Engineering High-Performance Community Detection Heuristics for Massive Graphs

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, high-performance analytics algorithms and software tools are necessary. One common graph analytics kernel is 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 strengths of 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 exceeds 10M edges/second, making it suitable for massive data streams. We recommend the parallel Louvain method and our variant with refinement as both qualitatively strong and relatively fast. Moreover, our fast ensemble algorithm yields a good tradeoff between quality and speed for community detection in very large networks.

Keywords: Community detection, graph clustering, high-performance network analysis, parallel algorithm engineering, parallel Louvain method

I. INTRODUCTION

The data volume produced by electronic devices is growing at an enormous rate. Networks are an important class of such data, and we use graphs as their mathematical abstraction. A few examples: Online social networks are increasingly popular, the largest being Facebook with more than 700 million daily active users. The WWW forms a network of hyperlinked webpages in excess of 30 billion nodes. To be able to analyze network-structured data of this magnitude in near real-time, algorithms and hardware have to keep up with these data volumes and rates. However, only few algorithms are capable of handling such massive inputs. A particular challenge is not only the size of the data, but also its structure. Complex networks, in contrast to more regular data sets like meshes, have topological features which pose computational challenges: 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. This increases the importance of structural network analysis. As a result, costly algorithms can be applied only to certain relevant parts of the network after they have been identified in a precedent analysis.

In this work, we approach the task of community detection in networks, also known as graph clustering, with a focus on large networks. Among manifold applications, community detection has been used to counteract search engine rank manipulation, to discover scientific communities in publication databases, to identify functional groups of proteins in cancer research, and to organize content on social media sites. 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 and Fortunato. 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. In the following we determine non-overlapping communities, i.e. a partition of the node set. The quality measure modularity 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 nor alternatives, but has emerged as a well-accepted measure of community quality. This makes modularity our measure of choice for evaluating our results. While optimizing modularity is NP-hard, efficient heuristics have been introduced which explicitly increase modularity. For the time being, most of them are sequential and only a few parallel approaches exist.

A. Motivation

For graphs with millions to billions of edges, only (near) linear-time community detection algorithms can be considered in practice. Several fast methods have been developed in recent years. There is, however, 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

1A preliminary version of this paper appeared in Proceedings of the 2013 International Conference on Parallel Processing.
2Facebook key facts: http://newsroom.fb.com/Key-Facts
Partitioning and Graph Clustering. 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 performance. Ten solver families were submitted (with a total of 15 different implementations) for optimizing modularity. However, only two of them relied on parallel processing and only very few could handle the largest graphs with billions of edges in a reasonable amount of time.

Accordingly, our objective is the development and implementation of parallel community detection heuristics which are able to handle massive graphs quickly while also producing a high-quality solution. In the following, the data sets and results from the DIMACS challenge will serve as a major benchmark for our own work presented here. 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.

B. Methods

We implement three parallel algorithms, each of which can be used as a relatively efficient standalone solver. Label propagation 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 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 their strengths. 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. Finally, the communities found in the coarse graph are prolonged to the input graph. Within this extensible framework, which we call the ensemble preprocessing method (EPP), we apply PLP as base algorithms and PLM/PLMR as the final algorithm. We combine multiple base solutions with a highly parallel hashing-based scheme, implicitly finding nodes at the boundary of communities whose affiliation is disputable.

C. Capabilities

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 exceeds $10^7$ edges per second for many 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 combines the advantages of PLP and PLM, and yields a good quality-speed tradeoff 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. VI-E and VI-F): Three of our algorithms are Pareto-optimal and the remaining one seems to be the most effective on very large networks.

Our community detection algorithm framework, written in C++, is flexible, extensible and supports rapid iteration between design, implementation and testing required for algorithm engineering [15]. In this work, we focus on specific configurations of algorithms, but novel combinations can be quickly evaluated in the future. We distribute our community detection code as a component of NetworKit, our opensource 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 the aforementioned surveys [22], [7]. 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 [5] 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 [6] 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. [19], [20] developed a similar implementation, which follows the same principle but does not provide the adaptation to star-like structures.

