Parallel Graph Partitioning for Complex Networks

Henning Meyerhenke, Peter Sanders, and Christian Schulz

Abstract—Processing large complex networks like social networks or web graphs has attracted considerable interest. To do this in parallel, we need to partition them into pieces of roughly equal size. Unfortunately, previous parallel graph partitioners originally developed for more regular mesh-like networks do not work well for complex networks. Here we address this problem by parallelizing and adapting the *label propagation* technique originally developed for graph clustering. By introducing size constraints, label propagation becomes applicable for both the coarsening and the refinement phase of multilevel graph partitioning. This way we exploit the hierarchical cluster structure present in many complex networks. We obtain very high quality by applying a highly parallel evolutionary algorithm to the coarsest graph. The resulting system is both more scalable and achieves higher quality than state-of-the-art systems like ParMetis or PT-Scotch. For large complex networks the performance differences are very big. As an example, our algorithm partitions a web graph with 3.3 G edges in 16 *seconds* using 512 cores of a high-performance cluster while producing a high quality partition—none of the competing systems can handle this graph on our system.

Index Terms— Load balancing and task assignment, graph algorithms, combinatorial algorithms, algorithm design and analysis

1 Introduction

Graph partitioning (GP) is a key prerequisite for efficient large-scale parallel graph algorithms. A prominent example is the PageRank algorithm, which is one of the measures used by web search engines to rank web pages displayed to the user. As huge networks become abundant, there is a need for their parallel analysis, requiring a sensible distribution of the graphs to the PEs (processing elements). In many cases, this means to partition a graph into k blocks of roughly equal size such that the communication between PEs in the underlying application is minimized. The latter is often estimated by the number of edges between the blocks (pieces). In this paper we focus on a version of the problem that constrains the maximum block size to $(1+\epsilon)$ times the average block size where $\epsilon \geq 0$ is an imbalance parameter. Our objective is to minimize the total cut size, i.e., the number of edges that run between blocks.

It is well-known that there are more realistic (and more complicated) objective functions involving also the block that is worst and the number of its neighboring nodes [2], but minimizing the cut size has been adopted as a kind of standard. The graph partitioning problem is NP-complete [3] and there is no approximation algorithm with a constant ratio factor for general graphs [4]. Thus, heuristic algorithms are used in practice.

A successful heuristic for partitioning large graphs is the *multilevel graph partitioning* (MGP) approach depicted in Fig. 1, where the graph is recursively *contracted* to achieve smaller graphs which should reflect the same basic structure as the input graph. After applying an *initial partitioning* algorithm to the smallest graph, the contraction is undone and, at each level, a *local search* method is used to improve the partitioning induced by the coarser level.

The main contributions of this paper are a scalable parallelization of the size-constrained label propagation algorithm and an integration into a multilevel framework that enables us to partition large complex networks. The parallel size-constrained label propagation algorithm is used to compute a graph clustering. A clustering of this kind is recursively contracted and recomputed on the coarser graph until the coarsest graph is small enough. The coarsest graph is then partitioned by the coarse-grained distributed evolutionary algorithm KaFFPaE [5]. During uncoarsening the size-constraint label propagation algorithm is used as a simple, yet effective, parallel local search algorithm.

The presented scheme speeds up the running time needed for partitioning and improves solution quality on graphs that have a very irregular and often also hierarchically clustered structure such as social networks or web graphs. On these graphs the *strengths* of our new algorithm unfold in particular: average solution quality *and* running time is much better than what is observed by using ParMetis [6]. A variant of our algorithm is able to compute a partition of a web graph with billions of edges in only a few seconds while producing much better solutions.

We organize the paper as follows. We begin in Section 2 by introducing basic concepts and outlining related work. Section 3 reviews the recently proposed cluster contraction algorithm [7] to partition complex networks, which is parallelized in this work. The main part of the paper is Section 4, which covers the parallelization of the size-constrained label propagation algorithm, the parallel contraction and uncontraction algorithm, as well as the overall parallel system. A summary of extensive experiments to evaluate the

- *H. Meyerhenke and P. Sanders are with the Karlsruhe Institute of Technology (KIT), Karlsruhe 76131, Germany. E-mail: {meyerhenke, sanders}@kit.edu.*
- *C. Schulz is with the Karlsruhe Institute of Technology (KIT), Karlsruhe 76131, Germany, and the University of Vienna, Vienna 1010, Austria. E-mail: christian.schulz@kit.edu.*

Manuscript received 26 July 2016; revised 3 Feb. 2017; accepted 14 Feb. 2017. Date of publication 20 Feb. 2017; date of current version 9 Aug. 2017.
Recommended for acceptance by U.V. Catalyurek.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2671868

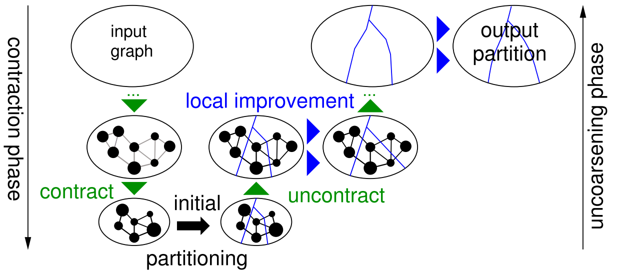


Fig. 1. Multilevel graph partitioning. The graph is recursively contracted to achieve smaller graphs. After the coarsest graph is initially partitioned, a local search method is used on each level to improve the partitioning induced by the coarser level.

algorithm's performance is presented in Section 5. Finally, we conclude in Section 6.

2 Preliminaries

2.1 Basic Concepts

Let $G = (V = \{0, \ldots, n-1\}, E, c, \omega)$ be an undirected graph with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. $N(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v. A node $v \in V_i$ that has a neighbor $w \in V_j, i \neq j$, is a *boundary node*. We are looking for *blocks* of nodes $V_1, \ldots, V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in \{1..k\} : c(V_i) \leq L_{\max} := (1 + \epsilon) \lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter ϵ. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. We call a block V_i *underloaded* [*overloaded*] if $c(V_i) < L_{\max}$ [if $c(V_i) > L_{\max}$].

A *clustering* is also a partition of the nodes. However, k is usually not given in advance and the balance constraint is removed. A size-constrained clustering constrains the size of the blocks of a clustering by a given upper bound U such that $c(V_i) \leq U$. Note that by adjusting the upper bound one can somewhat control the number of blocks of a feasible clustering. For example, when using $U = 1$, the only feasible size-constrained clustering in an unweighted graph is the singleton clustering, where each node forms a block on its own.

An abstract view of the partitioned graph is the so-called *quotient graph*, in which nodes represent blocks and edges are induced by connectivity between blocks. The *weighted* version of the quotient graph has node weights which are set to the weight of the corresponding block and edge weights which are equal to the total weight of the edges that run between the respective blocks. By default, our initial inputs will have unit edge and node weights. However, even those will be translated into weighted problems in the course of the multilevel algorithm. In order to avoid tedious notation, G will denote the current state of the graph before and after a (un)contraction in the multilevel scheme throughout this paper.

2.2 Related Work

Graph partitioning is a thoroughly studied research problem, we refer the reader to [8], [9], [10] for a broad overview. Here, we focus on issues closely related to our main contributions. Most, if not all, general-purpose methods that are able to obtain good partitions for large real-world graphs in reasonable time are based on the multilevel principle. The basic idea can be traced back to multigrid solvers for solving systems of linear equations [11] but more recent practical methods are based on mostly graph theoretic aspects, in particular edge contraction and local search. There are many ways to create graph hierarchies such as matching-based schemes [12], [13], [14] or variations thereof [15] and techniques similar to algebraic multigrid, e.g., [16]. We refer the interested reader to the respective papers for more details. Well-known software packages based on this approach include Jostle [12], Metis [13] and Scotch [17].

Most probably the fastest available parallel code is the parallel version of Metis, ParMetis [6]. This parallelization has problems maintaining the balance of the partitions since at any particular time, it is difficult to say how many nodes are assigned to a particular block. PT-Scotch [17], the parallel version of Scotch, is based on recursive bipartitioning. This approach is more difficult to parallelize efficiently compared to k-partitioning since in the initial bipartition, there is less parallelism available. The unused processor power is used by performing several independent attempts in parallel. The involved communication effort is reduced by considering only nodes close to the boundary of the current partitioning (band-refinement). KaPPa [18] is a parallel matching-based MGP algorithm which is also restricted to the case where the number of blocks equals the number of processors used. PDibaP [19] is a multilevel diffusion-based algorithm that is targeted at small-to medium-scale parallelism with dozens of processors.

As reported by [20], most large-scale graph processing toolkits based on cloud computing use ParMetis or rather straightforward partitioning strategies such as hash-based partitioning. While hashing often leads to acceptable balance, the edge cut obtained for complex networks is very high. To address this problem, Tian et al. [20] have recently proposed a partitioning algorithm for their toolkit Giraph++. The algorithm uses matching-based coarsening and ParMetis on the coarsest graph. This strategy leads to better cut values than hashing-based schemes. Yet, it introduces significant imbalance, so that their results are incomparable to ours.

Recent work by Kirmani and Raghavan [21] solves a relaxed version of the graph partitioning problem where no strict balance constraint is enforced. The blocks only have to have approximately the same size so that the results are incomparable. Note that the problem is easier than fulfilling a strict balance constraint. Their approach attempts to obtain information on the graph structure by computing an embedding into the coordinate space with multilevel force-directed graph drawing. Afterwards partitions are computed using a geometric scheme. Since force-directed graph drawing usually results in "hairballs" for complex networks [22], this approach does not seem very promising in our context.

The label propagation clustering algorithm was initially proposed by Raghavan et al. [23]. A single round of simple label propagation can be interpreted as the randomized agglomerative clustering approach proposed by Catalyurek and Aykanat [24]. Moreover, the label propagation algorithm has been used to partition networks by Uganer and Backstrom [25]. The authors do not use a multilevel scheme and rely on a given or random partition which is improved by combining the unconstrained label propagation approach with linear programming. The approach does not yield high quality partitions.

