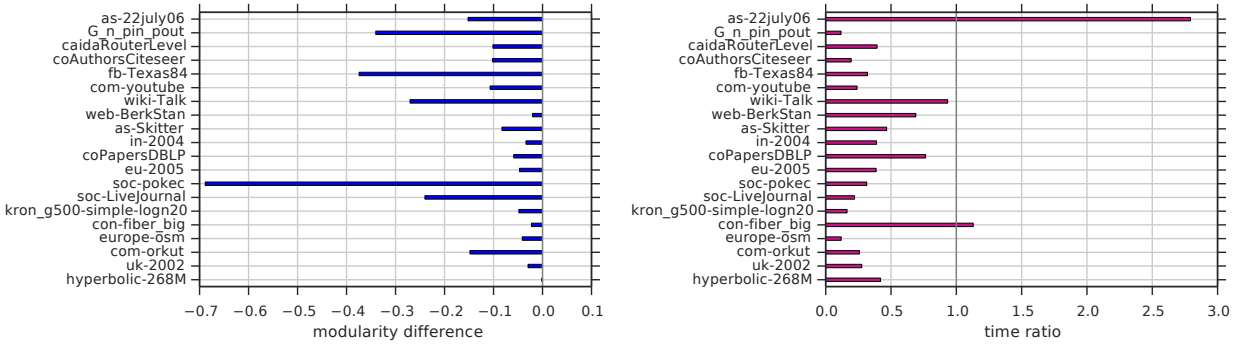


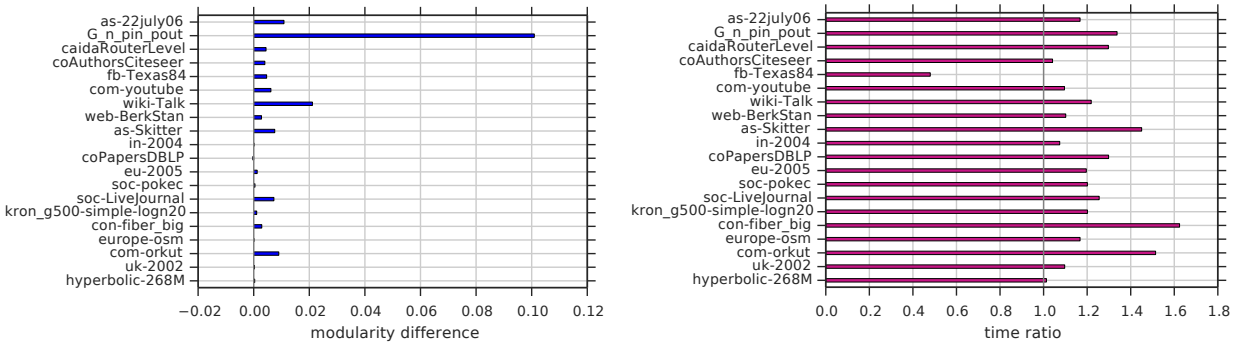
Figure 6: LFR benchmark ($n = 10^5$): accuracy in recognizing ground truth while increasing inter-community edges



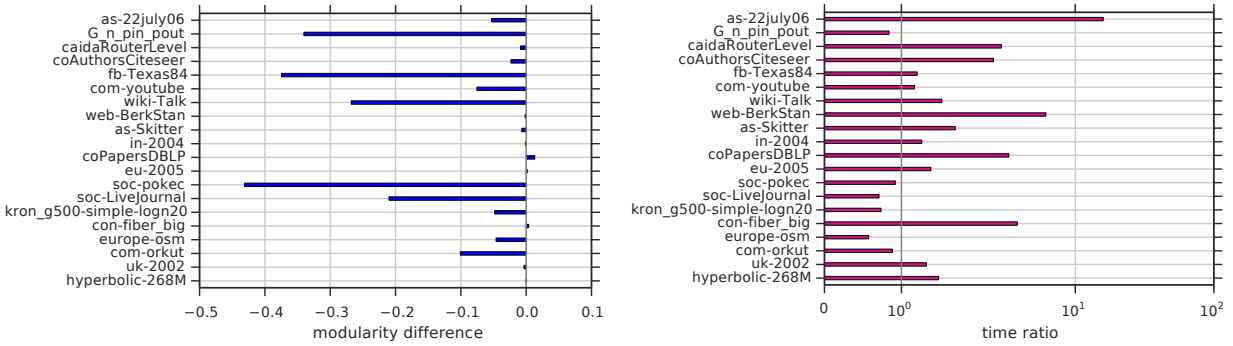
(a) PLM : absolute quality and speed serve as baseline for comparison