Community detection by label propagation belongs to the class of local move heuristics. It has originally been described by Raghavan et al. [18], and several variants of the algorithm exist. One of these variants under the name peer pressure clustering is due to Gilbert et al. [10], who use the algorithm as a prototype application within a parallel toolbox that uses numerical algorithms for combinatorial problems. Unfortunately, the latter 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 [23] for multicore and GPU architectures exists, which seeks to improve quality by re-weighting the graph.


Ovelgönne and Geyer-Schulz [17] 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 challenge, which related quality to speed according to specific rules. Recently Ovelgönne [16] 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 [16] p. 10. (Our OpenMP-based implementation of the similar EPP algorithm requires only a few minutes on one shared-memory machine with 16 physical cores.)

We observe that most efficient 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 this work we compare our own algorithms with the best currently available, sequential and parallel alike.

### III. Preliminaries

We denote a graph, the abstraction of a network data set, 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 $\omega : E \rightarrow \mathbb{R}^+$. 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 $\zeta : V \rightarrow \mathcal{P}(V)$ where $\zeta(v)$ returns the community containing node $v$. For our implementation, the nodes have consecutive integer identifiers $id(v)$ in the range $[0, n - 1]$ and edges are pairs of node identifiers. A solution is represented as an array of size $n$, indexed by integer node identifiers and containing integer community identifiers, i.e. a mapping $\zeta : \mathbb{N} \rightarrow \mathbb{N}$.

### IV. 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.

#### A. Parallel Label Propagation (PLP)

**a) Algorithm:** Community detection by label propagation, as originally introduced by Raghavan et al. [18], extracts communities from a labelling $V \rightarrow \mathbb{N}$ 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 and uniformly). 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 near 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. [14]. PLP 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 $\sum_{u \in N(v) ; \zeta(u) = l} \omega(v,u)$. 
Iteration continues until the number of nodes which changed their labels falls below a threshold $\theta$.

### Algorithm 1: PLP: Parallel Label Propagation

**Input:** graph $G = (V, E)$

**Result:** communities $\zeta : V \to \mathbb{N}$

```plaintext
parallel for $v \in V$
  $\zeta(v) \leftarrow id(v)$
updated $\leftarrow n$
$V_{active} \leftarrow V$
while updated $> \theta$
do
  updated $\leftarrow 0$
  parallel for $v \in \{u \in V_{active} : \deg(u) > 0\}$
    $I^* \leftarrow \arg \max \left\{ \sum_{u \in N(v) : \zeta(u) = l} \omega(v, u) \right\}$
    if $\zeta(v) \neq I^*$ then
      $\zeta(v) \leftarrow I^*$
      updated $\leftarrow$ updated + 1
      $V_{active} \leftarrow V_{active} \cup N(v)$
    else
      $V_{active} \leftarrow V_{active} \setminus \{v\}$
return $\zeta$
```

### b) Implementation:
We make a few modifications to the original algorithm. In the original description [18], 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 for nearly all graphs. We avoid unnecessary computation by distinguishing between active and inactive nodes. It is unnecessary to recompute the label weights for a node whose neighborhood has 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 1 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 [17] as it does not lead to convergence on some inputs. In the original description [18], the criterion is to stop when all nodes have the label of the relative majority in their neighborhood.

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 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 [18]. When dealing with scale-free networks whose degree distribution follows a power law, assigning node ranges of equal size to each thread can lead to load imbalance as computational cost depends on the node degree. Instead of statically dividing the iteration among the threads, guided scheduling (with parallel for schedule(guided)) can help to overcome load balancing issues, although this introduces some overhead. We observed that dynamic scheduling is generally superior to static parallelization in terms of PLP’s speed.

### B. Parallel Louvain Method (PLM)

**Algorithm:** The Louvain method for community detection was first presented by Blondel et al. [3]. 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 so 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 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 $\text{vol}(u) := \sum_{\{u,v\} : v \in N(u)} \omega(u, v) + 2 \cdot \omega(u, u)$ and $\text{vol}(C) := \sum_{u \in C} \text{vol}(u)$, respectively. The modularity of a solution is defined as

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

(IV.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:

![Figure 1: Number of active and updated labels per iteration of PLP for the web graph uk-2002.](image-url)
\[
\Delta_{\text{mod}}(u, C \rightarrow D) = \frac{\omega(u, D \setminus \{u\}) - \omega(u, C \setminus \{u\})}{\omega(E)}
+ \frac{(\text{vol}(C \setminus \{u\}) - \text{vol}(D \setminus \{u\})) \cdot \text{vol}(u)}{2 \cdot \omega(E)^2}
\]

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 \(\Delta_{\text{mod}}\) 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.