Recently, Slota et al. [26] have used label propagation for partitioning complex networks as well. Their algorithm, termed PuLP by its authors, differs in one very important aspect: It does not use the multilevel approach. Thus, as we will see in our experimental comparison, the partitioning quality suffers. Also, it may be difficult to adhere to a tight node balance constraint such as the typical 3 percent with PuLP. On the other hand, PuLP can balance according to edges as well and consider multiple optimization objectives, properties our algorithm does not have. As Slota et al. also point out, state-of-the-art distributed-memory partitioning tools "adopt a graph distribution scheme, a specific partitioning method, and then organize inter-node communication around these choices." Here distribution refers to the way the graph's sparse adjacency matrix is distributed over the processors in the parallel partitioner. Previous work [27] has indicated that for complex networks a 2D distribution (computed on the basis of 1D graph or hypergraph partitioning) can be advantageous. As Slota et al. and other established parallel partitioning tools (such as ParMetis or PT-Scotch), we focus in this paper on the partitioning aspect and use internally a 1D distribution with natural ordering.

Wang et al. [28] present a multi-level label propagation algorithm. Their approach computes results with solution quality being comparable to Metis. However, there is no strict balance constraint on the blocks so that the results reported in [28] are incomparable to ours.

2.3 KaHIP

Within this work, we use the open source multilevel graph partitioning framework KaHIP [29] (Karlsruhe High Quality Partitioning). KaHIP implements many different algorithms, for example flow-based methods and more-localized local searches within a multilevel framework called KaFFPa, as well as several coarse-grained parallel and sequential meta-heuristics. Recently, also specialized methods to partition social networks and web graphs have been included into the framework [7]. In the present paper, we parallelize the main techniques presented therein; they are reviewed in Section 3.

2.3.1 KaFFPaE

We use the evolutionary algorithm KaFFPaE [5] to obtain a high-quality partition of the coarsest graph in the hierarchy. KaFFPaE [5] is a coarse-grained evolutionary algorithm, i.e., each PE has its own population (set of partitions) and a copy of the graph. After initially creating the local population, each processor performs combine and mutation operations on the local population/partitions. The algorithm contains a general combine operator framework provided by modifications of the multilevel framework KaFFPa. Roughly speaking, a combine operation in the framework is a modified multilevel V-cycle that will not contract edges that are cut in one of the input partitions. This ensures that the output individual is at least as good as the better of both input individuals. This is combined with a scalable communication protocol similar to randomized rumor spreading to mix the local populations. For more details, we refer the reader to [5].

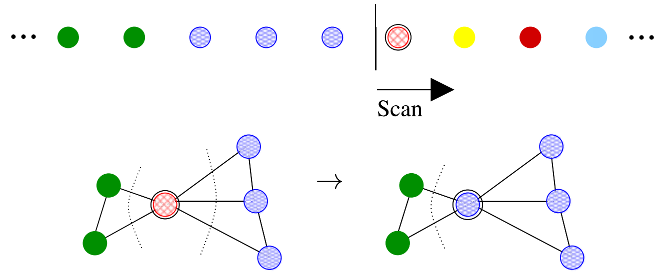


Fig. 2. An example round of the label propagation graph clustering algorithm. Initially each node is in its own block. The algorithm scans all vertices in a random order and moves a node to the block with the strongest connection in its neighborhood.

3 Size-Constrained Label Propagation

We now review the basic idea for contraction and local search [7] which we chose to parallelize. The approach is targeted at complex networks such as social networks and web graphs. Such networks often have a pronounced and hierarchical cluster structure. Also, they often contain star-like structures. A matching-based algorithm for coarsening matches only a single edge in these star-like structures and hence cannot shrink the graph effectively. Moreover, it may contract "wrong" edges such as bridges.

In our approach, the size-constrained label propagation algorithm is used to compute a clustering of the graph. To compute a graph hierarchy, the clustering is contracted by replacing each cluster by a single node, and the process is repeated recursively until the graph is small. This makes the contraction of edges in small cuts unlikely since the cluster structure of complex networks is detected by the algorithm.

Note that cluster contraction is an aggressive coarsening strategy. In contrast to most previous approaches, it can drastically shrink the size of irregular networks. Regarding complexity, experiments in [7] indicate that already one contraction step can shrink the graph size by orders of magnitude and that the average degree of the contracted graph is smaller than the average degree of the input network.

3.1 Label Propagation with Size Constraints

Originally, the *label propagation clustering* algorithm was proposed by Raghavan et al. [23] for graph clustering. It is a very fast, near linear-time algorithm that locally optimizes the number of edges cut. Initially, each node is in its own cluster/block, i.e., the initial block ID of a node is set to its node ID. The algorithm then works in rounds. In each round, the nodes of the graph are traversed in a random order. When a node v is visited, it is *moved* to the block that has the strongest connection to v, i.e., it is moved to the cluster V_i that maximizes $\omega(\{(v,u) \mid u \in N(v) \cap V_i\})$. Ties are broken randomly. Originally, the process is repeated until the process has converged. We perform at most ℓ iterations of the algorithm instead, where ℓ is a tuning parameter. One round of the algorithm can be implemented to run in $\mathcal{O}(n+m)$ time. An example is shown in Fig. 2.

The computed clustering is contracted to obtain a coarser graph. *Contracting a clustering* works as follows: each block of the clustering is contracted into a single node. The weight of the node is set to the sum of the weight of all nodes in the original block. There is an edge between two nodes u and v

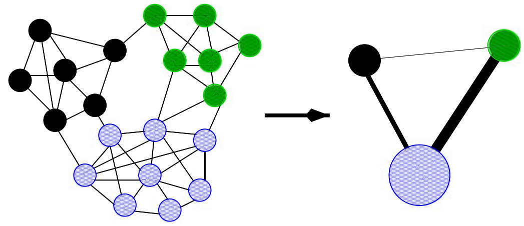


Fig. 3. Contraction of clusterings. Each cluster of the graph on the left hand side corresponds to a node in the graph on the right hand side. Weights of the nodes and the edges are choosen such that a partition of the coarse graph induces a partition of the fine graph having the same cut and balance.

in the contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G, i.e., block u and block v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weight of edges that run between block A and block B of the clustering. Due to the way contraction is defined, a partition of the coarse graph corresponds to a partition of the finer graph with the same cut and balance. An example contraction is shown in Fig. 3.

In our previous work [7] we adapted the original label propagation algorithm [23] in order to ensure that each block of the cluster fulfills a size constraint. There are two reasons for this. First, consider a cluster of the graph in which the weight of the block would exceed $L_{\max}$. After contracting this clustering, it would be impossible to find a partition of the contracted graph that fulfills the balance constraint. Second, it has been shown that using graph hierarchies with node weights that are more balanced is usually beneficial when computing high quality graph partitions [18]. To ensure that blocks of the clustering do not become too large, an upper bound $U := \max(\max_v c(v), W)$ on the size of the blocks is introduced, where W is a parameter that will be chosen later. When the algorithm starts to compute a graph clustering on the input graph, the constraint is fulfilled since each of the blocks contains exactly one node. A neighboring block V_ℓ of a node v is called *eligible* if V_ℓ will not be overloaded once v is moved to V_ℓ. Now when a node v is visited, it is moved to the *eligible block* that has the strongest connection to v. Hence, after moving a node, the size of each block is still smaller than or equal to U. Moreover, after contracting the clustering, the weight of each node is smaller or equal to U. One round of the modified version of the algorithm can still run in linear time by using an array of size $|V|$ to store the block sizes. Note that, when parallelizing the algorithm, this is something that needs to be adjusted since storing an array of size $|V|$ on a single processor would cost too much memory. The parameter W is set to $\frac{L_{\max}}{f}$, where f is a tuning parameter. Note that the constraint is rather soft during coarsening, i.e., in practice it does no harm if a cluster contains slightly more nodes than the upper bound. We go into more detail in the next section.

The process of computing a size-constrained clustering and contracting it is repeated recursively. As soon as the graph is small enough, it is initially partitioned. That means each node of the coarsest graph is assigned to a block. Afterwards, the solution is transferred to the next finer level. To do this, a node of the finer graph is assigned to the block of its coarse representative. Local improvement methods of KaHIP then try to improve the solution on the current level, i.e., reducing the number of edges cut.

Recall that the label propagation algorithm traverses the nodes in a random order and moves a node to a cluster with the strongest connection in its neighborhood to compute a clustering. Our previous work [7] has shown that using the ordering induced by the node degree (increasing order) improves the overall solution quality *and* running time on average. Using this node ordering means that in the first round of the label propagation algorithm, nodes with small node degree can change their cluster before nodes with a large node degree.

By using a different size-constraint—the constraint $W := L_{\max}$ of the original partitioning problem—the label propagation is also used as a simple and fast local search algorithm to improve a solution on the current level [7]. Note that in this case the definition of the strongest connection is similar to the concept of gain that is usually used in Fiduccia-Mattheyses refinements [30], i.e., when looking at a node you move it to the block yielding the strongest reduction in cut size. However, small modifications to handle overloaded blocks have to be made. The block selection rule is modified when the algorithm is used as a local search algorithm in case that the current node v under consideration is from an overloaded block V_ℓ. In this case it is *moved* to the eligible block that has the strongest connection to v without considering the block V_ℓ that it is contained in. This way it is ensured that the move improves the balance of the partition (at the cost of the number of edges cut).

4 Parallelization

We now present the main contributions of the paper. We begin with the distributed memory parallelization of the size-constrained label propagation algorithm and continue with the parallel contraction and uncoarsening algorithm. At the end of this section, we describe the overall parallel system.

