High-Quality Shared-Memory Graph Partitioning

Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz

Abstract—Partitioning graphs into blocks of roughly equal size such that few edges run between blocks is a frequently needed operation in processing graphs. Recently, size, variety, and structural complexity of these networks has grown dramatically. Unfortunately, previous approaches to parallel graph partitioning have problems in this context since they often show a negative trade-off between speed and quality. We present an approach to multi-level shared-memory parallel graph partitioning that produces balanced solutions, shows high speedups for a variety of large graphs and yields very good quality independently of the number of cores used. For example, in an extensive experimental study, at 79 cores, one of our closest competitors is faster but fails to meet the balance criterion in the majority of cases and another is mostly slower and incurs about 13 percent larger cut size. Important ingredients include parallel label propagation for both coarsening and refinement, parallel initial partitioning, a simple yet effective approach to parallel localized local search, and fast locality preserving hash tables.

Index Terms—Parallel graph partitioning, shared-memory parallelism, local search, label propagation

1 Introduction

Partitioning a graph into k blocks of similar size such that a minimum number of edges are cut is a fundamental problem with many applications. For example, it often arises when processing a single graph on k processors.

The graph partitioning problem is NP-hard and there is no approximation algorithm with a constant ratio factor for general graphs [1]. Thus, to solve the graph partitioning problem in practice, one needs to use heuristics. A very common approach to partition a graph is the multi-level graph partitioning (MGP) approach. The main idea is to contract the graph in the *coarsening* phase until it is small enough to be partitioned by more sophisticated but slower algorithms in the *initial partitioning* phase. Afterwards, in the *refinement* phase (also called uncoarsening/local search), the quality of the partition is improved on every level of the computed hierarchy using a local improvement algorithm.

There is a need for shared-memory parallel graph partitioning algorithms that efficiently utilize all cores of a machine. This is because providing a large number of cores has been the main way to use growing transistor budgets of microprocessors in recent years. Moreover, shared-memory parallel algorithms implemented without message-passing libraries (e.g. MPI) usually give better speedups and running times than their MPI-based counterparts. Shared-memory parallel graph partitioning algorithms can also be used as a component of a distributed graph partitioner, which distributes parts of a graph to nodes of a compute cluster and then employs a shared-memory parallel graph partitioning algorithm to partition the corresponding part of the graph on the node level.

Contribution: We present a high-quality shared-memory parallel multi-level graph partitioning algorithm that parallelizes all of the three MGP phases – coarsening, initial partitioning and refinement – using C + +17 multi-threading. Our approach uses a scalable parallel label propagation algorithm that is able to quickly shrink large complex networks during the coarsening phase. Our parallelization of localized local search [2] is able to obtain high-quality solutions and guarantees balanced partitions despite performing most of the work in mostly independent local searches of individual threads. Using *cache-aware hash tables*, we limit memory consumption and improve locality.

After presenting preliminaries and related work in Section 2, we explain details of the multi-level graph partitioning approach and the algorithms that we parallelize in Section 3. Section 4 presents our approach to the parallelization of the multi-level graph partitioning phases. More precisely, we present a parallelization of label propagation with size-constraints [3], as well as a parallelization of k-way multi-try local search [2]. Section 5 describes further optimizations. Extensive experiments are presented in Section 6. Our approach scales comparatively better than other parallel partitioners and has considerably higher quality which does not degrade with increasing number of processors.

2 Preliminaries

2.1 Basic Concepts

Let $G = (V = \{0, \ldots, n-1\}, E)$ be an undirected graph, where $n = |V|$ and $m = |E|$. We consider positive, real-valued edge and vertex weight functions ω and c extending them to sets, e.g., $\omega(M) := \sum_{x \in M} \omega(x)$. $N(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v. The degree of a vertex v is $d(v) := |N(v)|$. Δ is the maximum vertex degree. A vertex is a *boundary vertex* if it is incident to a vertex in a different block. We are looking for

- *Y. Akhremtsev is with Google, Zurich, Switzerland. E-mail: classboxmail@gmail.com.*
- *P. Sanders is with the Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany. E-mail: sanders@kit.edu.*
- *C. Schulz is with the Faculty of Computer Science, University of Vienna, Vienna 1010, Austria. E-mail: christian.schulz@univie.ac.at.*

Manuscript received 25 Oct. 2019; revised 9 May 2020; accepted 8 June 2020. Date of publication 11 June 2020; date of current version 24 June 2020. (Corresponding author: C. Schulz).
Recommended for acceptance by K. Madduri.
Digital Object Identifier no. 10.1109/TPDS.2020.3001645

disjoint *blocks* of vertices V_1,...,V_k that partition V; i.e., $V_1 \cup \ldots \cup V_k = V$. The *balancing constraint* demands that all blocks have weight $c(V_i) \leq (1+\epsilon)\lceil\frac{c(V)}{k}\rceil =: L_{\max}$ for some imbalance parameter ϵ. We call a block V_i *overloaded* if its weight exceeds $L_{\max}$. The objective is to minimize the total *cut* $\omega(E \cap \bigcup_{i<j} V_i \times V_j)$. We define the gain of a vertex as the maximum decrease in cut size when moving it to a different block. We denote the number of processing elements (PEs) as p.

A clustering is also a partition of the vertices. 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.

An abstract view of the partitioned graph is a *quotient graph*, in which vertices represent blocks and edges are induced by connectivity between blocks. The *weighted* version of the quotient graph has vertex weights that are set to the weight of the corresponding block and edge weights that are equal to the weight of the edges that run between the respective blocks. Our input graphs G have unit edge weights and vertex weights. However, even those will be translated into weighted problems in the course of the multi-level algorithm. In order to avoid a tedious notation, G will denote the current state of the graph before and after a (un)contraction in the multi-level scheme throughout this paper.

Atomic concurrent updates of memory cells are possible using the compare-and-swap operation CAS(x, y, z). If $x = y$ then this operation assigns $x \leftarrow z$ and returns $True$; otherwise it returns $False$.

We analyze algorithms using the concept of total *work* (the time needed by one processor) and *span*; i.e., the time needed using an unlimited number of processors [4].

2.2 Related Work

There has been intensive research on graph partitioning so that we refer the reader to [5], [6], [7], [8] for more details. Here, we focus on issues closely related to our main contributions. All general-purpose methods that are able to obtain good partitions for large real-world graphs are based on the multi-level principle. Well-known software packages based on this approach include Jostle [6], KaHIP [2], Metis [9] and Scotch [10].

Probably the fastest commonly-used distributed memory multi-level parallel code is the parallel version of Metis, ParMetis [11]. This parallelization has problems maintaining the balance of the blocks since at any particular time, it is difficult to say how many vertices are assigned to a particular block. In addition, ParMetis only uses very simple greedy local search algorithms that do not yield high-quality solutions. Mt – Metis by LaSalle and Karypis [12], [13] is a shared-memory parallel partitioner inspired by ParMetis. Mt – Metis uses a hill-climbing technique during refinement. The local search method is a simplification of k-way multi-try local search [2] in order to make it fast. The idea is to find a set of vertices (hill) whose move to another block is beneficial and then to move this set accordingly. However, it is possible that several PEs move the same vertex. To handle this, each vertex is assigned a PE, which can move it exclusively. Other PEs use a message queue to send a request to move this vertex.

PT-Scotch [10], the parallel version of Scotch, is based on recursive bipartitioning. This is more difficult to parallelize than direct 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 vertices close to the boundary of the current partitioning (band-refinement). KaPPa [14] 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 [15] is a multi-level diffusion-based algorithm that is targeted at small- to medium-scale parallelism with dozens of processors.