**Algorithm 2:** move: Local node moves for modularity gain

**Input:** graph \(G = (V, E)\), communities \(\zeta : V \rightarrow \mathbb{N}\)

**Result:** communities \(\zeta : V \rightarrow \mathbb{N}\)

1. \(\zeta \leftarrow \text{singleton}(G)\)
2. repeat
3. parallel for \(u \in V\)
4. \(\delta \leftarrow \max_{v \in N(u)} \{\Delta_{\text{mod}}(v, \zeta(u) \rightarrow \zeta(v))\}\)
5. \(C \leftarrow \zeta(\arg\max_{v \in N(u)} \{\Delta_{\text{mod}}(v, \zeta(u) \rightarrow \zeta(v))\})\)
6. if \(\delta > 0\) then
7. \(\zeta(u) \leftarrow C\)
8. until \(\zeta\) stable
9. return \(\zeta\)

**Algorithm 3:** PLM: Parallel Louvain Method

**Input:** graph \(G = (V, E)\)

**Result:** communities \(\zeta : V \rightarrow \mathbb{N}\)

1. \(\zeta \leftarrow \text{singleton}(G)\)
2. move\((G, \zeta)\)
3. if \(\zeta\) changed then
4. \(G' \leftarrow \text{coarsen}(G, \zeta)\)
5. \(\zeta' \leftarrow \text{PLMR}(G')\)
6. \(\zeta \leftarrow \text{prolong}(\zeta', G)\)
7. return \(\zeta\)

**Implementation:** Our implementation of PLM (Algorithm 3) employs parallel iteration over the node set. Since the computation of the \(\Delta_{\text{mod}}\) 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 \texttt{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 by a factor of 2 or more. The current implementation only stores and updates the volume of each community, and requires locking for this purpose. Two strategies are available for apportioning the node set among the threads, simple (static scheduling) and balanced (guided scheduling), whose effects are discussed in Section VI-B.

**C. Parallel Louvain Method with Refinement (PLMR)**

Following up on the work by Noack and Rotta on multilevel techniques and refinement heuristics [21], 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 which uses the same concepts as PLM.

**Algorithm 4:** PLMR: Parallel Louvain Method with Refinement

**Input:** graph \(G = (V, E)\)

**Result:** communities \(\zeta : V \rightarrow \mathbb{N}\)

1. \(\zeta \leftarrow \text{singleton}(G)\)
2. move\((\zeta, G)\)
3. if \(\zeta\) changed then
4. \(G' \leftarrow \text{coarsen}(G, \zeta)\)
5. \(\zeta' \leftarrow \text{PLMR}(G')\)
6. \(\zeta \leftarrow \text{prolong}(\zeta', G)\)
7. return \(\zeta\)

**D. Ensemble Techniques (EPP and EML)**

**Algorithms:** In machine learning, ensemble learning is a strategy in which multiple base classifiers or weak classifiers are combined to form a strong classifier. It has been shown that combining results of classifiers which are only slightly better than random guessing yields qualitatively good results. 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 [17]. Subsequently, we describe two ensemble techniques EPP and EML of which EPP emerges as an efficient method. We also briefly describe algorithms for combining multiple base solutions.

1. **Ensemble Preprocessing (EPP):** When aiming for a good tradeoff between speed and quality, the following approach emerges as the most promising one: In a preprocessing
Algorithm 5: EPP: Ensemble Preprocessing

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

1. parallel for \( \text{Base} \in B \)
2. \( \zeta \leftarrow \text{Base}(G) \)
3. \( \bar{\zeta} \leftarrow \text{combine}(\zeta_1, \ldots, \zeta_b) \)
4. \( G' \leftarrow \text{coarsen}(G, \bar{\zeta}) \)
5. \( \zeta' \leftarrow \text{Final}(G') \)
6. \( \zeta \leftarrow \text{prolong}(\zeta', G) \)
7. return \( \bar{\zeta} \)

step, assign \( G \) to an ensemble of base algorithms. The graph
is then coarsened according to the core communities \( \bar{\zeta} \), which
represents 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. We instantiate this scheme with PLP as a base
algorithm and PLM or 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 PLM or 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.