4.1 Parallel Label Propagation

We shortly outline our parallel graph data structure and the implementation of the methods that handle communication. First of all, each PE gets a subgraph, i.e., a contiguous range of nodes $a..b$, of the whole graph as its input, such that the subgraphs combined correspond to the input graph. Each subgraph consists of the nodes with IDs from the interval $I := a..b$ and the edges incident to the nodes of those blocks, as well as the end points of edges which are not in the interval I (so-called ghost or halo nodes). This implies that each PE may have edges that connect it to another PE and the number of edges assigned to the PEs might vary significantly. Conceptually, this corresponds to a 1D partition of the graph's (sparse) adjacency matrix. Actually, the subgraphs are stored using an adjacency array representation, a standard sparse matrix data structure. This means that we have one array to store edges and one array for nodes storing head pointers to the edge array. In any case, we only store edges for the local vertices. However, the node array is divided into two parts. The first part stores local nodes and the second part stores ghost nodes. The method used to keep local node IDs and ghost node IDs consistent is explained in the next paragraph.

Additionally, we store information about the nodes, i.e., its current block and its weight.

Instead of using the node IDs provided by the input graph (called global IDs), each PE maps those IDs to the range $0 \mathinner{\ldotp\ldotp} n_p - 1$, where n_p is the number of distinct nodes of the subgraph. Note that this number includes the number of ghost nodes the PE has. Each global ID $i \in a \mathinner{\ldotp\ldotp} b$ is mapped to a local node ID $i - a$. The IDs of the ghost nodes are mapped to the remaining $n_p - (b - a + 1)$ local IDs in the order in which they appeared during the construction of the graph structure. Transforming a local node ID to a global ID or vice versa, can be done by adding or subtracting a. We store the global ID of the ghost nodes in an extra array and use a hash table to transform global IDs of ghost nodes to their corresponding local IDs. Additionally, we store for each ghost node the ID of the corresponding PE, using an array for $\mathcal{O}(1)$ lookups.

To parallelize the label propagation algorithm, each PE performs the algorithm on its part of the graph. Recall, when we visit a node v, it is moved to the block that has the strongest eligible connection. Note that the cluster IDs of a node can be arbitrarily distributed in the range $0 \mathinner{\ldotp\ldotp} n - 1$ so that we use a hash map to identify the cluster with the strongest connection. Since we know that the number of distinct neighboring cluster IDs is bounded by the maximum degree in the graph, we use hashing with linear probing. At this particular point of the algorithm, hashing with linear probing is much faster than using the hash map of the STL.

During the course of the algorithm, local nodes can change their block and hence the blocks in which ghost nodes are contained can change as well. Since communication is expensive, we do not want to perform communication each time a node changes its block. We use the following scheme to *overlap* communication and computation. The scheme is organized in phases. We call a node *interface node* if it is adjacent to at least one ghost node. The PE associated with the ghost node is called adjacent PE. Each PE stores a separate send buffer for all adjacent PEs. During each phase, we store the block ID of interface nodes that have changed into the send buffer of each adjacent PE of this node. Communication is then implemented asynchronously using non-blocking operations, i.e., while messages are routed through the network, we compute new labels of the nodes of the next phase. In phase κ, we send the current updates to the adjacent PEs and receive the updates of the adjacent PEs from round $\kappa - 1$, for $\kappa > 1$. Note that in case the label propagation algorithm has converged, i.e., no node changes its block any more, the communication volume is really small.

The degree-based node ordering approach of the label propagation algorithm that is used during coarsening is parallelized by considering only the local nodes for this ordering. In other words, the ordering in which the nodes are traversed on a PE is determined by the node degrees of the local nodes of this PE. During uncoarsening random node ordering is used.

Note that due to the parallelization it is possible that oscillations occur, i.e., nodes can change their label without convergence. Overlapping computation and communication reduces this effect. On the other hand, we do not need an optimal clustering so that we tolerate oscillations, stop prematurely and still get good quality.

4.2 Balance/Size Constraint

Maintaining the balance of blocks is more difficult in the parallel case than in the sequential case. We use two different approaches to maintain balance, one for coarsening and the other one for uncoarsening. The reason for using two approaches is that during coarsening there is a large number of blocks and the constraint is rather soft ($\frac{L_{\max}}{f}$), whereas during uncoarsening the number of blocks is small and the constraint is tight ($L_{\max}$).

We maintain the balance of different blocks *during coarsening* as follows. Roughly speaking, a PE maintains and updates only the local amount of node weight of the blocks of its local and ghost nodes. Due to the way the label propagation algorithm is initialized, each PE knows the exact weights of the blocks of local nodes and ghost nodes in the beginning. Label propagation then uses the local information to bound the block weights. Once a node changes its block, the local block weight is updated. Note that this does not involve additional communication. We decided to use this localized approach since the balance constraint is not tight during coarsening. More precisely, the bound on the cluster sizes during coarsening is a tuning parameter and the overall performance of the system does not directly depend on the exact choice of the parameter.

During uncoarsening we use a different approach since the number of blocks is much smaller and it is unlikely that the previous approach yields a feasible partition in the end. This approach is similar to the approach that is used within ParMetis [6]. Initially, the exact block weights of all k blocks are computed locally. The local block weights are then aggregated and broadcast to all PEs. Both can be done using one allreduce operation. Now each PE knows the global block weights of all k blocks. The label propagation algorithm then uses this information and locally updates the weights. For each block, a PE maintains and updates the total amount of node weight that local nodes contribute to the block weights. Using this information, one can restore the exact block weights with one allreduce operation which is done at the end of each computation phase. This approach would not be feasible during coarsening as there are n blocks in the beginning of the algorithm and each PE holds the block weights of all blocks.

4.3 Parallel Contraction and Uncoarsening

The *parallel contraction* algorithm works as follows. After the parallel size-constrained label propagation algorithm has been performed, each node is assigned to a cluster. Recall the definition of our general contraction scheme. Each of the clusters of the graph corresponds to a coarse node in the coarse graph and the weight of this node is set to the total weight of the nodes that are in that cluster. Moreover, there is an edge between two coarse nodes iff there is an edge between the respective clusters and the weight of this edge is set to the total weight of the edges that run between these clusters in the original graph.

In the parallel scheme, the IDs of the clusters on a PE can be arbitrarily distributed in the interval $0 \mathinner{\ldotp\ldotp} n - 1$, where n is the total number of nodes of the input graph of the current level. Consequently, we start the parallel contraction algorithm by finding the number of distinct cluster IDs, which is also the number of coarse nodes. To do so, a PE p is assigned to count the number of distinct cluster IDs in the interval $I_p := p\lceil\frac{n}{P}\rceil + 1 \,..\, (p+1)\lceil\frac{n}{P}\rceil$, where P is the total number of PEs used. That means each PE p iterates over its local nodes, collects cluster IDs a that are not local, i.e., $a \notin I_p$, and then sends the non-local cluster IDs to the responsible PEs. Afterwards, a PE counts the number of distinct local cluster IDs so that the number of global distinct cluster IDs can be derived by using a reduce operation.

Let n' be the global number of distinct cluster IDs. Recall that this is also the number of coarse nodes after the contraction has been performed. The next step in the parallel contraction algorithm is to compute a mapping $q : 0 \,..\, n-1 \rightarrow 0 \,..\, n'-1$ which maps the current cluster IDs to a contiguous interval over all PEs. This mapping can be easily computed in parallel by computing a prefix sum over the number of distinct local cluster IDs a PE has. Once this is done, we compute the mapping $C : 0 \,..\, n-1 \rightarrow 0 \,..\, n'-1$ which maps a node ID of G to its coarse representative. Note that, if a node v is in cluster V_ℓ after the label propagation algorithm has converged, then $C(v) = q(\ell)$. After computing this information locally, we also propagate the necessary parts of the mapping to neighboring PEs so that we also know the coarse representative of each ghost node. When the contraction algorithm is fully completed, PE p will be *responsible* for the subgraph $p\lceil\frac{n'}{P}\rceil + 1 \,..\, (p+1)\lceil\frac{n'}{P}\rceil$ of the coarse graph. To construct the final coarse graph, we first construct the weighted quotient graph of the local subgraph of G using hashing. Afterwards, each PE sends an edge (u, v) of the local quotient graph, including its weight and the weight of its source node, to the responsible PE. After all edges are received, a PE can construct its coarse subgraph locally.

The implementation of the *parallel uncoarsening* algorithm is simple. Each PE knows the coarse node for all its nodes in its subgraph (through the mapping C). Hence, a PE requests the block ID of a coarse representative of a fine node from the PE that holds the respective coarse node. This can be implemented using an all-to-all operation.

4.4 Iterated Multilevel Schemes

A common approach to obtain high quality partitions is to use a multilevel algorithm multiple times using different random seeds and use the best partition that has been found. However, one can do better by transferring the solution of the previous multilevel iteration down the hierarchy. In the graph partitioning context, the notion of V-cycles was introduced by Walshaw [31]. More recent work augmented them to more complex cycles [32]. These previous works use matching-based coarsening with cut edges not being matched (and hence cut edges are not contracted). Thus, an input partition on the finest level is used as partition of the coarsest graph—having the same balance and cut as the partition of the finest graph.

Iterated V-cycles are also used within clustering-based coarsening in our previous work [7]. To adapt the iterated multilevel technique for this coarsening scheme, it has to be ensured that cut edges are not contracted after the first multilevel V-cycle. This is done by modifying the label propagation algorithm such that each cluster of the computed clustering is a subset of a block of the input partition. In other words, each cluster only contains nodes of one unique block of the input partition. Hence, when contracting the clustering, every cut edge of the input partition will remain. Recall that the label propagation algorithm initially puts each node in its own block so that in the beginning of the algorithm each cluster is a subset of one unique block of the input partition. This property is kept during the course of the label propagation algorithm by restricting the movements of the label propagation algorithm, i.e., we move a node to an eligible cluster with the strongest connection in its neighborhood that is in the same block of the input partition as the node itself. We do the same in our parallel approach to realize V-cycles.