The label propagation clustering algorithm was initially proposed by Raghavan *et al.* [16]. A single round of simple label propagation can be interpreted as the randomized agglomerative clustering approach proposed by Catalyurek and Aykanat [17]. Moreover, the label propagation algorithm has been used to partition networks by Ugander and Backstrom [18]. The authors do not use a multi-level scheme and rely on a given or random partition which is improved by combining the unconstrained label propagation approach with linear programming. This approach does not yield high quality partitions.

Meyerhenke *et al.* [19] propose ParHIP, to partition large complex networks on distributed memory parallel machines. The partition problem is addressed by parallelizing and adapting the label propagation technique for graph coarsening and refinement. The resulting system is more scalable and achieves higher quality than the state-of-the-art systems like ParMetis or PT-Scotch. Wang *et al.* [20] introduce a multi-level partitioning algorithm based on label propagation without size-constraints. In this case, no strict size-constraint on the blocks is enforced.

Recently, Slota *et al.* [21] have used single-level label propagation for partitioning complex networks in their algorithm called PuLP as well. However, experiments on a wide range of graphs indicate that solution quality is significantly worse than ParHIP [19]. Later, Slota *et al.* [22] proposed XtraPulP, which enables PuLP scale to much larger instances. XtraPULP successfully partitioned a graph with 1.1 trillion edges.

Another related area are streaming graph partitioning algorithms [23], [24], [25], [26], [27]. Here the algorithms operate in a model in which vertices and their neighborhood arrive one at a time and the algorithm directly has to assign a block to the current vertex which can not be changed. All of these algorithms perform a single-pass (or in case of restreaming multiple-passes) over the stream of vertices. However, for example experiments conducted in the respective papers indicate that FENNEL [24] or Spinner [26] cut significantly more edges than Metis [9] while using more imbalance. Metis on the other hand computes significantly worse cuts than other recent high-quality multi-level schemes [28], [29].

3 Multi-Level Graph Partitioning

We now give an in-depth description of the three main phases of a multi-level graph partitioning algorithm: coarsening, initial partitioning and refinemen. In particular, we give a description of the sequential algorithms that we parallelize in the following sections. Our starting point here is the fast social configuration of KaHIP which uses label propagation for coarsening and as the only local search algorithm during refinement. For the development of the parallel algorithm, we add a k-way multi-try local search scheme that gives higher quality, and improve it to perform less work than the original sequential version. The original sequential implementations of these algorithms are contained in the KaHIP [2] graph partitioning framework A general principle is to randomize tie-breaking whenever possible. This diversifies the search and allows improved solutions by repeated tries.

3.1 Coarsening

To create a new level of a graph hierarchy, the rationale here is to compute a clustering with clusters that are bounded in size and then to *contract* each cluster into a supervertex.[1] Contracting a clustering works by replacing each cluster with a single vertex. The weight of this new vertex (or supervertex) is set to the sum of the weight of all vertices in the original cluster. There is an edge between two vertices u and v in the contracted graph if the two corresponding clusters in the clustering are adjacent to each other in G; i.e., if the cluster of u and the cluster of 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 cluster A and cluster B of the clustering. The hierarchy created in this recursive manner is then used by the partitioner. Due to the way the contraction is defined, it is ensured that a partition of the coarse graph corresponds to a partition of the finer graph with the same cut and balance. We now describe the clustering and the matching algorithms that we parallelize.

Clustering: We denote the set of all clusters as C and the cluster ID of a vertex v as $C[v]$. There are a variety of clustering algorithms. We use the label propagation algorithm by Meyerhenke *et al.* [3] that creates a clustering fulfilling a size-constraint.

The size constrained label propagation algorithm works in iterations; i.e., the algorithm is repeated ℓ times, where ℓ is a tuning parameter. Initially, each vertex is in its own cluster ($C[v] = v$) and all vertices are put into a queue Q in increasing order of their degrees. During each iteration, the algorithm iterates over all vertices in Q. A neighboring cluster $\mathcal{C}$ of a vertex v is called *eligible* if $\mathcal{C}$ will not become overloaded once v is moved to $\mathcal{C}$. When a vertex v is visited, it is *moved* to the eligible cluster that has the strongest connection to v; i.e., it is moved to the eligible cluster $\mathcal{C}$ that maximizes $\omega(\{(v,u) \mid u \in N(v) \cap \mathcal{C}\})$. If a vertex changes its cluster ID then all its neighbors are added to a queue Q' for the next iteration. At the end of an iteration, Q and Q' are swapped, and the algorithm proceeds with the next iteration. It stops after a preset number of iterations or when Q is empty. The sequential running time of one iteration of the algorithm is $\mathcal{O}(m + n)$.

The *contraction* algorithm takes a graph $G = (V, E)$ as well as a clustering C and constructs a coarse graph $G' = (V', E')$. The contraction process consists of three phases: the remapping of cluster IDs to a consecutive set of IDs, edge weight accumulation, and the construction of the coarse graph. The remapping of cluster IDs assigns new IDs in the range $[0, |V'| - 1]$ to the clusters where $|V'|$ is the number of clusters in the given clustering. We do this by calculating a prefix sum on an array that contains ones in the positions equal to the current cluster IDs. This phase runs in $\mathcal{O}(n)$ time. The edge weight accumulation step calculates weights of edges in E' using hashing. More precisely, for each cut edge $(v, u) \in E$ we insert a pair $(C[v], C[u])$ such that $C[v] \neq C[u]$ into a hash table and accumulate weights for the pair if it is already contained in the table. Due to hashing cut edges, the expected running time of this phase is $\mathcal{O}(|E'| + m)$. To construct the coarse graph we iterate over all edges E' contained in the hash table. This takes time $\mathcal{O}(|V'| + |E'|)$. Hence, the total expected running time to compute the coarse graph is $\mathcal{O}(m + n + |E'|)$ when run sequentially.

3.2 Initial Partitioning

We adopt the algorithm from KaHIP [2]: After coarsening, the coarsest level of the hierarchy is partitioned into k blocks using a recursive bisection algorithm [30]. More precisely, it is partitioned into two blocks and then the subgraphs induced by these two blocks are recursively partitioned into $\lceil \frac{k}{2} \rceil$ and $\lfloor \frac{k}{2} \rfloor$ blocks each. Subsequently, this partition is improved using local search and flow techniques. To get a better solution, the coarsest graph is partitioned into k blocks I times and the best solution is returned.

3.3 Refinement

After initial partitioning, a local search algorithm is applied to improve the cut of the partition. When local search has finished, the partition is transferred to the next finer graph in the hierarchy (uncoarsening); i.e., a vertex in the finer graph is assigned the block of its coarse representative. This process of subsequent local search and uncoarsening is repeated for each level of the hierarchy.

There are a variety of local search algorithms: size-constrained label propagation, Fiduccia-Mattheyses k-way local search [31], max-flow min-cut based local search [2], k-way multi-try local search [2] .... Sequential versions of KaHIP use combinations of those. Since k-way local search is P-complete [32], our algorithm uses size-constrained label propagation in combination with k-way multi-try local search. More precisely, the size-constrained label propagation algorithm can be used as a fast local search algorithm if one starts from a partition of the graph instead of a clustering and uses the size-constraint of the partitioning problem. On the other hand, k-way multi-try local search is able to find high quality solutions. Overall, this combination allows us to achieve a parallelization with good solution quality and good parallelism.