2) Ensemble Multilevel (EML): A natural way to extend
the ensemble preprocessing method is to apply it recursively:
After the core communities have been computed, the original
graph \( G \) is coarsened to a smaller graph \( G' \) according to the
communities. Then the algorithm is called recursively on \( G' \),
again assigning the coarsened graph to an ensemble. Several
options for stopping the recursion are possible: (i) if the
communities remain singletons and thus \( G = G' \), (ii) if the
coarsest graph is smaller than a threshold, (iii) if the change in
modularity from one recursion level to the next is non-positive,
or (iv) if the quality of the solution has not improved for a
number of levels. Clearly, option (i) requires the program to
be stopped to prevent an infinite loop. The other options are
not strictly necessary, but save running time in case further
quality improvements are unlikely.

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

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

We introduce a 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 which
appears sufficient for our purposes.

If base solutions with connected communities need to be
converted to core communities with all communities
connected, an alternative approach may be necessary. We suggest
a method inspired by region growing. Starting with singletons
\( \bar{\zeta} \), every edge \( \{u, v\} \) is traversed in a breadth-first search of
the graph and nodes are assigned according to the rule

\[
\forall \zeta_i : \zeta_i(u) = \zeta_i(v) \implies \bar{\zeta}(v) \leftarrow \bar{\zeta}(u).
\]

However, this approach is relatively slow, does not easily
support parallelism, and may not yield communities according
to Eq. (IV.2), so we discarded it early. The use of a \( b \)-way hash
function is advantageous for creating the core communities,
as it is significantly faster than region growing due to a high
degree of parallelism.

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.

Our implementations of the ensemble techniques EML and
EPP are agnostic to the base and final algorithms and can be
instantiated with a variety of such algorithms.

V. IMPLEMENTATION AND EXPERIMENTAL SETUP

A. Framework

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 custom adjacency array graph data structure.
A high-level interface encapsulates the data structure and
enables a clean and concise notation of graph algorithms. In
particular, our interface conveniently supports parallel pro-
gramming through parallel node and edge iteration methods
which receive a function (generally a closure) and apply it to
all elements in parallel. Parallelism is achieved in the form
of loop parallelization with OpenMP, using the parallel
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, our growing
toolkit for network analysis. The software combines high-
performance algorithms written in C++ with an interactive
Python interface for flexible data analysis workflows.

\(^3\text{hash functions: }\text{http://www.cse.yorku.ca/~oz/hash.html}\)

\(^4\text{NetworKit: }\text{http://parco.iti.kit.edu/software/}\)
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, among them web graphs, internet topology networks, social networks, scientific coauthorship networks and street networks (see Table II in the Appendix). 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. With a set of 13+1 networks from different domains, we aim for generalizable results. Note that the achievable modularity for a network depends on the 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 and are freely available on the web.\(^5\) They are undirected, unweighted graphs stored in files of the METIS adjacency format. Two additional large complex networks, as-Skitter and soc-LiveJournal, are derived from the Stanford Large Network Dataset Collection\(^6\). Table I in the appendix lists the set of test graphs and gives a short characterization of each. We evaluate solution quality and running time for all of our own algorithms as well as several relevant competitors on this set. For foreign implementations we rely on the modularity calculations performed by them after we validated the correctness on a sample of graphs. For those algorithms which proved scalable, we also add experiments the largest graph available to us, a web graph of the .uk domain with \(m \approx 3.3 \cdot 10^9\) (see Section VI-F).

To measure strong scaling, we run our basic parallel algorithms on the uk-2002 web graph and increase the number of threads from 1 to 32. 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. For weak scaling experiments (Figure 11 and 11 in the appendix), we use a series of synthetic Kronecker graphs where each graph has twice the size of its predecessor (from \(\log n = 16 \ldots 22\)), and double the number of threads simultaneously from 1 to 32.


Figure 2: Size comparison of all graphs used in experiments

C. Settings

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.

VI. 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 3 and 4 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 discuss the results in the following sections. Section VI-F presents the results in a condensed form.

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.