(b) PLP

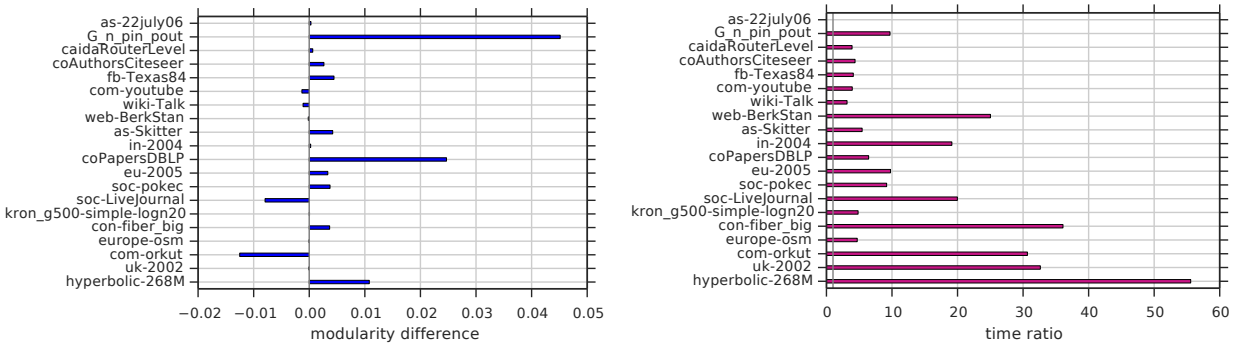


(c) PLMR

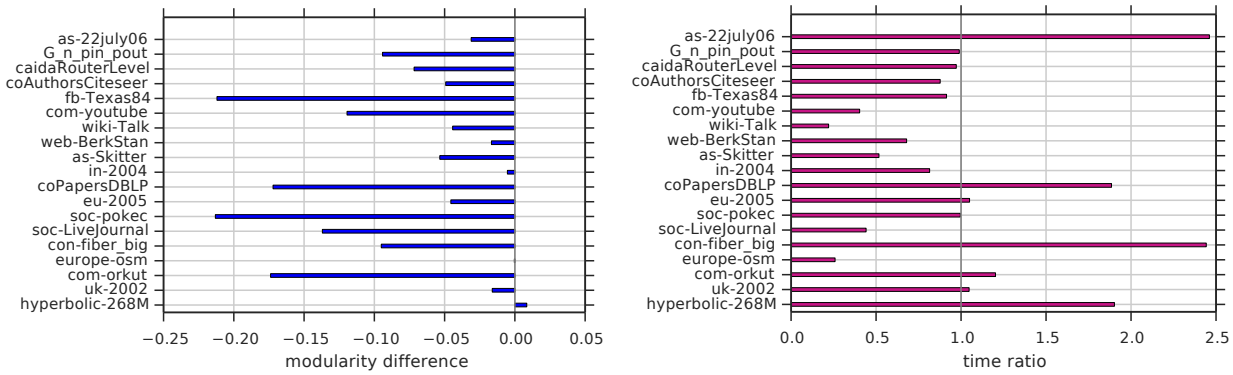


(d) EPP(4, PLP, PLMR)

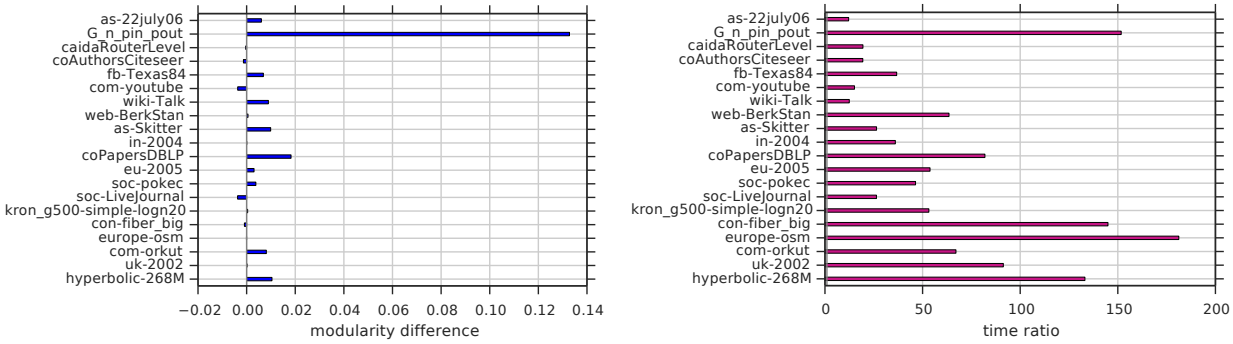
Figure 7: Performance of our algorithms in comparison: PLM serves as the baseline. 32 threads used.