We now describe improved localized multi-try k-way local search (LMLS) which differs from the original version of the localized multi-try a more sophisticated for global iterations. In contrast to previous k-way local search methods LMLS is not initialized with *all* boundary vertices; that is, not all boundary vertices are eligible for movement at the beginning. Instead, the method is repeatedly initialized

[1] Many graph partitioners only contract clusters of two vertices (matchings). Our system also has that option but this is rarely advantageous for us.

with a *single* boundary vertex. This enables more diversification and has a better chance of finding nontrivial improvements that begin with negative gain moves [2].

The algorithm is organized in a nested loop of global and local iterations. A global iteration works as follows. First, the algorithm constructs a hash table that contains all boundary vertices. We use a hash table since after each local iteration the set of boundary vertices changes and must be updated. Next, instead of putting *all* boundary vertices directly into a priority queue, boundary vertices under consideration are put into a todo list T. Initially, all vertices are unmarked. Afterwards, the algorithm repeatedly chooses and removes a random vertex $v \in T$. If the vertex is unmarked, it starts to perform k-way local search around v, marking every vertex that is moved during this search. More precisely, the algorithm inserts v and $N(v)$ into a priority queue using gain values as keys and marks them. Next, it extracts a vertex with a maximum key from the priority queue and performs the corresponding move updating the hash table with boundary vertices. If a neighbor of the vertex is unmarked then it is marked and inserted in the priority queue. If a neighbor of the vertex is already in the priority queue then its key (gain) is updated. Note that not every move can be performed due to the size-constraint on the blocks. The algorithm stops when the adaptive stopping rule by Osipov and Sanders [33] decides to stop or when the priority queue is empty. More precisely, if the overall gain is negative then the stopping rule estimates the probability that the overall gain will become positive again and signals to stop if this is unlikely. In the end, the best partition that has been seen during the process is reconstructed. In one local iteration, this is repeated until the todo list is empty.

After a local iteration, the algorithm reinserts moved vertices into the todo list in random order. Afterwards, it applies the quantile-based stopping rule described in Section 3.3.1 to decide whether to proceed to the next local iteration or not. This allows to further decrease the cut size without significant impact to the running time. When the quantile-based stopping rule stops local iterations, the current global iteration finishes. Next, the algorithm uses the quantile-based stopping to decide whether to stop or not. Note that when another (global) iteration is started, the algorithm initializes the todo list with all boundary vertices in the hash table. This nested loop of local and global iterations is an improvement over the original LMLS search from [2] since it allows more control over the trade-off between running time and quality of the algorithm.

The running time of one local iteration is $\mathcal{O}(n + \sum_{v \in V} d(v)^2)$. Because each vertex can be moved only once during a local iteration and we update the gains of its neighbors using a bucket heap. Since we update the gain of a vertex at most $d(v)$ times, the $d(v)^2$ term is the total cost to update the gain of a vertex v. Note, that this is an upper bound for the worst case, usually local search stops significantly earlier due the stopping rule or an empty priority queue.

3.3.1 Quantile-Based Stopping Rule

We developed a heuristic stopping rule that considers work-to-gain ratios of global (local) iterations to decide whether to perform a new global (local) iteration – see the thesis [34] for details. Here work is the number of accesses to partition IDs of vertices during an iteration and gain is the reduction of the cut size performed during the iteration. This approach is based on our empirical observation that work-to-gain ratios have a distribution similar to a log-normal distribution. During refinement, the parameters of this distribution are estimated. When the current ratio exceeds a ρ-quantile of this distribution or the gain is zero, search stops. Here, $1/2 < \rho < 1$ is a tuning parameter expressing how unlikely it is that continued search will find improvements worth the invested effort.

4 Parallel Multi-Level Partitioning

Profiling the sequential algorithm shows that each of the components of the multi-level scheme has a significant contribution to the overall algorithm. Hence, we now describe the parallelization of each phase of the multi-level algorithm described above. The section is organized along the phases of the multi-level scheme: first we show how to parallelize coarsening, then initial partitioning and finally refinement. Our general approach is to avoid bottlenecks as well as performing independent work as much as possible.

4.1 Coarsening

In this section, we present the parallel version of the size-constrained label propagation algorithm to build a clustering and the parallel contraction algorithm.

Parallel Size-Constrained Label Propagation: To parallelize the size-constrained label propagation algorithm, we adapt a clustering technique by Staudt and Meyerhenke [35] to coarsening. Initially, we sort the vertices by increasing degree using the fast parallel sorting algorithm by Axtmann *et al.* [36]. We then form work packets representing a roughly equal amount of work and insert them into a TBB (threading building blocks) concurrent queue Q [37]. Note that we also tried the work-stealing approach from [38] but it showed worse running times. Our constraint is that a packet contains vertices with a total number of neighbors at most B. We set $B = \max(1\,000, \sqrt{m})$ in our experiments – the $1\,000$ limits contention for small graphs and the term $\sqrt{m}$ further reduces contention for large graphs. Additionally, we have an empty queue Q' that stores packets of vertices for the next iteration. During an iteration, each PE checks if the queue Q is not empty, and if so it extracts a packet of active vertices from the queue. A PE then chooses a new cluster for each vertex in the currently processed packet. A vertex is then moved if the cluster size is still feasible to take on the weight of the vertex. Cluster sizes are updated atomically using a compare-and-swap (CAS) instruction. This is important to guarantee that the size constraint is not violated. Neighbors of moved vertices are inserted into a packet for the next iteration. If the sum of vertex degrees in that packet exceeds the work bound B, then this packet is inserted into queue Q' and a new packet is created for subsequent vertices. When the queue Q is empty, the main PE exchanges Q and Q' and we proceed with the next iteration. One iteration of the algorithm can be done with $\mathcal{O}(n + m + p^2 + p\Delta)$ work and in $\mathcal{O}((n + m)/p + p + \Delta)$ parallel time.

Parallel Contraction: The parallel contraction algorithm works as follows. First, we remap the cluster IDs using the parallel prefix sum algorithm by Singler *et al.* [38]. Edge weights are accumulated by iterating over the edges of the original graph in parallel. We use the concurrent hash table of Maier and Sanders [39] initializing it with a capacity of $\min(\text{avg\_deg} \cdot |V'|, |E|/10)$. Here $\text{avg\_deg} = 2|E|/|V|$ is the average degree of G since we hope that the average degree of G' remains the same. Note that this is a rough estimation of $|E|$ and in case it underestimates the real value the concurrent hash table is able to grow. To parallelize the last phase, we first calculate degrees of coarse vertices by iterating over the concurrent hash table in parallel. Then we use the parallel prefix sum algorithm to compute offsets of coarse vertices in the array of coarse edges. Finally, we construct the array of coarse edges by iterating over the concurrent hash table in parallel one more time.

4.2 Initial Partitioning

To improve the quality of the resulting partitioning of the coarsest graph $G' = (V', E')$, we partition it into k blocks $\max(p, I)$ times instead of I times. We perform each partitioning step independently in parallel using different random seeds. To do so, each PE creates a copy of the coarsest graph and runs KaHIP sequentially on it. Assume that one partitioning can be done in time T. Then $\max(p, I)$ partitions can be built with $\mathcal{O}(\max(p, I) \cdot T + p \cdot (|E'| + |V'|))$ work and $\mathcal{O}(\frac{\max(p,I) \cdot T}{p} + |E'| + |V'|)$ span, where the additional terms $|V'|$ and $|E'|$ account for the time each PE copies the coarsest graph.