Table I: Platform for experiments

<table>
<thead>
<tr>
<th>compiler</th>
<th>gcc 4.7.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2 x 8 Cores: Intel(R) Xeon(R) E5-2680 0 @ 2.70GHz, 32 threads</td>
</tr>
<tr>
<td>RAM</td>
<td>256 GB</td>
</tr>
<tr>
<td>OS</td>
<td>SUSE 12.2-64</td>
</tr>
</tbody>
</table>

Figure 3: PLP strong scaling on the uk-2002 web graph

PLP exhibits good scaling behavior, apart from some overhead introduced by OpenMP parallelization. To demonstrate strong scaling behavior, we apply PLP to the large uk-2002 web graph and increase the number of threads from 1 to 32 (Figure 3). A speedup of about factor 7 is achieved when scaling from 1 to 32 threads. We conjecture that the exception when transitioning from 1 to 2 threads is caused not only by OpenMP overhead, but also by the Intel CPU technology of our platform where processor clock frequency is adapted to the number of cores currently utilized. Our results indicate that PLP can benefit strongly from increased parallelism.
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. Comparing simple and balanced OpenMP parallelization, we observe that quality and speed vary depending on the graph. Using guided scheduling may result in both speedup and slowdown, as well as loss or gain of modularity. This results from the order in which nodes are visited, which can have an influence on quality. Therefore, we confirm that the choice between simple and balanced parallelization should remain a configurable parameter of the implementation. In the following, we use the balanced variant as default. Even large instances are processed in no more than a few minutes. In comparison to PLP (Figure 6b), we observe that PLP can solve instances in only 10-20 percent of the time required by PLM, but at a significant loss of modularity. As discussed in Sec. VII, the communities detected by the two algorithms are markedly different. Because the Louvain method for community detection is well-known and accepted, we chose the performance of PLM as our baseline (Figure 6a) and present quality and running time of other algorithms relative to PLM. PLM detects communities with relatively high modularity in the majority of networks.

![Figure 4: PLM strong scaling on the uk-2002 web graph](image)

Figure 4 shows the scaling behavior of PLM. (Weak scaling results on PLP and PLM are shown in Figure 11 in the appendix.) The low speedup values are a consequence of Amdahl’s law: While the move phases of the algorithm are highly efficient and are accelerated by increasing parallelism (very similar to PLP), the overall running time is dominated by the coarsening phases, which are sequential in our current implementation. Future improvements on the implementation will therefore focus on introducing parallel coarsening, which requires a concurrent graph data structure.

C. Parallel Louvain Method with Refinement (PLMR)

As shown by Figure 6c, 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.

D. Ensemble Techniques

1) Ensemble Multilevel (EML): Preliminary experiments with EML led us to discard the approach early, because the iterated scheme does not pay off in terms of quality in most cases. Usually the quality obtained on the first or second level cannot be further improved. This contrasts results by Ovelgönne and Geyer-Schulz [17]. One reason for this discrepancy may well be small differences in our implementation of label propagation. Furthermore, it seems that PLP is not the ideal sole base algorithm on coarser levels. There its solutions become quite similar. Similar base solutions limit the optimization process. EPP, discussed next, is therefore faster and even improves quality compared to EML.

2) Ensemble Preprocessing (EPP): Figure 5 demonstrates the efficiency of the simpler ensemble approach. Results were generated by an EPP instance with a 4-piece PLP ensemble and PLM 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 6d), the ensemble approach can be several times faster on large networks, but quality is slightly worse in most cases. Using the qualitatively stronger PLMR as the final algorithm seems to have a negligible effect on quality and running time (Figure 6e). This indicates that refinement on the already strongly coarsened graph is less effective.

The effect of ensemble size remains to be explored. In preliminary experiments, we vary the ensemble size, doubling the number of base algorithms from 1 to 8, and observe the difference in modularity. On average, quality can be gained by increasing the ensemble size, although the actual difference depends strongly on the individual graph. On a few small graphs, a bigger ensemble makes the difference between failure to distinguish communities and a solution with reasonable quality, while on a large graph like uk-2002, the difference is near zero. Running time increases at least proportionally to the number of base algorithms. We therefore conclude that forming a large ensemble is not justified, since a small ensemble already provides quality improvement, and settle on four base algorithms as the default choice.