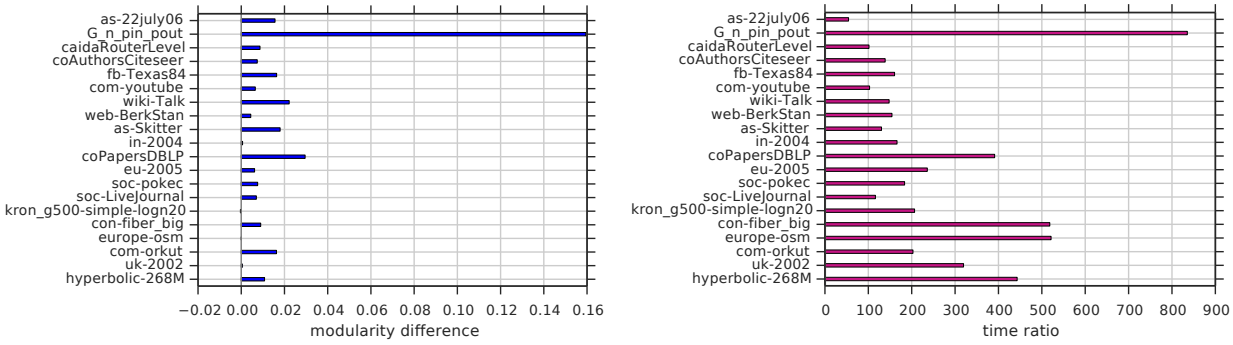
(a) Louvain



(b) CLU_TBB



(c) RG



(d) CGGC

Figure 8: Performance of competitors relative to baseline PLM. 32 threads used for CLU_TBB.

H. One More Massive Network

In addition to the experiments that went into the Pareto evaluation, we run our parallel algorithms on the web graph `uk-2007-05`, at about 3.3 billion edges the largest real-world data set currently available to us. CLU_TBB fails at reading the input file. This leaves us with five of our own parallel algorithms for Figure 9: EPP(4,PLP,PLMR) takes about 219 seconds, while PLM requires about 156 seconds to arrive at a slightly higher modularity. As expected, PLP is by far the fastest algorithm and terminates in less than a minute. If a certain modularity loss (here 0.02) is acceptable, PLP is also an appropriate choice for quickly detecting communities in billion-edge networks. The processing rate for PLP is over 53M edges/second and over 21M edges/second for PLM with respect to a complete run of each algorithm. These rates confirm the suitability of our algorithms for analyzing massive complex networks on a commodity shared-memory server.

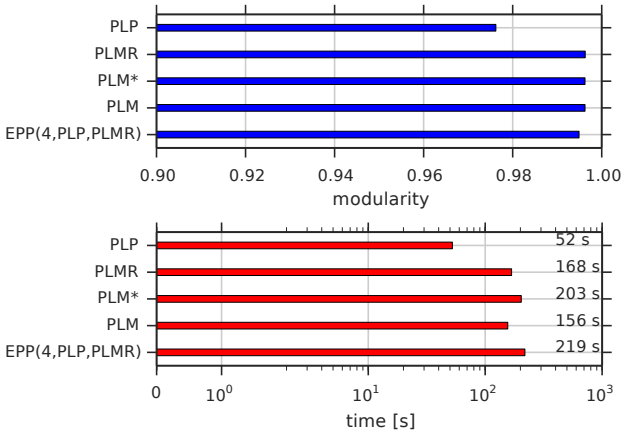


Figure 9: Modularity and running time at 32 threads for our parallel algorithms on the massive web graph `uk-2007-05`

I. Weak Scaling

For weak scaling experiments, we use a series of synthetic graphs where each graph has twice the size of its predecessor (from $\log m = 25 \ldots 30$), and double the number of threads simultaneously from 1 to 32. The graphs were created using a generator [22] based on a unit-disk graph model in hyperbolic geometry [17] (HUD), which produces both a power law degree distribution and distinctive dense communities. Figure 10 shows the results of weak scaling experiments for PLP and PLM. It must be noted that perfect scaling cannot be expected due to the complex structure of the input. The results of the respective last column have been obtained with hyperthreading, which explains the steeper increase. Figure 14 in the supplementary material show results for additional weak scaling experiments on synthetic graphs generated with the R-MAT model.