4.3 Refinement

Our parallel algorithm first uses size-constraint parallel label propagation to improve the current partition and afterwards applies our parallel LMLS. The rationale behind this combination is that label propagation is fast and easy to parallelize and will do all the easy improvements. Subsequent LMLS will then invest considerable work to find a few nontrivial improvements. In this combination, only few nodes actually need be moved globally which makes it easier to parallelize LMLS scalably. When using the label propagation algorithm to improve a partition, we set the upper bound U to the size-constraint of the partitioning problem $L_{\max}$.

Parallel LMLS works in a nested loop of local and global iterations as in the sequential version. Initialization of a global iteration copies all boundary vertices to a parallel todo list T. T is split into local buckets – one for each PE. Since the sets of boundary vertices are stored in hash tables, the copying operation assigns them to buckets in random order. During a local iteration, each PE extracts vertices v from the parallel todo list T. This is scalable because each PE can mostly access its local bucket. Afterwards, it performs *local* moves around v (PerformMoves); that is, global block IDs and the sizes of the blocks remain *unchanged*. When the todo list T is empty, the algorithm applies the best found sequences of moves to the global data structures (ApplyMoves). In the paragraphs that follow, we describe how to perform local moves in PerformMoves and then how to update the global data structures in ApplyMoves.

*Performing Moves (*PerformMoves*):* Starting from a single boundary vertex, each PE moves vertices to find a sequence of moves that decreases the cut – by storing vertices in a PQ with gain as priority and always moving the vertex having the largest gain). However, all moves are local; that is, they do not affect the current global partition – moves are stored in the local memory of the PE performing them. To perform a move, a PE chooses a vertex with maximum gain and marks it so that other PEs cannot move it. Then, we update the sizes of the affected blocks and save the move. During the course of the algorithm, we store the sequence of moves yielding the best cut. We stop if there are no moves to perform or the adaptive stopping rule signals the algorithm to stop. When a PE finished, the sequences of moves yielding the largest decrease in the edge cut is returned.

Implementation Details of PerformMoves*:* In order to improve scalability, only the array for marking moved vertices is global. Note that within a local iteration, only bits in this array are set (using CAS) and they are never unset. Hence, the marking operation can be seen as priority update operation (see Shun *et al.* [40]) and thus causes only little contention. The algorithm keeps a local array of block sizes, a local priority queue, and a local hash table storing changed block IDs of vertices. Note that since the local hash table is small, it often fits into cache which is crucial for parallelization due to memory bandwidth limits. When the call of PerformMoves finishes and the thread executing it notices that the todo list T is empty, it sets a global variable to signal the other threads to finish the current call of the function PerformMoves.

Let each PE process a set of edges $\mathcal{E}$ and a set of vertices $\mathcal{V}$. Since each vertex can be moved only by one PE and moving a vertex requires the gain computation of its neighbors, the span of the function PerformMoves is $\mathcal{O}(\sum_{v \in \mathcal{V}} \sum_{u \in N(v)} d(u) + |\mathcal{V}|) = \mathcal{O}(\sum_{v \in \mathcal{V}} d^2(v) + |\mathcal{V}|)$ since the gain of a vertex v can be updated at most $d(v)$ times. Note that this is a pessimistic bound and it is possible to implement this function with $\mathcal{O}(|\mathcal{E}| \log \Delta + |\mathcal{V}|)$ span using a priority queue. In our experiments, we use the implementation with the former running time since it requires less memory and the worst case – the gain of a vertex v is updated $d(v)$ times – is quite unlikely.

*Applying Moves (*ApplyMoves*):* Let $M_i = \{B_{i1}, \ldots\}$ denote the set of sequences of moves performed by PE i, where B_{ij} is a set of moves performed by the j-th call of PerformMoves. We apply moves sequentially in the following order $M_1, M_2, \ldots, M_p$. We can not apply the moves directly in parallel since a move done by one PE can affect a move done by another PE. More precisely, assume that we want to move a vertex $v \in B_{ij}$ but we have already moved its neighbor w on a different PE. Since the PE only knows local changes, it calculates the gain to move v (in PerformMoves) according to the old block ID of w. If we then apply the rest of the moves in B_{ij} it may even increase the cut. To prevent this, we recalculate the gain of each move in a given sequence and remember the best cut. If there are no affected moves, we apply all moves from the sequence. Otherwise we apply only the part of the moves that gives the best cut with respect to the correct gain values. Finally, we insert all moved vertices into the todo list T. Let M be the set of all moved vertices during this procedure. The overall running time is then given by $\mathcal{O}(\sum_{v \in M} d(v))$. Note that our initial partitioning algorithm generates balanced solutions. Since moves are applied sequentially our parallel local search algorithm maintains balanced solutions; i.e. the balance constraint of our solution is never violated.

4.4 Differences to $\mathsf{Mt-Metis}$

We now discuss the differences between our algorithm and $\mathsf{Mt-Metis}$. In the coarsening phase, our approach uses general cluster contraction while Metis always contracts matchings. Cluster contraction is better suited for networks that have hierarchical cluster structure. For example, in networks with star-like structures, a matching-based coarsener can only match a single edge per level. Moreover, it may contract "wrong" edges such as bridges [19]. Initial partitioning is similar in both algorithms except that different sequential partitioners are used for the base case (KaHIP and Metis). In terms of local search, unlike $\mathsf{Mt-Metis}$, our approach guarantees that the updated partition is balanced if the input partition is balanced and that the cut can only decrease or stay the same. The hill-climbing technique, however, may increase the cut of the input partition or may compute an imbalanced partition even if the input partition is balanced. Our algorithm has these guarantees since each PE performs moves of vertices locally in parallel. When all PEs finish, one PE globally applies the best sequences of local moves computed by all PEs. Usually, the number of applied moves is significantly smaller than the number of the local moves performed by all PEs, especially on large graphs. Thus, the main work is still made in parallel. Additionally, we introduce a cache-aware hash table in the following section that we use to store local changes of block IDs made by each PE. This hash table is more compact than an array and takes the locality of data into account.

5 Further Optimization

In this section, we describe further optimization techniques that we use to achieve better scalability and efficiency. More precisely, we use cache-aligned arrays to mitigate the problem of false-sharing and the TBB scalable allocator [37] for concurrent memory allocations. We pin threads to cores to avoid rescheduling overheads. Additionally, we use a cache-aware hash table which we describe now. In contrast to usual hash tables, this hash table allows us to exploit locality of data and hence to reduce the overall running time of the algorithm.

5.1 Cache-Aware Hash Table

The main goal here is to improve the performance of our algorithm on large graphs. For large graphs, the gain computation in the LMLS routine takes most of the time. Recall, that computing the gain of a vertex requires a local hash table. Hence, using a cache-aware technique reduces the overall running time. A cache-aware hash table combines both properties of an array and a hash table. It tries to store data with similar integer keys within the same cache line, thus reducing the cost of subsequent accesses to these keys. On the other hand, it still consumes less memory than an array which is crucial for the hash table to fit into caches.

We implement a cache-aware hash table using the linear probing technique and tabulation hashing [41]. Linear probing typically outperforms other collision resolution techniques in practice and the computation of the tabular hash function can be done with a small overhead. The tabular hash function works as follows. Let $x = x_1 \dots x_k$ be a key to be hashed, where x_i are t bits of the binary representation of x. Let $T_i, i \in [1, k]$ be tables of size 2^t, where each element is a random 32-bit integer. Using $\oplus$ as exclusive-or operation, the tabular hash function is then defined as $h(x) = T_1[x_1] \oplus \dots \oplus T_k[x_k]$.