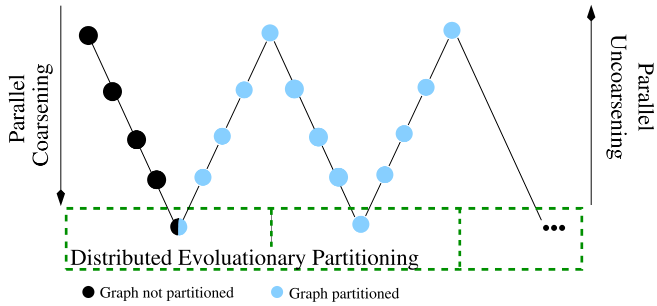


Fig. 4. The overall parallel system. It uses the parallel cluster coarsening algorithm, the coarse-grained distributed evolutionary algorithm KaFFPaE to partition the coarsest graph and parallel uncoarsening/local search. After the first iteration of the multilevel scheme the input partition is used as a partition of the coarsest graph and used as a starting point by the evolutionary algorithm.

4.5 The Overall Parallel System

The overall parallel system works as follows. We use ℓ iterations of the parallel size-constrained label propagation algorithm to compute graph clusterings and contract them in parallel. We do this recursively until the remaining graph has less than 20,000 nodes left. The distributed coarse graph is then collected on each PE, i.e., each PE has a copy of the complete coarsest graph. We use this graph as input to the coarse-grained distributed evolutionary algorithm KaFFPaE, to obtain a high quality k-partition of it. We have modified KaFFPaE to use combine operations that also use the clustering-based coarsening scheme from above. The best solution of the evolutionary algorithm is then broadcast to all PEs which transfer the solution to their local part of the distributed coarse graph. Afterwards, we use the parallel uncoarsening algorithm to transfer the solution of the current level to the next finer level and apply r iterations of the parallel label propagation algorithm with the size constraints of the original partitioning problem (setting $W = L_{\max}$) to improve the solution on the current level. We do this on each level of the hierarchy and obtain a good partition of the input network in the end. If we use iterated V-cycles, we use the given partition of the coarse graph as input to the evolutionary algorithm. More precisely, one individual of the population is the input partition on each PE. This way it is ensured that the evolutionary algorithm computes a partition that is at least as good as the given partition. Note that our initial partitioner is usually able to compute partitions that fulfill the desired balance constraint on the coarsest level. Hence, to ensure that the final partition of our parallel algorithm is balanced, we do not perform any parallel local search during the last V-cycle. A sketch of the overall system is shown in Fig. 4.

5 Experiments

5.1 Methodology

We have implemented the algorithm described above using C++ and MPI. Overall, our parallel program consists of about 7,000 lines of code (not including the source of KaHIP 0.61). We make our program available in the KaHIP framework [33]. We compiled it using g++ 4.8.2 and OpenMPI 1.6.5. For the following comparisons we use ParMetis 4.0.3. All programs have been compiled using 64 bit index data types. We also ran PT-Scotch 6.0.0, but the results have been consistently worse in terms of solution quality and running time compared to the results computed by ParMetis, so that we do not present detailed data for PT-Scotch. A few additional comparisons, in particular to PuLP, are shown at the end of this section.

Our default value for the allowed imbalance is 3 percent—this is one of the values used in [34] and the default value in Metis. By default we perform ten repetitions for each configuration of our algorithm and ParMetis using different random seeds for initialization and report the arithmetic average of computed cut size, running time and the best cut found. When further averaging over multiple instances, we use the *geometric mean* in order to give every instance a comparable influence on the final score. Unless otherwise stated, we use the following factor f of the size constraint (see Section 4.2 for the definition): during the first V-cycle the factor f is set to 14 on complex networks and to 20,000 on mesh type networks. In later V-cycles we use a random value $f \in_{\text{rnd}} [10, 25]$ to increase the diversification of the algorithms. Our experiments mainly focus on the cases $k \in \{2, 8, 32\}$ to save running time and to keep the experimental evaluation simple. Moreover, we use $k = 16$ for the number of blocks when performing the weak scalability experiments in Section 5.2.

5.1.1 Algorithm Configurations

Any multilevel algorithm has a considerable number of choices between algorithmic components and tuning parameters. For the tuning parameters that we set here, we get the predictable effect that more work yields better solutions albeit at a decreasing return on investment. We define two "good" choices: the fast setting aims at a low execution time that still gives good partitioning quality and the eco setting targets even better partitioning quality without investing an excessive amount of time. When not otherwise mentioned, we use the parameter set of the fast configuration.

The fast configuration of our algorithm uses three label propagation iterations during coarsening and six during refinement. We also tried larger amounts of label propagation iterations during coarsening, but did not observe a significant impact on solution quality. This configuration gives the evolutionary algorithm only enough time to compute the initial population and performs two V-cycles.

The eco configuration of our algorithm also uses three label propagation iterations during coarsening and six label propagation iterations during refinement, but performs five V-cycles. Time spent during initial partitioning is dependent on the number of processors used. To be more precise, when we use one PE, the evolutionary algorithm has $t_1 = 2^{11}$ seconds to compute a partition of the coarsest graph during the first V-cycle. When we use p PEs, then it gets time $t_p = t_1/p$ to compute a partition of an instance.

There is also a minimal variant of the algorithm, which is similar to the fast configuration but only performs one V-cycle. We use this variant of the algorithm only in one scenario—to create a partition of the largest web graph uk-2007 on machine B (described below).

TABLE 1
Basic Properties of the Benchmark Set with a Rough Type Classification

graph	n	m	Type	Ref.
amazon	≈407 K	≈2.3 M	C	[35]
eu-2005	≈862 K	≈16.1 M	C	[36]
youtube	≈1.1 M	≈2.9 M	C	[35]
in-2004	≈1.3 M	≈13.6 M	C	[36]
packing	≈2.1 M	≈17.4 M	M	[36]
enwiki	≈4.2 M	≈91.9 M	C	[37]
channel	≈4.8 M	≈42.6 M	M	[36]
hugebubble-10	≈18.3 M	≈27.5 M	M	[36]
nlpkkt240	≈27.9 M	≈373 M	M	[38]
uk-2002	≈18.5 M	≈262 M	C	[37]
del26	≈67.1 M	≈201 M	M	[18]
rgg26	≈67.1 M	≈575 M	M	[18]
rhg1G	100.0 M	≈1 G	C	[39]
rhg2G	100.0 M	≈2 G	C	[39]
arabic-2005	≈22.7 M	≈553 M	C	[37]
sk-2005	≈50.6 M	≈1.8 G	C	[37]
uk-2007	≈105.8 M	≈3.3 G	C	[37]
rhg6G	300.0 M	≈6 G	C	[39]
Graph Families				
delX	$[2^{19}, \dots, 2^{31}]$	≈1.5 M–6.4 G	M	[18]
rggX	$[2^{19}, \dots, 2^{31}]$	≈3.3 M–21.9 G	M	[18]

C stands for complex networks, M is used for mesh type networks.

5.1.2 Systems

We use two different systems for our experimental evaluation. *System A* is mainly used for the evaluation of the solution quality of the different algorithms in Table 2. It is equipped with four Intel Xeon E5-4640 Octa-Core processors (Sandy Bridge) running at a clock speed of 2.4 GHz. The machine has 512 GB main memory, 20 MB L3-Cache and 8×256 KB L2-Cache. *System B* is a cluster where each node is equipped with two Intel Xeon E5-2670 Octa-Core processors (Sandy Bridge) which run at a clock speed of 2.6 GHz. Each node has 64 GB local memory, 20 MB L3-Cache and 8×256 KB L2-Cache. All nodes have local disks and are connected by an InfiniBand 4X QDR interconnect, which is characterized by its very low latency of about 1 microsecond and a point to point bandwidth between two nodes of more than 3,700 MB/s—2K cores of that machine can be allocated by users. We use machine B for the scalability experiments.

5.1.3 Instances

We evaluate our algorithms on graphs that we mostly collected from [35], [36], [38], [39], [40], [41]. Table 1 summarizes the main properties of the benchmark set. Our benchmark set includes a number of graphs from numeric simulations as well as complex networks (for the latter with a focus on social networks and web graphs).

The graphs rhg* are complex networks generated with NetworKit [39] according to the *random hyperbolic graph*

TABLE 2
Average Performance (Cut And Running Time) and Best Result Achieved by Different Partitioning Algorithms