VI. Qualitative Aspects

In this work we concentrate on achieving a good tradeoff between high modularity, a widely accepted quality measure

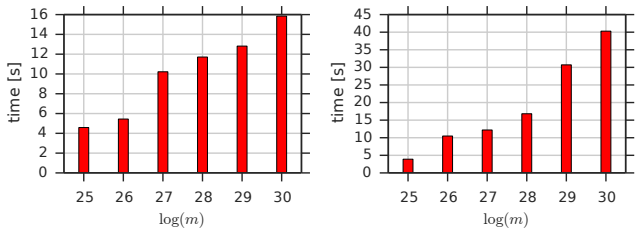


Figure 10: PLP (left) and PLM (right) weak scaling on the series of HUD graphs

for community detection, and low running time. Ideally one should also look for further validation of the detected communities beyond good modularity. This is a difficult task for several reasons. For most networks, we do not have a reliable ground-truth partition, especially because community structure is likely a multi-factorial phenomenon in real networks. Our task is to uncover the hidden community structure of the network. In order to know whether we have succeeded in this data mining task, we would have to check whether the solution helps us to formulate hypotheses to predict and explain real-world phenomena on the basis of network data. Whether one solution is more appropriate than another may strongly depend on the domain of the network. Domain-specific validation of this kind goes beyond the scope of this paper as we focus on parallelization aspects. Also, most sequential counterparts of our algorithms have been validated before, see e. g. [5].

However, we give an example to illustrate differences between our algorithms in a more qualitative way. Coarsening the input graph according to the detected communities yields a *community graph*, which we then visualize by drawing the size of nodes proportional to the size of the respective community. Figure 11 shows community graphs for the `PGPgiantcompo` graph, a social network and web of trust resulting from signatures on PGP keys. From top to bottom, the solutions were produced by PLP, PLM, PLMR and EPP(4, PLP, PLMR). It is apparent that PLP has a much finer resolution and detects ca. 1000 small communities. This is true for most of our data sets, but the inverse case also appears. On this network, higher modularity is associated with coarser resolution. PLM, PLMR and EPP(4, PLP, PLMR) have a very similar resolution and divide the network into ca. 100 communities. While `PGPgiantcompo` is admittedly a very small graph, this example shows how community detection can help to reduce the complexity of networks for visual representation.

VII. Conclusion and Future Work

We have developed and implemented several parallel algorithms for community detection, a common and challenging task in network analysis. Successful techniques and parameter settings have been identified in extensive experiments on synthetic and real-world networks. They include three standalone parallel algorithms, all of which are placed on the Pareto frontier with respect to running time and modularity in an experimental comparison with other state-of-the-art