Exploiting Locality of Data: As our experiments show, the distribution of keys that we access during the computation of the gains is not uniform. Instead, it is likely that the time between accesses to two consecutive keys is small. On typical systems currently used, the size of a cache line is 64 bytes (16 elements with 4 bytes each). Now suppose our algorithm accesses 16 consecutive vertices one after another. If we would use an array storing the block IDs of all vertices instead of a hash table, we can access all block IDs of the vertices with only one cache miss. A hash table on the other hand does not give any locality guarantees. On the contrary, it is very probable that consecutive keys are hashed to completely different parts of the hash table. However, due to memory constraints we can not use an array to store block IDs for each PE in the $\mathsf{PerformMoves}$ procedure. However, even if the arrays fit into memory this could still be slower compared to hash tables that fit into cache.

For this, we modify the tabular hash function from above [42]. More precisely, let $x = x_1 \dots x_{k-1}x_k$, where x_k are the t' least significant bits of x and $x_1, \dots, x_{k-1}$ are t bits each. Then we compute the tabular hash function as $h(x) = T_1[x_1] \oplus \dots \oplus T_{k-1}[x_{k-1}] \oplus x_k$. This guarantees that if two keys x and x' differ only in the first t' bits and, hence, $|x - x'| < 2^{t'}$ then $|h(x) - h(x')| < 2^{t'}$. Thus, if $t' = \mathcal{O}(\log c)$, where c is the size of a cache line, then x and x' are in the same cache line when accessed. This hash function introduces at most $2^{t'}$ additional collisions since if we do not consider t' least significant bits of a key then at most $2^{t'}$ keys have the same remaining bits. In our experiments, we choose $k = 3, t' = 5, t = 10$.

6 Experiments

6.1 Methodology

We run our experiments on two different machines. Machine A has 80 cores (4× Intel Xeon Gold 6138) and 768 GB RAM. Machine B has 32 cores (2× Intel Xeon E5-2683v2) and 512 GB RAM. In our experiments we leave one core unused since this yields more stable performance.

We implemented our algorithm $\mathsf{Mt-KaHIP}$ (Multi-threaded Karlsruhe High Quality Partitioning) within the KaHIP [2] framework using $\mathsf{C++}$ and the $\mathsf{C++}$17 multi-threading library. Our framework is available online [43]. All binaries are built using $\mathsf{g++-5.2.0}$ with the $\mathsf{-O3}$ flag and 64-bit index data types.

We compare ourselves to $\mathsf{Mt-Metis0.6.0}$ using the default configuration with hill-climbing being enabled ($\mathsf{Mt-Metis}$) as well as sequential $\mathsf{KaHIP2.0}$ using the $\mathsf{fastsocial}$ configuration (KaHIP) and $\mathsf{ParHIP2.0}$ [19] using the $\mathsf{fastsocial}$ configuration (ParHIP). According to LaSalle and Karypis [13] $\mathsf{Mt-Metis}$ has better speedups and running times compared to $\mathsf{ParMetis}$ and $\mathsf{Pt-Scotch}$. At the same time, it has similar quality of the partition. Hence, we do not perform experiments with $\mathsf{ParMetis}$ and $\mathsf{Pt-Scotch}$.

Our default value of allowed imbalance is 3 percent ($\epsilon = 0.03$) – this is the most frequently used value in previous studies. We call a solution imbalanced if at least one block exceeds this amount. By default, we perform ten repetitions for every algorithm using different random seeds for initialization and report the arithmetic average of computed cut size and running time on a per instance (graph and number of blocks k) basis. When further averaging over multiple instances, we use the *geometric mean* for quality in order to give every instance a comparable influence on the final score. For the number of blocks, we consider $k \in \{16, 64, 256, 1024\}$.

We use performance plots to present the quality comparisons: For each instance, and each considered code, we plot one point in such a plot. Let c denote the "cost" of the instance for the considered code and let c^* denote the best observed cost for the same instance over all codes. We then plot $1 - c^*/c$. These values are then sorted. Thus, the result of the best algorithm is in the bottom of the plot. Assuming for simplicity that all considered codes produce only balanced partitions, the costs are the observed cut values averaged over all performed repetitions. Since some codes sometimes compute highly imbalanced solutions, we also have to take imbalance into account. For this purpose we introduce a notion of *penalized* cut. Specifically, we multiply average cut values for solutions that *violate* the balance constraint with a penalty factor $\frac{1+\epsilon'}{1+\epsilon}$ where ϵ' is the average imbalance parameter observed for that code.

In order to decide whether one algorithm performs significantly better than another one, we use the Wilcoxon signed-rank test [52] with a 5 percent p-value.

Algorithm Configuration: Any multi-level algorithm has a considerable number of choices between algorithmic components and tuning parameters. We adopt parameters from the coarsening and initial partitioning phases of KaHIP. The Mt – KaHIP configuration uses 10 and 25 label propagation iterations during coarsening and refinement (as those are the values used in KaHIP using the fastsocial configuration), respectively, partitions a coarse graph $\max(p, 4)$ times in initial partitioning. The quantile-based stopping rule for local search described in Section 3.3.1 uses the parameter $\rho = 0.9$. Additionally, there are at most 3 global iterations.

Instances: We evaluate our algorithms on a number of large graphs. These graphs are collected from [44], [47], [48], [49], [51], [53]. Table 1 summarizes the main properties of the 38 graphs. 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). Note that Mt – Metis and ParHIP were not able to partition all instances. Therefore, we calculate mean performance based on subsets of instances that were partitioned by all codes. The detailed description of these subsets can be found in the thesis of Akhremtsev [34].

The rhg graph is a complex network generated with NetworKit [49] according to the *random hyperbolic graph* model [54]. In this model, vertices are represented as points in the hyperbolic plane; vertices 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 vertices where vertices represent random points in the (euclidean) unit square and edges connect vertices 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. del X is a Delaunay triangulation of 2^X random points in the unit square. The graph er-fact1.5-scale23 is generated using the Erdös-Rényi $G(n, p)$ model with $p = 1.5 \ln n/n$.

TABLE 1
Benchmark Graphs

Graph	n	m	Type	Reference
amazon	≈0.4M	≈2.3M	C	[44]
youtube	≈1.1M	≈3.0M	C	[44]
amazon-2008	≈0.7M	≈3.5M	C	[45]
ba_2_22	≈4.2M (2^{22})	≈8.4M	C	[46]
in-2004	≈1.4M	≈13.6M	C	[45]
eu-2005	≈0.9M	≈16.1M	C	[45]
er_2_22_2_23	≈4.2M (2^{22})	≈16.3 (2^{23})	O	[46]
packing	≈2.1M	≈17.5M	M	[47]
hugebubbles-00	≈18.3M	≈27.5M	M	[47]
rhg_2_23	≈8.4M (2^{23})	≈32.1M	C	[46]
com-LiveJournal	≈4M	≈34.7M	C	[44]
channel	≈4.8M	≈42.7M	M	[47]
cage15	≈5.2M	≈47.0M	M	[47]
ljournal-2008	≈5.4M	≈49.5M	C	[45]
europe.osm	≈50.9M	≈54.1M	O	[48]
enwiki-2013	≈4.2M	≈91.9M	C	[45]
er-fact1.5-scale23	≈8.4M	≈100.3M	O	[48]
hollywood-2011	≈2.2M	≈114.5M	C	[45]
com-Orkut	≈3.1M	≈117.2M	C	[44]
enwiki-2018	≈5.6M	≈117.2M	C	[45]
indochina-2004	≈7.4M	≈151.0M	C	[45]
rhg	≈10.0M	≈199.6M	C	[49]
del_2_26	≈67.1M (2^{26})	≈201.3M	M	[50]
uk-2002	≈18.5M	≈261.8M	C	[45]
del_2_27	≈134.2M (2^{27})	≈303.2M	M	[46]
nlpkkt240	≈28.0M	≈373.2M	M	[51]
rgg_2_26_3d	≈67.1M (2^{26})	≈379.6M	M	[46]
del_2_26_3d	≈67.1M	≈521.3M	M	[46]
arabic-2005	≈22.7M	≈553.9M	C	[45]
rgg_2_26	≈67.1M (2^{26})	≈574.6M	M	[46]
uk-2005	≈39.5M	≈783.0M	C	[45]
rgg_2_27_3d	≈134.2M (2^{27})	≈787.7M	M	[46]
webbase-2001	≈118.1M	≈854.8M	C	[45]
it-2004	≈41.3M	≈1.0G	C	[45]
del_2_27_3d	≈134.2M (2^{27})	≈1.0G	M	[46]
rgg_2_27	≈134.2M (2^{27})	≈1.2G	M	[46]
sk-2005	≈50.6M	≈1.8G	C	[45]
uk-2007	≈106M	≈3.3G	C	[45]