$k = 2$	ParMetis			Fast			Eco		
graph	avg. cut	best cut	t[s]	avg. cut	best cut	t[s]	avg. cut	best cut	t[s]
amazon	48,104	47,010	0.49	46,641	45,872	1.85	*44,703*	**44,279**	71.04
eu-2005	33,789	24,336	30.60	20,898	18,404	1.63	*18,565*	**18,347**	70.04
youtube	181,885	171,857	6.10	174,911	171,549	8.74	*167,874*	**164,095**	105.87
in-2004	7,016	5,276	3.43	3,172	3,110	1.38	*3,027*	**2,968**	69.19
packing	11,991	11,476	0.24	10,185	9,925	1.84	*9,634*	**9,351**	68.69
enwiki	9,578,551	9,553,051	326.92	9,622,745	9,565,648	157.32	*9,559,782*	**9,536,520**	264.64
channel	*48,798*	**47,776**	0.55	56,982	55,959	2.71	52,101	50,210	71.95
hugebubbles	1,922	1,854	4.66	1,918	1,857	38.00	*1,678*	**1,620**	216.91
nlpkkt240	*1,178,988*	**1,152,935**	15.97	1,241,950	1,228,086	35.06	1,193,016	1,181,214	192.78
uk-2002	787,391	697,767	128.71	434,227	390,182	19.62	*415,120*	**381,464**	146.77
del26	18,086	17,609	23.74	17,002	16,703	165.02	*15,826*	**15,690**	697.43
rgg26	44,747	42,739	8.37	38,371	37,676	55.91	*34,530*	**34,022**	263.81
rhg1G	1,442	725	26.51	522	522	65.15	*518*	**518**	290.10
rhg2G	8,017	3,492	54.42	*1,952*	**1,950**	106.46	*1,952*	**1,950**	417.55
arabic-2005	*1,078,415	*968,871	*1,245.57	551,778	**471,141**	33.45	*511,316*	475,140	184.01
sk-2005	*	*	*	3,775,369	3,204,125	471.16	*3,265,412*	**2,904,521**	1,688.63
uk-2007	*	*	*	1,053,973	1,032,000	169.96	*1,010,908*	**981,654**	723.42
$k = 8$	ParMetis			Fast			Eco		
graph	avg. cut	best cut	t[s]	avg. cut	best cut	t[s]	avg. cut	best cut	t[s]
amazon	149,493	144,251	0.60	137,834	135,090	2.41	*131,295*	**129,627**	73.37
eu-2005	217,902	204,967	31.68	253,738	193,032	2.44	*187,859*	**163,405**	73.04
youtube	*530,822*	523,733	9.90	605,005	545,126	7.29	535,842	**511,966**	97.44
in-2004	21,293	20,073	4.85	14,838	14,114	1.81	*13,391*	**12,460**	70.50
packing	126,389	123,160	0.31	130,223	129,058	6.41	*116,659*	**115,620**	73.38
enwiki	22,346,150	22,223,189	349.78	22,555,856	22,120,305	237.79	*22,223,882*	**21,883,882**	313.21
channel	410,942	397,888	0.77	408,086	406,027	7.78	*362,985*	**360,114**	78.38
hugebubbles	11,032	10,688	6.10	10,661	10,561	43.68	*9,377*	**9,243**	189.18
nlpkkt240	*3,413,423*	**3,364,277**	19.11	3,710,051	3,686,347	43.97	3,497,526	3,457,692	186.92
uk-2002	2,021,162	1,962,461	435.12	1,273,662	1,244,025	22.39	*1,220,789*	**1,204,064**	144.95
del26	67,401	65,263	22.61	61,899	61,156	121.85	*56,908*	**56,597**	436.34
rgg26	170,608	165,753	9.28	142,380	140,843	51.12	*127,034*	**125,780**	239.45
rhg1G	7,344	6,686	27.80	2,700	2,624	66.10	*2,484*	**2,453**	288.43
rhg2G	39,813	31,992	54.68	12,182	11,583	103.32	*11,395*	**11,108**	428.38
arabic-2005	*2,842,365	*2,740,020	*1,482.36	1,914,633	1,515,775	32.94	*1,333,028*	**1,231,026**	183.33
sk-2005	*	*	*	17,780,585	13,572,997	619.93	*13,295,857*	**11,424,463**	2,649.67
uk-2007	*	*	*	3,298,092	3,147,335	166.73	*3,128,609*	**3,035,653**	648.81
$k = 32$	ParMetis			Fast			Eco		
graph	avg. cut	best cut	t[s]	avg. cut	best cut	t[s]	avg. cut	best cut	t[s]
amazon	253,568	249,071	0.62	235,614	231,169	3.20	*224,550*	**222,450**	81.83
eu-2005	*974,279*	**951,537**	33.28	1,218,484	1,154,916	3.30	1,089,613	1,010,128	79.87
youtube	918,520	916,657	10.41	951,591	936,333	13.86	*905,330*	**889,941**	137.61
in-2004	34,445	32,711	4.76	26,618	25,819	1.97	*23,795*	**22,371**	73.22
packing	349,000	343,611	0.28	338,458	335,732	24.65	*318,242*	**315,684**	92.55
enwiki	*32,539,098*	**32,279,759**	364.88	33,464,700	33,256,794	787.41	33,358,352	32,579,351	989.20
channel	934,264	**919,975**	0.63	989,570	983,211	26.23	*932,175*	927,128	100.86
hugebubbles	28,844	28,443	4.98	27,832	27,607	117.72	*25,358*	**25,102**	342.75
nlpkkt240	*7,296,962*	**7,217,145**	17.09	8,048,555	7,987,330	104.96	7,770,995	7,726,512	274.67
uk-2002	2,636,838	2,603,610	193.48	1,710,106	1,677,872	33.44	*1,635,757*	**1,610,979**	218.27
del26	167,208	165,361	23.04	153,835	152,889	274.75	*145,902*	**145,191**	859.33
rgg26	423,643	419,911	8.14	356,589	352,749	125.07	*326,743*	**323,997**	376.27
rhg1G	32,105	29,268	27.86	12,090	11,806	70.07	*11,413*	**10,963**	289.44
rhg2G	155,739	134,854	56.97	56,120	54,278	102.53	*54,047*	**52,795**	422.50
arabic-2005	*4,095,660	*3,993,166	*1,414.83	3,309,602	2,648,126	45.40	*2,372,631*	**2,178,837**	251.38
sk-2005	*	*	*	58,107,145	46,972,182	693.91	*34,858,430*	**29,868,523**	2,183.10
uk-2007	*	*	*	5,682,545	5,114,349	223.68	*4,952,631*	**4,779,495**	794.87

*Results are for $k = 2$ (top), $k = 8$ (middle) and for $k = 32$ (bottom). All tools used 32 PEs of machine A. Results indicated by a * mean that the amount of memory needed by the partitioner exceeded the amount of memory available on that machine when 32 PEs are used (512 GB RAM). The ParMetis result on arabic have been obtained using 15 PEs (the largest number of PEs so that ParMetis could solve the instance).*

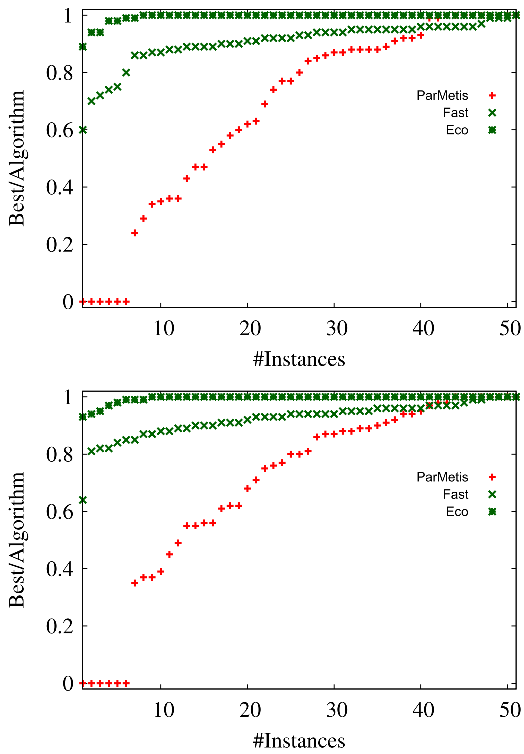


Fig. 5. Performance plots (top: Average cuts, bottom: Best cuts). A value of one indicates that the corresponding algorithm produces the best solution.

model [42]. In this model nodes are represented as points in the hyperbolic plane; nodes are connected by an edge if their hyperbolic distance is below a threshold. Moreover, we use the two graph families rgg and del for comparisons. rgg$X$ is a *random geometric graph* with 2^X nodes where nodes represent random points in the (Euclidean) unit square and edges connect nodes whose euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost certainly connected. The largest graph of this class is rgg31, which has about 21.9 G edges. del$X$ is a Delaunay triangulation of 2^X random points in the unit square. Our largest del$X$ is del31; it has about 6.4 G edges.

The largest graphs (with 2^{26} to 2^{31} nodes) of these families have been generated using modified code from [18]. We make these graphs available on request.

5.2 Main Results and Comparison to ParMetis

In this section we compare variants of our algorithm against ParMetis in terms of solution quality, running time as well as scalability. We start with the comparison of solution quality (average cut, best cut) and average running time on most of the graphs from Table 1 when 32 PEs of machine A are used. Table 2 gives detailed results per instance for the cases $k = \{2, 8, 32\}$. To get a visual impression of the solution quality of the different algorithms, Fig. 5 presents *performance plots* using most instances from Table 2. A curve in a performance plot for algorithm X is obtained as follows: For each instance, we calculate the ratio between the best cut obtained by any of the considered algorithms and the cut for algorithm X. These values are then sorted. The balance constraint of $\epsilon = 3$ percent was fulfilled for our algorithms and for ParMetis unless mentioned otherwise.

First of all, ParMetis could not solve the large instances arabic-2005, sk-2005 and uk-2007 when 32 PEs of machine A are used. This is due to the fact that ParMetis cannot coarsen the graphs effectively so that the coarsening phase is stopped too early. Since the smallest graph is replicated on each of the PEs, the amount of memory needed by ParMetis is larger than the amount of memory provided by the machine (512 GB RAM). For example, when the coarsening phase of ParMetis stops on the instance uk-2007, the coarsest graph still has more than 60 M vertices. This is less than a factor of two reduction in graph order compared to the input network. The same behavior is observed on machine B, where even less memory per PE is available. Contrarily, our algorithm is able to shrink the graph order significantly. For instance, after the first contraction step, the graph is already two orders of magnitude smaller and contains a factor of 300 less edges than the input graph uk-2007. We also tried to use a smaller amount of PEs for ParMetis. It turns out that ParMetis can partition arabic-2005 when using 15 PEs, cutting nearly twice as many edges and consuming thirty-seven times more time than our fast variant. Moreover, ParMetis could not solve the instances sk-2005 and uk-2007 for any number of PEs.

Overall, Fig. 5 indicates that our algorithms find significantly smaller cuts than ParMetis. When only considering the networks that ParMetis could solve in Table 2, our fast and eco configuration compute cuts that are 29.0 and 40.4 percent smaller on average than the cuts computed by ParMetis, respectively. On average, fast and eco need more time to compute a partition, i.e., fast and eco need 60 percent and roughly a factor 12 more time on average, respectively.

Moreover, there is a well defined *gap* between mesh type networks and complex networks, as described below. The reason for this discrepancy is that meshes usually do not have a community structure to be found and contracted by our algorithm. Complex networks, in turn, often feature a hierarchical cluster structure that our algorithm exploits. This is in particular true for social networks and also for web graphs with their domain-based cluster structure. Random hyperbolic graphs have been shown to have a hierarchical structure, too [42].