Combining the results of multiple classifiers in an ensemble learning scenario is only meaningful if the base classifiers disagree about some elements. Therefore, we inspected the diversity of the base solutions by calculating the Jaccard dissimilarity measure [12], observing that the communities produced by multiple PLP instances are not necessarily different, that the dissimilarity varies non-deterministically between multiple runs, and that quality produced by EPP depends on the degree of dissimilarity to some extent. As a possible solution for creating diversity among the solutions, explicitly
randomizing the order in which PLP traverses the nodes becomes interesting again in an ensemble setting. However, we find that explicit randomization has no significant effect on quality in an ensemble setting, while it slows down the algorithm for large graphs. We therefore confirm that explicit randomization should be omitted. As an alternative solution, we try to perturb the communities initially by randomly choosing a small number of seed nodes and deactivating them, or activating only this seed set. However, deactivation of seed nodes does not seem to influence solution diversity or result quality in a reproducible way.

We conclude that the ensemble technique EPP represents a good speed-quality tradeoff when very large networks are processed. While somewhat lower in modularity, the communities detected are similar (see Sec. VII) to those of the Louvain method, without requiring as much computation time and memory.

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 \(^7\) and RG, as well as CGGC and CGGCI \(^9\), ensemble algorithms based on RG. We also include the widely used original sequential Louvain\(^8\) implementation. 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.

---

\(^7\)CLU_TBB \(\text{http://www.staff.science.uu.nl/~faggi101/}\)
\(^8\)RG etc: \(\text{http://www.umiacs.umd.edu/~mov/}\)
\(^9\)Louvain \(\text{https://sites.google.com/site/findcommunities/}\)
Figure 6: Performance of our algorithms in comparison: PLM serves as the baseline.
Figure 7: Performance of competitors relative to baseline PLM
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 13 networks with the running time of PLM as the baseline, while the modularity score is the arithmetic mean of absolute modularity differences. Figure 8 shows the resulting points. It becomes clear that most algorithms are in fact Pareto optimal, and all of them are placed close to the Pareto frontier. PLP is unrivaled in terms of time to solution, but solution quality is suboptimal. The scores for EPP(4,PLP,PLM) and EPP(4,PLP,PLMR) are so close that we show them as a single point. 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 emerges as a qualitatively strong and fast candidate. It is also evident that our extended version PLMR can improve solution quality for a small computational extra charge. We recommend both as the default algorithms for parallel community detection in large networks. The original sequential implementation of the Louvain method is still a viable choice, but ultimately cannot benefit from multicore systems. It should be noted that PLM, PLMR and EPP in their current implementation are still held back by sequential parts, and can probably be further improved by introducing parallel coarsening on a concurrent graph data structure. 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.

G. One More Massive Network

In addition to the experiments presented so far, we run our parallel algorithms on the web graph uk-2007-05, at about 3.3 billion edges the largest data set currently available to us. CLU_TBB would probably be capable of solving this instance, but the current implementation failed at reading the input file. This leaves us with five of our own parallel algorithms, which generate interesting experimental results (Figure 9): On a network of this size, the speed difference between the ensemble approach and the parallel Louvain variants becomes significant. EPP(4,PLP,PLM) takes under 8 minutes, while PLM requires about 36 minutes. The modularity difference between them is near zero. Although overall worse in the Pareto evaluation, EPP emerges as the strongest algorithm for this particular network. This indicates that the ensemble approach may play out its strengths on even larger real-world networks than those readily available to our research community. As expected, PLP is by far the fastest algorithm and terminates in only 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.

VII. Qualitative Aspects

In this work we concentrate on achieving a good tradeoff between high modularity, a widely accepted quality measure for community detection, and low running time. Ideally we 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. Unlike the related task of graph partitioning, where finding a good partition often directly translates into time savings for a parallel application, we cannot make a cost/benefit analysis in the same way for community detection. 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 therefore goes beyond the scope of this paper.
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 10 shows community graphs for the power graph, a network representing the topology of the USA Western States Power Grid with 4941 nodes and 6594 edges [26]. From top to bottom, the solutions were produced by PLP, PLM, PLMR and EPP(4, PLP, PLM). It is apparent that PLP has a much finer resolution and detects 363 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, PLM) have a very similar resolution and divide the network into ca. 40 communities. While power is admittedly a very small graph, this example shows how community detection can help to reduce the complexity of networks for visual representation.