C = complex networks, M = mesh type networks, and O = other networks (sorted by number of edges).

6.2 Quality Comparison

In this section, we compare our algorithm against other state-of-the-art algorithms in terms of quality. The performance plot in Fig. 1 shows the results of our experiments performed on Machine A for all of our benchmark graphs shown in Table 1. Overall, we analyze 152 instances partitioned by Mt – KaHIP, 151 instances partitioned by ParHIP, 130 instances partitioned by Mt – Metis, and 139 instances partitioned by KaHIP.

Our algorithm dominates all other algorithms in this performance plot. Mt – KaHIP usually computes the overall best solutions or is only a few percent off. Both ParHIP and

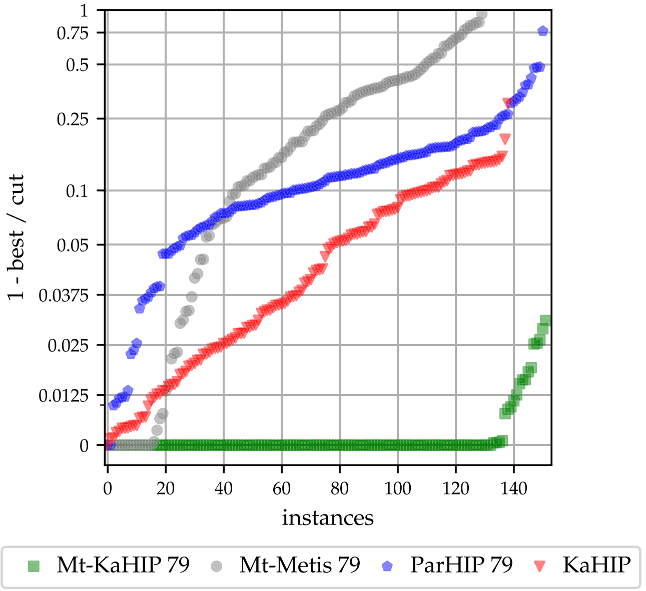

Fig. 1. Performance plot for the cut size. The number behind the algorithm name denotes the number of threads used. On the y-axis we use linear scaling for $[0, 0.05]$ and logarithmic scaling for $(0.05, 1]$.

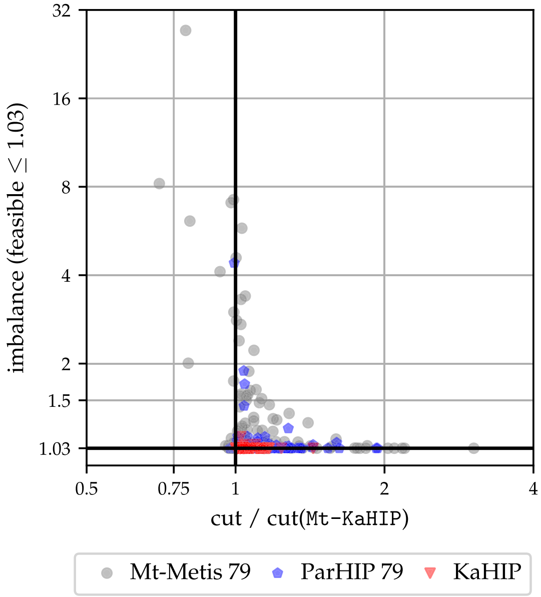

Fig. 2. Scatter plot showing imbalance on the x-axis and comparing the cut for competing codes with that of Mt − KaHIP79.

Mt − Metis have cost values that are more then 10 percent off most of the time. Mt − Metis is even more than 50 percent off for a significant fraction of instances. Closer inspection reveals that these bad cases are due to severely imbalanced solutions computed by the competing codes. To illustrate that, the scatter plot in Fig. 2 compares imbalance and cut separately.

The overall solution quality of Mt − KaHIP does not heavily depend on the number of PEs used. Indeed, more PEs give slightly higher partitioning quality since more initial partitioning attempts are done in parallel. This is an important difference to most other parallel graph partitioners – previously, parallelization was always associated with some loss in quality.

The original fast social configuration of KaHIP as well as ParHIP produce worse quality than Mt − KaHIP. This is due to the high quality local search scheme that we use; i.e., parallel LMLS significantly improves solution quality. Mt − Metis with $p = 79$ has worse quality than our algorithm on almost all instances. The exceptions are three networks (all of type mesh). For Mt − Metis this is expected since it is considerably faster than our algorithm. However, Mt − Metis also suffers from deteriorating quality and many imbalanced partitions as the number of PEs goes up. This is mostly the case for complex networks. This can also be seen from the geometric means of the cut sizes over all instances, including the imbalanced solutions.

We summarize the quality advantage of our code by presenting geometric mean cut sizes for instances which were partitioned by all frameworks in Table 2. We see that Mt − KaHIP has a quality advantage over all other systems. On the first glance, the advantage over Mt − Metis decreases for large k. However, closer inspection shows that for large k Mt − Metis produces highly imbalanced solutions. These could be viewed either as infeasible or one could make a kind of bicriteria comparison of the two systems. It looks like either way Mt − KaHIP comes out as more robust regardless of the choice of k. Still, all systems produce at least slightly imbalanced solutions for large k. For (Mt) − KaHIP, this is due to (small) problems in the initial partitioner which seems a topic for future work. Mt − Metis and, to a lesser degree, ParHIP make approximations in the refinement phase that can inherently lead to highly imbalanced solutions. Furthermore, significance tests indicate that the quality advantage of our solver over the other solvers is statistically significant.