Considering only the complex networks, our fast algorithm is more than a factor two faster on average and improves the cuts produced by ParMetis by 49.8 percent (the eco configuration computes cuts that are 63.3 percent smaller than the cuts computed by ParMetis). The largest speedup over ParMetis in Table 2 was obtained on eu-2005 ($k = 2$) where our algorithm is more than 18 times faster than ParMetis and cuts 61.6 percent less edges on average. The largest improvement over ParMetis was obtained for $k = 2$ on the largest random hyperbolic graph rhg2G. Here, ParMetis cuts more than four times as many edges on average as our fast configuration.

In contrast, on mesh type networks our algorithm does not have the same advantage as on complex networks. For example, our fast configuration improves on ParMetis by 3 percent while needing more than ten times as much running time. This is due to the fact that this type of network usually has no community structure so that the graph sizes do not shrink as fast. Still, the eco configuration computes 12.0 percent smaller cuts than ParMetis. To obtain a fair

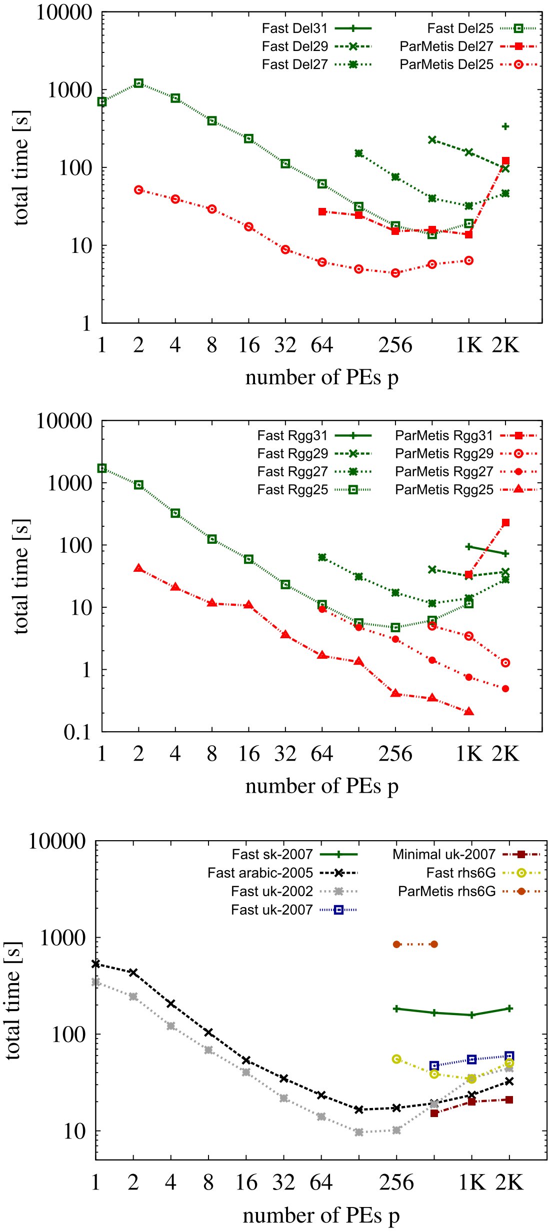


Fig. 6. *Top:* Strong scaling experiments on Delaunay networks. The largest graph that ParMetis could partition from this graph family was del27. *Middle:* Strong scaling experiments on random geometric networks. *Bottom:* Strong scaling experiments on the largest complex networks from our benchmark set. Plots always start when sufficient memory is available to partition the input graph. Due to ineffective coarsening, ParMetis was *not able* to partition any of these graphs on machine B. On the graph rhg6G, ParMetis did not finish after one hour of computation for $p \in \{1K, 2K\}$. On the largest web graph, uk-2007, we also used the minimal variant of our algorithm. *Note*, although our system is not built for mesh-type networks such as Delaunay and random geometric graphs, we can partition larger instances and compute better solutions than ParMetis.

comparison on this type of networks, we also compare the best cut found by ParMetis against the average cuts found by our algorithms. While the best cuts on mesh type networks of ParMetis are comparable to the average results of our fast configuration, the eco configuration still yields 9.3 percent smaller cuts on average. Also note, that ParMetis simplifies the problem $k = 32$ by relaxing it: On some instances it does not respect the balance constraint and computes partitions with up to 6 percent imbalance.

5.2.1 Scalability

To evaluate *strong scalability*, we use a subset of the random geometric and Delaunay graphs as well as the five complex networks arabic-2005, uk-2002, sk-2007, uk-2007 and rhg6G. In all cases we use up to 2,048 cores of machine B (except for del25 and rgg25, for which we only scaled up to 1,024 cores) and partition the graphs into $k = 2$ blocks. Again, we focus on the fast configuration of our algorithm and ParMetis to save running time. Fig. 6 summarizes the results of the experiments. First of all, we observe that the largest instances del29 and rgg31 experience decent running time improvements when the number of processors is increased from 512 or 1,024 to 2,048. Using all 2,048 cores, we need roughly 6.5 minutes to partition del31 and 73 seconds to partition rgg31. Note that the rgg31 graph has three times more edges than del31 but the running time needed to partition del31 is higher. This is due to the fact that the Delaunay graphs have very bad locality (due to the graph generator), i.e., when partitioning del31, more than 40 percent of the edges are ghost edges, whereas we observe less than 0.5 percent ghost edges when partitioning the largest random geometric graph. Although the scaling behavior of ParMetis is somewhat better on the random geometric graphs rgg25-29, our algorithm is eventually more than three times faster on the largest random geometric graph under consideration when all 2,048 cores are used. As a side note, the large running times of ParMetis for large number of processors seems to be due to a problem with the matching routine of ParMetis. Moreover, the quality of the partitions does not degrade in our strong scaling experiments neither for ParMetis nor for our algorithm.

As on machine A, ParMetis could not partition the instances uk-2002, arabic-2005, sk-2007 and uk-2007—this is again due to the amount of memory needed arising from ineffective coarsening. On the smaller graphs, uk-2002 and arabic-2005, our algorithm scales up to 128 cores obtaining a 35-fold and 32-fold speed-up compared to the case where our algorithm uses only one PE. On the larger graphs sk-2007, uk-2007 and rhg6G, we need more memory. The smallest number of PEs needed to partition sk-2007, uk-2007 and rhg6G on machine B is 256 PE, 512 PEs and 256 PEs respectively. We observe scalability up to 1 K cores on the graph sk-2007 and rhg6G (although, to be fair, the running time does not decrease much in that area). On uk-2007 we do not observe further scaling when switching from 512 to 2,048 cores so that it is unclear where the sweet spot is for this graph. The random hyperbolic graph rhg6G is the only complex network that ParMetis can partition in this setting. In this case, our algorithm is a factor 15 (for 256 PEs) to 22 (for 512 PEs) faster while computing cuts that are more than a factor three smaller. This is again due to ineffective coarsening—ParMetis spends almost all its running time in this part of the multilevel scheme. We have also applied the minimal configuration on machine B to the largest web graph uk-2007 in our set. The minimal configuration needs 15.2 seconds to partition the graph when 512 cores are used. The cut is 18.2 percent higher compared to the cut of the fast configuration, which needs ≈ 47 seconds to perform the partitioning task and cuts ≈ 1.03 M edges on average. This is 57 times faster than partitioning the graph with one core of machine A (which is faster than machine B). In this case,

TABLE 3
Average Performance (Cut and Running Time) and Best Result Achieved by PuLP [26]

$k=2$	avg. cut	best cut	t[s]
enwiki	11,048,842	10,193,062	14.92
channel	*51,841*	47,928	0.72
hugebubbles	4,206	2,405	37.58
nlpkkt240	1,541,633	1,259,553	3.25
uk-2002	1,735,517	1,481,055	4.53
del26*	108,149	70,202	37.91
rgg26	118,444	44,552	26.94
rhg1G	1,268,559	636,519	14.31
rhg2G	924,326	558,614	18.38
arabic-2005	2,660,263	1,912,044	6.75
sk-2005	8,113,117	6,445,828	31.19
uk-2007*	6,696,972	0*	36.55
$k=8$	avg. cut	best cut	t[s]
enwiki	28,614,478	24,829,587	17.87
channel	626,746	468,540	0.19
hugebubbles*	20,730	16,468	0.00
nlpkkt240	4,612,360	4,084,985	1.40
uk-2002	2,788,216	2,398,216	5.61
del26*	264,068	193,484	0.00
rgg26*	554,313	405,153	0.65
rhg1G	3,030,736	1,701,945	21.74
rhg2G	2,577,837	1,960,672	26.44
arabic-2005*	27,577,343	3,488,600	6.80
sk-2005	27,006,839	21,666,581	48.42
uk-2007*	11,148,115	9,592,812	39.15
$k=32$	avg. cut	best cut	t[s]
enwiki	40,491,967	38,048,077	19.90
channel	1,433,171	1,211,712	1.47
hugebubbles	47,707	42,052	43.63
nlpkkt240	9,146,706	8,571,499	6.05
uk-2002	3,506,131	3,352,617	6.85
del26	353,609	304,965	48.81
rgg26*	1,155,233	893,465	69.44
rhg1G	4,413,737	3,724,520	35.30
rhg2G	3,740,828	3,274,015	35.29
arabic-2005	6,338,579	4,920,370	11.81
sk-2005	52,996,595	42,575,940	66.90
uk-2007*	56,924,295	11,516,718	49.25

*Results are for the bipartitioning case $k=2$ (top), $k=8$ (middle) and for $k=32$ (bottom). The algorithm used 32 PEs of machine A. Results marked with a * indicate that some of the computed partitions did not fulfill the balance constraint.*

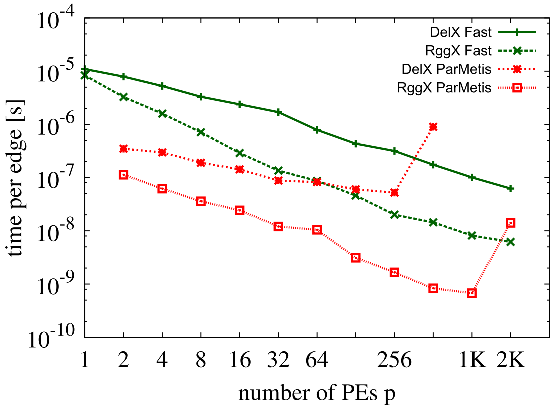
our algorithm spends roughly 60 percent of its total running time in coarsening, 3 percent in initial partitioning and 37 percent in uncoarsening. However, the running time of uncoarsening exceeds coarsening time during the first V-cycle. Hence, the lower contribution of uncoarsening in total running time is due to the fact that local search starts already from a very good partition in later V-cycles, and therefore needs much less time in these cases.