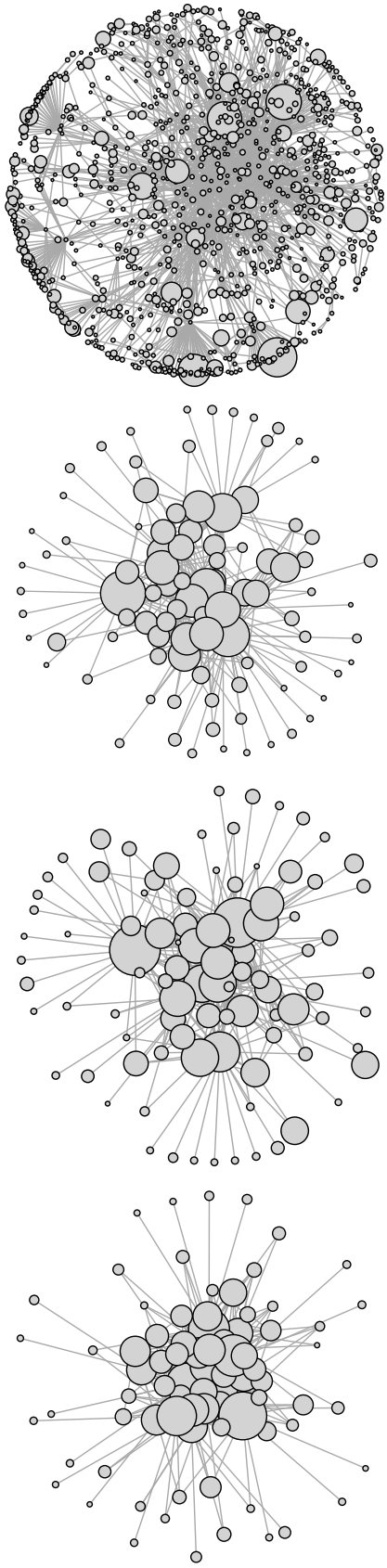


Figure 11: Community graphs of the PGPgiantcompo web of trust for (top to bottom) PLP, PLM, PLMR and EPP(4, PLP, PLMR)

implementations. While the PLP label propagation algorithm is extremely fast, its solution might not always be satisfactory for some applications. PLM is to the best of our knowledge the first parallel variant of the established Louvain algorithm which can handle massive inputs. On our machine, it detects high-quality communities in a network with 3.3 billion edges in under 3 minutes using 32 threads. Achieving significant parallel speedups over the frequently used sequential algorithm, it can accelerate analysis workflows now and even further on future multicore systems. Our modification PLMR of this method adds a refinement phase which enhances modularity for a small increase in running time.

Our implementations are published as a component of *NetworKit* [36], an open-source package of performant implementations for established and novel network analysis algorithms. We invite researchers in algorithm engineering and network science to benefit from our software development efforts and consider contribution to the project. *NetworKit* is under active development by several researchers and may be extended by additional community detection methods in the future, e. g. considering overlapping communities as well.

Acknowledgements: This work was partially supported by the project *Parallel Analysis of Dynamic Networks — Algorithm Engineering of Efficient Combinatorial and Numerical Methods*, which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg. We thank Pratistha Bhattarai for her contributions to the experimental study, Michael Hamann for optimizations to PLM, and numerous contributors to the *NetworKit* project.

REFERENCES

[1] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[2] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the community detection resolution limit with edge weighting," *Physical Review E*, vol. 83, no. 5, p. 056119, 2011.

[3] S. Bhowmick and S. Srinivasan, "A template for parallelizing the louvain method for modularity maximization," in *Dynamics On and Of Complex Networks, Volume 2*. Springer, 2013, pp. 111–124.

[4] C. Bichot and P. Siarry, *Graph Partitioning*, ser. ISTE. Wiley, 2011.

[5] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008.

[6] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "On modularity clustering," *IEEE Trans. Knowledge and Data Engineering*, vol. 20, no. 2, pp. 172–188, 2008.

[7] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," *arXiv preprint arXiv:1311.3144*, 2014.

[8] A. Clauset, M. E. Newman, and C. Moore, "Finding community structure in very large networks," *Physical review E*, vol. 70, no. 6, p. 066111, 2004.

[9] B. O. Fagginger Auer and R. H. Bisseling, "Graph coarsening and clustering on the GPU," in *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[10] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75 – 174, 2010.

[11] S. Fortunato and M. Barthelemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[12] U. Gargi, W. Lu, V. S. Mirrokni, and S. Yoon, "Large-scale community detection on youtube for topic discovery and exploration." in *International Conference on Weblogs and Social Media*, 2011.

[13] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer, 2007, pp. 260–269.

[14] M. Girvan and M. Newman, "Community structure in social and biological networks," *Proc. of the National Academy of Sciences*, vol. 99, no. 12, p. 7821, 2002.

[15] P. F. Jonsson, T. Cavanna, D. Zicha, and P. A. Bates, "Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis," *BMC Bioinformatics*, vol. 7, p. 2, 2006.

[16] K. Kothapalli, S. Pemmaraju, and V. Sardeshmukh, "On the analysis of a label propagation algorithm for community detection," in *Distributed Computing and Networking*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7730, pp. 255–269.

[17] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá, "Hyperbolic geometry of complex networks," *Physical Review E*, vol. 82, p. 036106, Sep 2010.

[18] R. Lambiotte, "Multi-scale modularity in complex networks," in *Modeling and optimization in mobile, ad hoc and wireless networks (WiOpt), 2010 Proceedings of the 8th International Symposium on*. IEEE, 2010, pp. 546–553.