6.2.1 *Effectiveness Tests*

We now compare the effectiveness of our algorithm Mt − KaHIP against other codes using a single PE and 79

TABLE 2
Overall Quality Parameters of Instances Partitioned by all Codes (all But KaHIP With $p = 79$ Cores)

k	Number of instances	Mt-KaHIP	Mt-Metis	ParHIP	KaHIP	Mt-KaHIP	Mt-Metis	ParHIP	KaHIP
		Geom. mean penalized cut	Penalized cut difference			Geom. mean imb. (%)	Imb (%). difference		
All	126	$2.724 \cdot 10^6$	$+46.2$ %	$+16.0$ %	$+6.6$ %	3.1 %	$+8.3$ %	$+0.6$ %	$+0.0$ %
16	35	$1.175 \cdot 10^6$	$+52.0$ %	$+16.8$ %	$+8.2$ %	3.0 %	$+5.1$ %	$+0.4$ %	$+0.0$ %
64	33	$2.374 \cdot 10^6$	$+44.0$ %	$+15.5$ %	$+7.3$ %	3.0 %	$+6.8$ %	$+0.5$ %	$+0.0$ %
256	29	$3.718 \cdot 10^6$	$+37.3$ %	$+14.1$ %	$+5.6$ %	3.1 %	$+8.4$ %	$+0.3$ %	$+0.0$ %
1024	29	$6.442 \cdot 10^6$	$+51.3$ %	$+17.6$ %	$+4.8$ %	3.2 %	$+17.5$ %	$+1.6$ %	$+0.1$ %

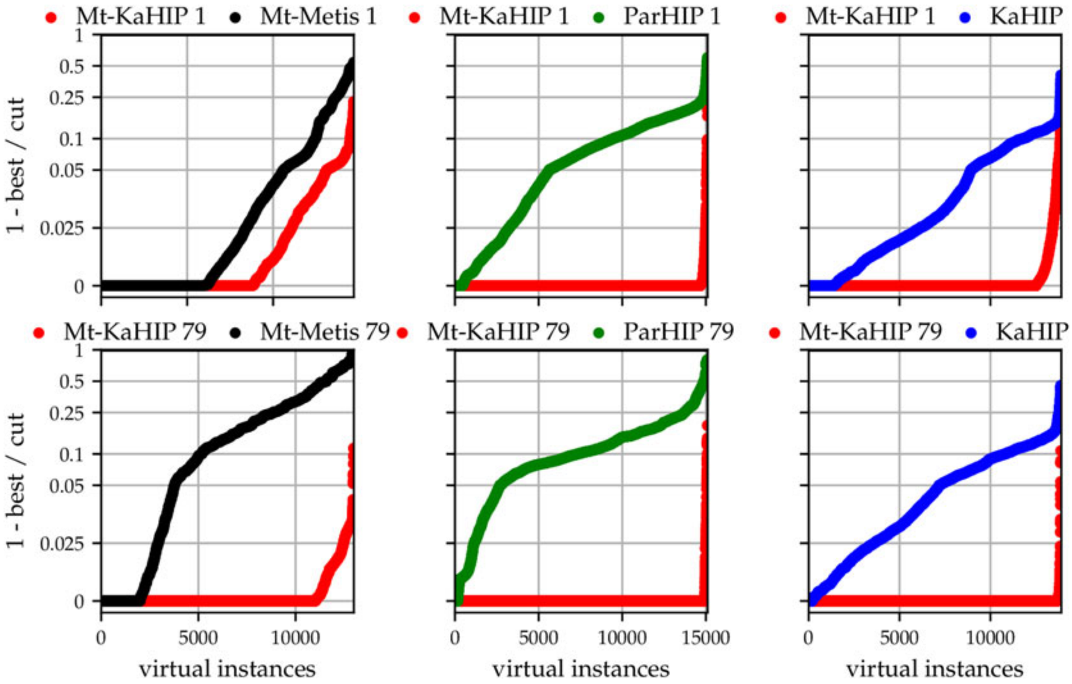

Fig. 3. Effectiveness tests for different codes. The number behind the algorithm name denotes the number of used threads.

PEs of Machine A. The idea is to give the faster algorithm the same amount of time as the slower algorithm for additional repetitions that can lead to improved solutions.[2] We have improved an approach used in [2] to extract more information out of a moderate number of measurements. Suppose we want to compare a repetitions of algorithm A and b repetitions of algorithm B on instance I. We generate a *virtual instance* as follows: We first sample one of the repetitions of both algorithms. Let t^1_A and t^1_B refer to the observed running times. Wlog assume $t^1_A \geq t^1_B$. Now we sample (without replacement) additional repetitions of algorithm B until the total running time accumulated for algorithm B exceeds t^1_A. Let t^ℓ_B denote the running time of the last sample. We accept the last sample of B with probability $(t^1_A - \sum_{1<i<\ell} t^i_B)/t^\ell_B$.

Theorem 1. *The expected total running time of accepted samples for B is the same as the time for the single repetition of A.*

Proof. Let $t = \sum_{1<i<\ell} t^i_B$. Consider a random variable T that is the total time of sampled repetitions. With probability $p = (t^1_A - t)/t^\ell_B$, we accept the ℓ-th sample and with probability $1-p$ we decline it. Then $\mathbf{E}[T] = p \cdot (t + t^\ell_B) + (1-p) \cdot t = (t^1_A - t)/t^\ell_B \cdot (t + t^\ell_B) + (1 - (t^1_A - t)/t^\ell_B) \cdot t = t^1_A$. □

The quality assumed for A in this virtual instance is the quality of the only run of algorithm A. The quality for B is the best quality observed for an accepted sample for B.

For our effectiveness evaluation, we used 100 virtual instances for each pair of graph and k derived from 10 repetitions of each algorithm. Fig. 3 presents the resulting performance plots.

Table 3 summarizes the results of effectiveness tests. Note that Mt − KaHIP always has smaller geometric mean cut size. Furthermore, Mt − KaHIP has only around 2.1 percent of imbalanced virtual instances in all effectiveness tests. We conclude that Mt − KaHIP considerably outperforms the other codes in terms of quality and that the gap increases when parallelism is used.

6.3 Speedup and Running Time Comparison

In this section, we investigate speedups and running times of the different algorithms. We calculate a relative speedup of an algorithm as a ratio between its running time (averaged over ten repetitions) and its running time with $p = 1$. Fig. 4 shows box plots with speedups and times per edge of the codes run on Machines A and B for $k \in \{16, 64\}$. Each group stands for instances that have the same order of magnitude of edges. Note that Machine A has four sockets and thus has strong NUMA effects that affect the resulting speedups and running times of the codes. Therefore, to investigate the performance of the codes more thoroughly, we additionally present experiments performed on Machine B which is smaller but more tightly connected. The evaluation is based on those instances where all codes successfully

TABLE 3
Percent of Virtual Instances Partitioned by Mt − KaHIP 79 With Better *Penalized cut*; Percent of Imbalanced Virtual Instances; and Difference in Geometric Mean Penalized Cut Size Between Mt − KaHIP and Corresponding Other Code for *all Instances*

	Mt-Metis 79	ParHIP 79	KaHIP
% of virtual instances with better penalized cut	84.7 %	98.9 %	98.7 %
% of imbalanced virtual instances	66.5 %	23.4 %	2.8 %
Geom. mean penalized cut size difference	32.8 %	16.0 %	6.8 %

[2] Indeed, we asked Dominique LaSalle how to improve the quality of Mt − Metis at the expense of higher running time and he independently suggested to make repeated runs.