We have also performed *weak scalability* experiments on machine B using the graph families rgg$X$ and del$X$, and use $k=16$ for the number of blocks for the partitioning task. We briefly outline the results. Moreover, we focus on the fast configuration of our algorithm and ParMetis to save running time. We expect that the scalability of the eco configuration of our algorithm is similar. When using p PEs, the instance with $2^{19}p$ nodes from the corresponding graph class is used, i.e., when using 2,048 cores, all algorithms partition the graphs del30 and rgg30. Our algorithm shows weak scalability *all the way down* to the largest number of cores used while the running time per edge has a somewhat stronger descent compared to ParMetis. ParMetis could, again, not solve some of the largest instances. For example, the largest Delaunay graph that ParMetis could partition was del28 using 512 cores. Considering the instances that ParMetis could solve, our fast configuration improves solution quality by 19.5 percent on random geometric graphs and by 11.5 percent on Delaunay triangulations on average. Since the running time of the fast configuration is mostly slower on both graph families, we again compare the best cut results of ParMetis achieved in ten repetitions against our average results to obtain a fair comparison (in this case ParMetis has a slight advantage in terms of running time). Doing so, our algorithm still yields an improvement of 16.8 percent on the random geometric graphs and an improvement of 9.5 percent on the Delaunay triangulations. For large number of processors and the largest instances, ParMetis is slower than the fast version of our partitioner. On the largest random geometric graph used during this test, we are about a factor two faster than ParMetis, while improving the results of ParMetis by 9.5 percent. In this case our partitioner needs roughly 65 seconds to compute a 16-partition of the graph. In addition, our algorithm is a factor five faster on the largest Delaunay graph that ParMetis could solve and produces a cut that is 9.5 percent smaller than the cut produced by ParMetis.

5.3 Additional Comparisons

Recall that the software PuLP [26] partitions complex networks in a single-level manner. PuLP uses shared-memory parallelism and is able to optimize multiple constraints. Moreover, it is very fast and has a small memory footprint as it avoids the multilevel overhead. The latter comes with a price, though, the solution quality. With the help of its authors, we configured PuLP to run without the balance constraint on the number of edges on the blocks in order to obtain a fair comparison. As shown in Table 3 and Fig. 7, which contain results for the twelve largest graphs in our benchmark set that fit into the memory of machine A, PuLP often cuts significantly more edges than our algorithm configurations. While for some instances the quality is comparable (or on the instance channel for $k=2$ even with a better average cut), many comparisons favor our algorithm configurations by a high margin. Our improvement seems particularly high for most web graphs as well as the synthetic graphs del26, rgg26, rhg1G and rhg2G. It will depend on the application which time-quality ratio is preferred by the user.

In another experiment, we perform one coarsening step using our cluster-based algorithms and then apply ParMetis on the coarsened graph (Contraction+ParMetis). Again, we compute results for the twelve largest graphs in our benchmark set. Detailed results are shown in Table 4 and Fig. 7. The data indicates that indeed performing one cluster coarsening step and applying ParMetis on the coarsened graph improves solution quality and running time on complex networks on average. The opposite is the case for networks that do not have a complex structure. This is similar to the experiments comparing our new algorithm against ParMetis in Section 5.2. For example, on the complex instances



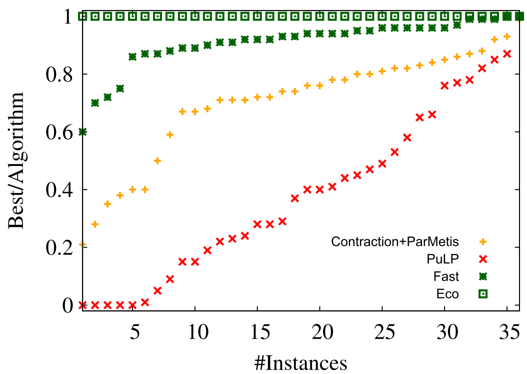


Fig. 7. *Top:* Weak scaling experiments for random geometric graph class rggX and the Delaunay triangulation graph class delX. When using p PEs, the instance with $2^{19}p$ nodes from the corresponding graph class was used, i.e., when using 2,048 cores all algorithms partition the graphs del30 and rgg30. The figure shows the time spent per edge. Sixteen blocks have been used for the partitioning task. *Bottom:* Performance plots for average cuts comparing against PuLP and one level cluster contraction and running ParMetis on the coarsened graph. A value of one indicates that the corresponding algorithm produces the best solution.

that ParMetis could solve in its original version, Contraction+ParMetis improves cuts over ParMetis by 16 percent on average while reducing the running time roughly by a factor five on average. On mesh type networks, solution quality achieved by Contraction+ParMetis is increased by 17 percent on average while running time also increases by factor of two. Overall, still many comparisons favor our algorithm configurations by a high margin (see Fig. 7). Additionally, using one cluster-based coarsening step enables ParMetis to solve the largest instances sk-2005 and uk2007, which the original version of ParMetis cannot solve.

Furthermore, we briefly compare to other algorithms based on data presented in the literature. As a matching-based multilevel algorithm, KaPPa [18] has similar problems as ParMetis on complex networks. For example, on coAuthorsDBLP and citationCiteseer used in [18], our new algorithm cuts 20 and 31 percent less edges than KaPPa-fast, while being a factor 29 and a factor 48 faster ($k = 16$). Note that KaPPa is restricted to the (also) important use case $\#p = k$. However, it is not very scalable on complex networks. Due to the large cuts that occur for large values of k on complex networks, we want to use recursive multi-partitioning in future work to improve the system presented in this work for that case. As opposed to multilevel algorithms, the algorithm by Ugander and Backstrom [25] lacks a global view on the problem. We cut 45 percent less edges than their approach on LiveJournal, which is the only publicly available graph from the paper. Moreover, we are a factor 26 faster ($k = 100$). The results of Kirmani and Raghavan [21] are incomparable since a relaxed problem is solved and the partition imbalance is not reported. As argued in Section 2.2, we do not expect it to perform well on complex networks.

TABLE 4
Average Performance (Cut and Running Time) and Best Result Achieved by ClusterContraction+ParMetis

$k = 2$	avg. cut	best cut	$t[s]$	$t_{\mathrm{pm}}[s]$
enwiki	12,896,672	12,877,326	58.34	30.97
channel	63,130	61,960	1.14	0.05
hugebubbles	2,194	2,085	11.33	1.49
nlpkkt240	1,484,520	1,456,834	12.17	0.17
uk-2002	473,740	454,811	10.08	2.72
del26	19,165	18,794	53.66	2.50
rggn26	48,780	47,153	25.90	0.26
rhg1G	1,305	871	39.51	1.46
rhg2G	9,237	4,790	77.51	0.77
arabic-2005	551,809	482,017	14.32	1.33
sk-2005	3,295,324	2,652,245	49.80	3.50
uk-2007	1,711,982	1,575,003	84.96	8.96
$k = 8$	avg. cut	best cut	$t[s]$	$t_{\mathrm{pm}}[s]$
enwiki*	31,076,421	30,839,223	99.42	60.61
channel	514,723	506,940	1.32	0.06
hugebubbles	12,621	12,343	14.58	1.71
nlpkkt240	4,318,815	4,256,153	11.99	0.17
uk-2002	1,408,630	1,388,682	13.82	5.80
del26	70,908	68,945	65.34	3.88
rggn26	187,866	183,286	34.74	0.34
rhg1G	6,503	4,918	41.69	2.96
rhg2G	40,959	25,635	78.69	1.47
arabic-2005	1,561,553	1,491,810	15.60	2.91
sk-2005	19,755,401	19,094,883	75.04	8.86
uk-2007	4,319,060	4,205,929	87.07	15.83
$k = 32$	avg. cut	best cut	$t[s]$	$t_{\mathrm{pm}}[s]$
enwiki*	49,663,702	49,114,887	176.35	130.31
channel	1,189,504	1,181,686	2.23	0.19
hugebubbles	33,570	33,237	13.44	1.85
nlpkkt240	9,905,776	9,828,180	13.68	0.40
uk-2002	1,897,402	1,882,320	13.35	6.20
del26	177,005	175,514	58.61	3.79
rggn26	459,999	449,471	26.42	0.38
rhg1G	22,834	21,256	42.34	3.02
rhg2G	155,477	143,700	78.16	1.46
arabic-2005	2,579,827	2,527,410	16.01	3.02
sk-2005*	86,645,315	84,477,965	83.89	18.21
uk-2007*	5,902,226	5,841,361	92.87	15.38

*Results are for the bipartitioning case $k = 2$ (top), $k = 8$ (middle) and for $k = 32$ (bottom). The algorithm used 32 PEs of machine A. Results marked with a * indicate that some of the computed partitions did not fulfill the balance constraint. Column t_{pm} shows the average time that ParMetis needed to partition the coarser graph and column $t[s]$ shows the overall time needed.*

6 Conclusion and Future Work

Current state-of-the-art graph partitioners have difficulties when partitioning massive complex networks, at least partially due to ineffective coarsening. We have demonstrated that high quality partitions of such networks can be obtained in parallel using hundreds or sometimes thousands of processors. This was achieved by using a multilevel scheme based on the contraction of size-constrained clusterings, which can reduce the size of the graph very fast. The clusterings have been computed by our new parallelization of the size-constrained label propagation algorithm. As soon as the graph is small enough, we use a coarse-grained distributed memory parallel evolutionary algorithm to compute a high quality partitioning of the graph. By using the size constraint of the graph partitioning problem to solve, the parallel label propagation algorithm is also used as a very simple, yet effective, local search algorithm. Moreover, by integrating techniques like V-cycles and the evolutionary algorithm on the coarsest level, our system gives the user a gradual choice to trade solution quality for running time.