[19] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, no. 4, p. 046110, 2008.

[20] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. 27th IEEE Intl. Symposium on Parallel and Distributed Processing (IPDPS 2013)*. IEEE Computer Society, 2013, pp. 225–236.

[21] ——, "Multi-threaded modularity based graph clustering using the multilevel paradigm," *Journal of Parallel and Distributed Computing*, 2014, http://dx.doi.org/10.1016/j.jpdc.2014.09.012.

[22] M. von Looz, C. L. Staudt, H. Meyerhenke, and R. Prutkin, "Fast generation of dynamic complex networks with underlying hyperbolic geometry," Karlsruhe Institute of Technology, Karlsruhe Reports in Informatics 2014,14, November 2014. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043881

[23] P. D. Meo, E. Ferrara, G. Fiumara, and A. Provetti, "Mixing local and global information for community detection in large networks," *J. Comput. Syst. Sci.*, vol. 80, no. 1, pp. 72–87, 2014.

[24] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," in *Proc. 29th IEEE Intl. Symposium on Parallel and Distributed Processing (IPDPS)*, 2015, to appear.

[25] M. Müller-Hannemann and S. Schirra, Eds., *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*, ser. Lecture Notes in Computer Science, vol. 5971. Springer, 2010.

[26] M. Ovelgönne, "Distributed community detection in web-scale networks," in *Proc. Advances in Social Networks Analysis and Mining (ASONAM '13)*, 2013, pp. 66–73.

[27] M. Ovelgönne and A. Geyer-Schulz, "An ensemble learning strategy for graph clustering," in *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[28] U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks," *Physical Review E*, vol. 76, no. 3, p. 036106, 2007.

[29] E. J. Riedy and D. A. Bader, "Multithreaded community monitoring for massive streaming graph data," in *Workshop on Multithreaded Architectures and Applications (MTAAP'13)*, 2013, pp. 1646–1655.

[30] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Graph Partitioning and Graph Clustering*, ser. Contemporary Mathematics. AMS and DIMACS, 2013, no. 588.

[31] R. Rotta and A. Noack, "Multilevel local search algorithms for modularity clustering," *J. Exp. Algorithmics*, vol. 16, pp. 2.3:2.1–2.3:2.27, Jul. 2011.

[32] S. E. Schaeffer, "Graph clustering," *Computer Science Review*, vol. 1, no. 1, pp. 27–64, 2007.

[33] J. Soman and A. Narang, "Fast community detection algorithm with GPUs and multicore architectures," in *Proc. 25th IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 568–579.

[34] C. Staudt, A. Schumm, H. Meyerhenke, R. Görke, and D. Wagner, "Static and dynamic aspects of scientific collaboration networks," in *Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*. IEEE, 2012, pp. 522–526.

[35] C. L. Staudt and H. Meyerhenke, "Engineering high-performance community detection heuristics for massive graphs," in *proceedings of the 2013 International Conference on Parallel Processing*. Conference Publishing Services (CPS), 2013.

[36] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "NetworKit: An interactive tool suite for high-performance network analysis," *arXiv preprint arXiv:1403.3005*, 2014.

[37] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*. ACM, 2012, p. 3.

Christian L. Staudt received his Diplom degree in computer science from Karlsruhe Institute of Technology (KIT) in 2012. He is currently a researcher and PhD candidate in the Parallel Computing Group, Institute of Theoretical Informatics, KIT. His research focuses on developing efficient algorithms and software for the analysis of large complex networks. Beyond that, he is interested in how network analysis methods can enable the study of complex systems in various domains.

Henning Meyerhenke is an Assistant Professor (Juniorprofessor) in the Institute of Theoretical Informatics at Karlsruhe Institute of Technology (KIT), Germany, since October 2011. Before joining KIT, Henning was a Postdoctoral Researcher in the College of Computing at Georgia Institute of Technology (USA) and at the University of Paderborn (Germany) as well as a Research Scientist at NEC Laboratories Europe. Henning received his Diplom degree in Computer Science from Friedrich-Schiller-University Jena, Germany, in 2004 and his Ph.D. in Computer Science from the University of Paderborn, Germany, in 2008. Dr. Meyerhenke's main research interests are efficient sequential and parallel algorithms and tools for applications in network analysis, combinatorial scientific computing, and the life sciences.