VIII. 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 Pareto-optimal with respect to running time and modularity in an experimental comparison with other state-of-the-art 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. Our modification PLMR of this method adds a refinement phase which enhances modularity for a small increase in running time. Additionally, the combination of these basic classifiers in an ensemble scheme yields the EPP algorithm. While it did not perform optimally on most of our test data set, it turned out to be the best algorithm on a very large network with 3.3 billion edges, processing it in under 8 minutes using 32 threads. Moreover, we have introduced an algorithmic framework whose extensibility and flexibility allow a seamless addition of further high-performance network analysis methods. The framework has been published as the open-source package NetworKit, and we invite other researchers to contribute to this effort.

We have exhausted the DIMACS Challenge and Stanford Large Network Dataset graph collections in terms of graph size without reaching the limits of our algorithms and available hardware in terms of running time. In practice, the memory footprint of large graphs is a limiting factor, but the compactness of our graph data structure can be improved. We plan to extend NetworKit with a concurrent graph data structure, which will allow parallel coarsening and likely further improve the scalability of all multilevel algorithms. Our ongoing work adapts the methods of this paper to a dynamic scenario in which communities are maintained while the graph changes.
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.

REFERENCES


APPENDIX

A. Network Statistics

Table I gives an overview over graph sizes as well as some structural features: A high maximum node degree (max.deg.) 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 a good indicator for the presence of distinct communities.

Table II: Overview of graphs used in experiments

<table>
<thead>
<tr>
<th>network</th>
<th>n</th>
<th>m</th>
<th>max.deg.</th>
<th>comp.</th>
<th>LCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>power</td>
<td>4941</td>
<td>6594</td>
<td>19</td>
<td>1</td>
<td>0.080</td>
</tr>
<tr>
<td>PGgiantcompo</td>
<td>10680</td>
<td>24316</td>
<td>205</td>
<td>1</td>
<td>0.266</td>
</tr>
<tr>
<td>P1plpy2008</td>
<td>22967</td>
<td>48436</td>
<td>25920</td>
<td>1</td>
<td>0.230</td>
</tr>
<tr>
<td>G_n_6_2</td>
<td>1000000</td>
<td>501198</td>
<td>25</td>
<td>6</td>
<td>0.604</td>
</tr>
<tr>
<td>caidaRouterLevel</td>
<td>192244</td>
<td>669066</td>
<td>1071</td>
<td>308</td>
<td>0.157</td>
</tr>
<tr>
<td>caidAuthorsCiteseer</td>
<td>227320</td>
<td>814134</td>
<td>1372</td>
<td>1</td>
<td>0.676</td>
</tr>
<tr>
<td>as-Matters</td>
<td>1958412</td>
<td>11097298</td>
<td>354555</td>
<td>756</td>
<td>0.258</td>
</tr>
<tr>
<td>caidFederDBLP</td>
<td>540486</td>
<td>15245729</td>
<td>3295</td>
<td>1</td>
<td>0.801</td>
</tr>
<tr>
<td>eu-2005</td>
<td>862664</td>
<td>16138468</td>
<td>68963</td>
<td>1</td>
<td>0.608</td>
</tr>
<tr>
<td>soc-LiveJournal</td>
<td>4847571</td>
<td>43110428</td>
<td>20334</td>
<td>1876</td>
<td>0.338</td>
</tr>
<tr>
<td>europe-oum</td>
<td>50912018</td>
<td>50454860</td>
<td>13</td>
<td>1</td>
<td>0.001</td>
</tr>
<tr>
<td>uk-2002</td>
<td>18520486</td>
<td>261787258</td>
<td>194955</td>
<td>38359</td>
<td>0.601</td>
</tr>
</tbody>
</table>

B. Weak Scaling Experiments

Figure 11 shows the results of weak scaling experiments for PLP and PLM. The overhead due to parallelism is apparent when moving from 1 to 2 threads. Moreover, it must be noted that perfect scaling cannot be expected due to the complex structure of the input. The results of the last column have been obtained with hyperthreading. Recall that the overall running time of PLM is dominated by the coarsening phases, which are sequential in our current implementation.

Figure 11: PLP (left) and PLM (right) weak scaling on the series of Kronecker graphs