The strengths of our new algorithm unfolds in particular on complex networks such as social networks and web graphs, where average solution quality *and* running time is much better than what is observed by using ParMetis. This is due to the fact that, unlike matching-based approaches, our algorithm tends to find the inherent cluster hierarchy and avoids the contraction of important inter-cluster edges. Due to the ability to shrink complex networks drastically, our algorithm is able to compute high quality partitions of web scale networks in a matter of seconds, whereas ParMetis quite often fails to compute any partition. Considering the good results of our algorithm, we want to further improve and release its implementation.

Despite the progress reported above, we see numerous remaining challenges. While quality improvement for the small, mostly mesh-like graphs from the Walshaw benchmark [43] have stagnated in recent years, with no or only single digit percentages of improvement, the significant improvements we report for large complex networks raise the question whether we are even close to optimality yet. We suspect that further significant gains are possible.

Similarly, both ParMetis and our system scale quite well on large mesh-like graphs whereas even our system cannot effectively use more than around a 1,000 cores while the largest supercomputers out there now count *millions* of cores. One approach might be to rethink what we mean with partitioning. At least inside the partitioner itself, we might want to go away from plain 1D partitioning of the adjacency matrix in order to remove the bottlenecks introduced by nodes with very high degree. At the same time, current codes have problems computing balanced partitions for large number of blocks k. This can be addressed by recursively partitioning with a small value of k but new direct approaches are an interesting alternative for future work.

ACKNOWLEDGMENTS

We thank George Slota for providing the PuLP source code used in our experimental comparison and Moritz von Looz for providing the hyperbolic random graphs. We also would like to thank the Steinbuch Centre of Computing for giving us access to the IC2 machine. This work was partially supported by the German Research Foundation (DFG) grant TEAM (ME 3619/2-1). This paper has appeared in preliminary and shorter form in the proceedings of the *29th IEEE International Parallel & Distributed Processing Symposium* [1].

REFERENCES

[1] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 1055–1064. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.18

[2] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Comput.*, vol. 26, no. 12, pp. 1519–1534, 2000.

[3] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified NP-complete problems," in *Proc. 6th ACM Symp. Theory Comput.*, 1974, pp. 47–63.

[4] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is NP-hard," *Inf. Process. Lett.*, vol. 42, no. 3, pp. 153–159, 1992.

[5] P. Sanders and C. Schulz, "Distributed evolutionary graph partitioning," in *Proc. 12th Workshop Algorithm Eng. Experimentation*, 2012, pp. 16–29.

[6] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proc. ACM/IEEE Conf. Supercomputing*, 1996, pp. 35–35.

[7] H. Meyerhenke, P. Sanders, and C. Schulz, "Partitioning complex networks via size-constrained clustering," in *Proc. 13th Int. Symp. Exp. Algorithms*, 2014, pp. 351–363.

[8] K. Schloegel, G. Karypis, and V. Kumar, "Graph partitioning for high performance scientific simulations," in *The Sourcebook of Parallel Computing*. San Mateo, CA, USA: Morgan Kaufmann, 2003, pp. 491–541.

[9] C. Bichot and P. Siarry, Eds.,*Graph Partitioning*. Hoboken, NJ, USA: Wiley, 2011.

[10] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Eng. – Sel. Results and Surveys*, pp. 117–158, 2016, (Online). Avalibale: http://dx.doi.org/10.1007/9783319494876_4

[11] R. V. Southwell, "Stress-calculation in frameworks by the method of "systematic relaxation of constraints"," *Proc. Roy. Soc. London*, vol. 151, no. 872, pp. 56–95, 1935.

[12] C. Walshaw and M. Cross, "JOSTLE: Parallel multilevel graph-partitioning software—An overview," in *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Stirlingshire, U.K.: Saxe-Coburg Publications, 2007, pp. 27–58.

[13] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[14] F. Pellegrini. (1992). Scotch home page. [Online]. Available: http://www. labri.fr/pelegrin/scotch

[15] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, Art. no. 10.

[16] H. Meyerhenke, B. Monien, and S. Schamberger, "Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, Art. no. 10.

[17] C. Chevalier and F. Pellegrini, "PT-scotch," *Parallel Comput.*, vol. 34, no. 6–8, pp. 318–331, 2008.

[18] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," in *Proc. 24th Int. Parallal Distrib. Process. Symp.*, 2010, pp. 1–12.

[19] H. Meyerhenke, "Shape optimizing load balancing for MPI-parallel adaptive numerical simulations," in *Proc. 10th DIMACS Implementation Challenge—Graph Partitioning Graph Clustering*, 2013, pp. 67–82.

[20] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From" think like a vertex" to" think like a graph," *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[21] S. Kirmani and P. Raghavan, "Scalable parallel graph partitioning," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2013, Art. no. 51.

[22] A. Nocaj, M. Ortmann, and U. Brandes, "Untangling the hairballs of multi-centered, small-world online social media networks," *J. Graph Algorithms Appl.*, vol. 19, no. 2, pp. 595–618, 2015.

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

[24] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, Jul. 1999.

[25] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. 6th Int. Conf. Web Search Data Mining*, 2013, pp. 507–516.

[26] G. M. Slota, K. Madduri, and S. Rajamanickam, "Complex network partitioning using label propagation," *SIAM J. Sci. Comput.*, vol. 38, pp. S620–S645, 2016.

[27] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2013, pp. 50:1–50:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503293

[28] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 568–579.

[29] P. Sanders and C. Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *Proc. 12th Int. Symp. Exp. Algorithms*, 2013, pp. 164–175.

[30] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Conf. Des. Autom.*, 1982, pp. 175–181.

[31] C. Walshaw, "Multilevel refinement for combinatorial optimisation problems," *Ann. Operations Res.*, vol. 131, no. 1, pp. 325–372, 2004.

[32] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *Proc. 19th Eur. Symp. Algorithms*, 2011, pp. 469–480.

[33] P. Sanders and C. Schulz. (2013). KaHIP—Karlsruhe high qualtity partitioning homepage. [Online]. Available: http://algo2.iti.kit.edu/documents/kahip/index.html

[34] C. Walshaw and M. Cross, "Mesh partitioning: A multilevel balancing and refinement algorithm," *SIAM J. Sci. Comput.*, vol. 22, no. 1, pp. 63–80, 2000.

[35] J. Leskovec, "Stanford network analysis package (SNAP),"

[36] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, "Benchmarking for graph clustering and partitioning," in *Encyclopedia of Social Network Analysis and Mining*. Berlin, Germany: Springer-Verlag, 2014, pp. 73–82.

[37] U. of Milano's Laboratory of Web Algorithms, "Datasets,"

[38] T. Davis, "The University of Florida Sparse Matrix Collection,"

[39] M. von Looz, H. Meyerhenke, and R. Prutkin, "Generating random hyperbolic graphs in subquadratic time," in *Proc. 26th Int. Symp. Algorithms Comput.*, 2015, pp. 467–478.

[40] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. 13th Int. World Wide Web Conf.*, 2004, pp. 595–601.

[41] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Proc. of the 10th DIMACS Implementation Challenge*. Providence, RI, USA: AMS, 2012.

[42] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá, "Hyperbolic geometry of complex networks," *Phys. Rev. E*, vol. 82, no. 3, Sep. 2010, Art. no. 036106. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.82.036106

[43] A. J. Soper, C. Walshaw, and M. Cross, "A combined evolutionary search and multilevel optimisation approach to graph-partitioning," *J. Global Optimization*, vol. 29, no. 2, pp. 225–241, 2004.

Henning Meyerhenke received the diplom degree in computer science from Friedrich-Schiller-University Jena, Germany, in 2004 and the PhD (with highest distinction) degree in computer science from the University of Paderborn, Germany, in 2008. He is an assistant professor (*Juniorprofessor*) in the Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany, since October 2011. Previously, he was postdoc in the Georgia Institute of Technology. After his graduation, he was also research scientist with NEC Laboratories Europe and postdoc with the University of Paderborn. His main research interests include parallel algorithm engineering for massive data sets in three main application areas: combinatorial scientific computing, network analysis, and parallel combinatorial optimization. His work received the Best Algorithms Paper Award of the 22nd IEEE International Parallel and Distributed Processing Symposium and the Best Paper Award of the 2015 International Symposium on Foundations and Applications of Big Data Analytics.

Peter Sanders received the PhD degree in computer science from the Universität Karlsruhe, Germany, in 1996. After 7 years at the Max-Planck-Institute for Informatics, Saarbrücken, he returned to Karlsruhe as a full professor in 2004. He has more than 200 publications, mostly on algorithms for large data sets. This includes parallel algorithms (load balancing,...) memory hierarchies, graph algorithms (route planning, graph partitioning...), randomized algorithms, full text indices,... He is very active in promoting the methodology of algorithm engineering. He won a number of prices, perhaps most notably the DFG Leibniz Award 2012 which amounts to 2.5 million Euros of research money.

Christian Schulz received the master's and PhD (with highest distinction) degrees in mathematics and computer science from KIT. He is a postdoctoral researcher in the Karlsruhe Institute of Technology (KIT) and at the University of Vienna. He was the best student of the year 2010 of the faculty of informatics and received multiple awards for his dissertation. His research interests include graph partitioning and clustering, parallel algorithms and combinatorial optimization in the context of big data. His graph partitioning algorithms-KaHIP-have been able to improve or reproduce most of the benchmark entries in the Walshaw Benchmark and scored most of the points in the 10th DIMACS Implementation Challenge on Graph Partitioning and Clustering.