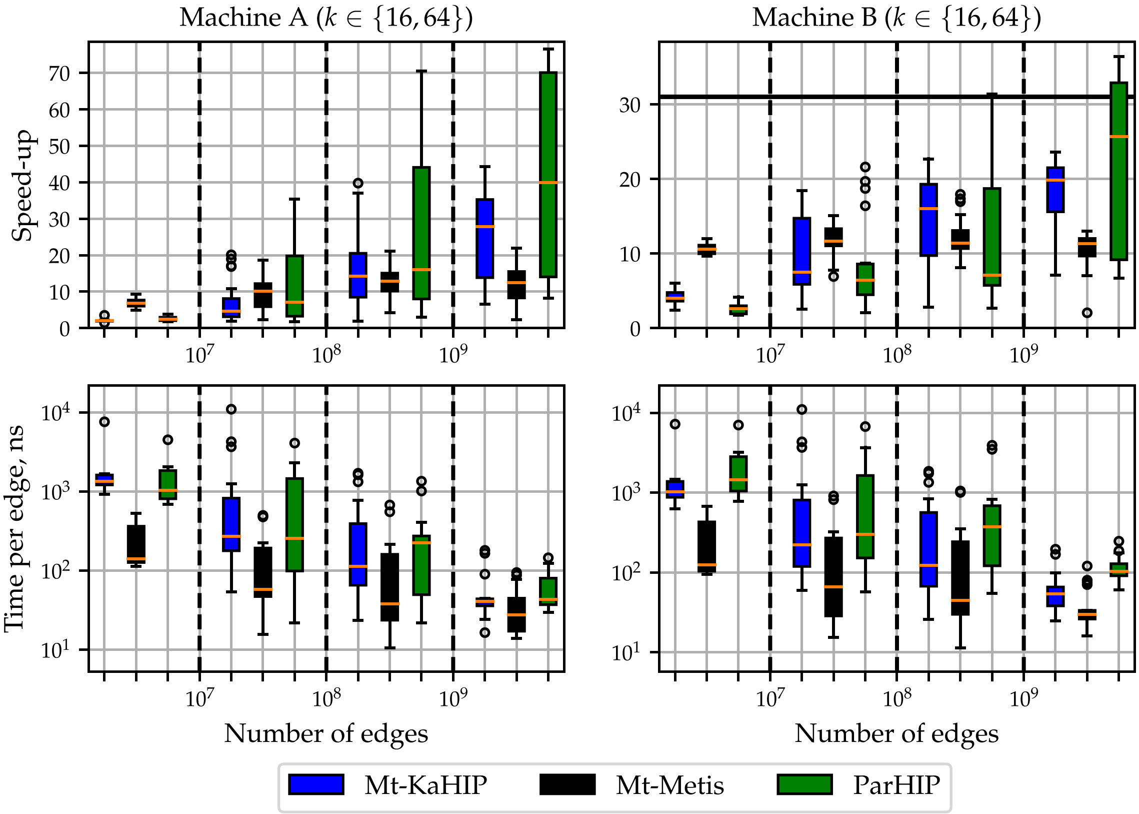

Fig. 4. Box plots with speedups and times per edge (ns) of the codes for Machine A, $p = 79$ and Machine B, $p = 31$ for $k \in \{16, 64\}$. Here each box spans between lower and upper quantiles, and a horizontal orange line is the median.

computed a (possibly imbalanced) partition – 126 instances on Machine A and 68 instances on Machine B.

First, we can see that all codes have problems making use of the large number of cores for smaller instances ($m < 10^7$). The best speedups are observed for large instances with $m \geq 10^8$. Moreover, Mt – Metis has the best overall running time but scales worse than the other codes. On both machines, speedups increase with increasing graph size for Mt – KaHIP and ParHIP. Considering the size of these machines, the speedups are quite good for Machine B and somewhat disappointing for Machine A. However, its not so surprising that an irregular code like graph partitioning does not scale well on a large loosely coupled NUMA machine.

Mt – KaHIP is somewhat faster than ParHIP and has comparable scalability. Considering its much better quality we can declare Mt – KaHIP a clear winner over ParHIP for shared memory graph partitioning. This comparison is a bit unfair though since ParHIP is actually a distributed-memory graph partitioner that scales much further for large "well-behaved" inputs [19]. We see a hint of this in an interesting difference between Machine A and B. For large instances, Mt – KaHIP scales better than ParHIP on Machine B but its the other way round on Machine A – apparently ParHIP is the only of the three codes that can make reasonable use of such a large, loosely coupled machine. A closer look reveals that this depends a lot on the instances. ParHIP has very high variability in the achieved speedup.

Fig. 6 shows speedup plots for three large instances from different instance families (rgg: 2d random geometric, del: 3d-Delaunay triangulations, and uk: web graph) and for $k \in \{16, 64\}$. On machine, B all instances show good speedup with the exception of the web graph for large k. On machine A, at least the geometric graphs (rgg/del) show good speedup. We see that increasing k significantly limits scalability. Further measurements indicate that the reason is that initial partitioning becomes a bottleneck – see also [34].

Additionally, we present speed-ups of all codes for $k \in \{256, 1024\}$ on Fig. 5. This plot encodes partition imbalances using colors. Here we can see that our competitors produce mostly imbalanced partitions (orange and red) unlike our code. On the other hand, Mt – KaHIP scales worse than other codes. Specifically, initial partitioning scales poorly. We conclude that none of codes shows good results for large k. However note that these problems can be circumvented by recursive k-partitioning using smaller values of k. For example, to obtain a 1024-partion, one can compute a 32-partition and recursively compute 32-partitions of each block. This might work better for Mt – KaHIP since one needs to tighten the imbalance parameters to meet the overall balance constraint. Codes that have difficulties to maintain balance might have problems in that situation.

6.4 Influence of Algorithmic Components

We now analyze how the parallelization of the different components affects solution quality of partitioning and present the speedups of each phase. We give a rough overview here and refer the reader to the PhD thesis of Yaroslav Akhremtsev [34] for more details. We perform experiments on Machine A with configurations of our algorithm in which only one of the components (coarsening, initial

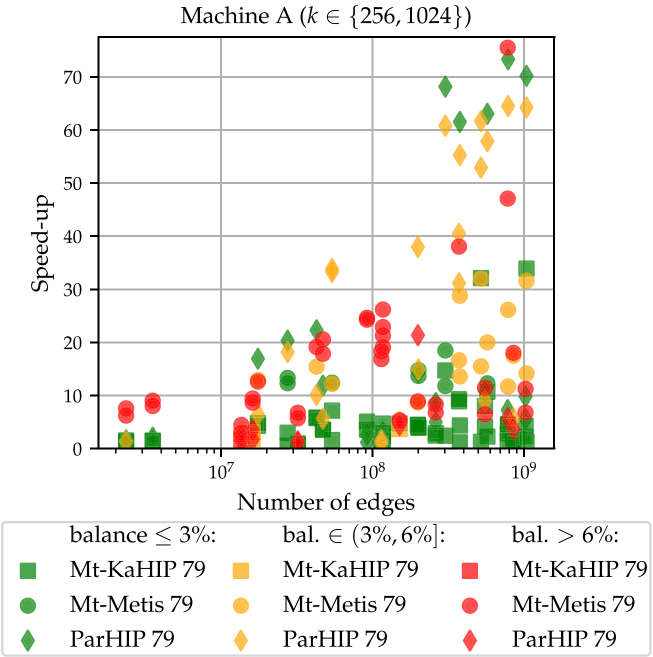

Fig. 5. Scatter plot with speed-ups of the codes for Machine A, $p = 79$ for $k \in \{256, 1024\}$. Colors encode balance as follows: green – balanced partitions, orange – balance between 3 and 6 percent, and red – balance greater than 6 percent.

partitioning, refinement) is parallelized. The respective parallelized component of the algorithm uses 79 PEs and the other components run sequentially. Running the algorithm with parallel coarsening or parallel local search increases the geometric mean of the cut by 0.68 or 0.57 percent respectively. These are small deteriorations that are not statistically significant. Running the algorithm with parallel initial partitioning decreases the geometric mean of the cut by 1.86 percent and this *is* statistically significant. The simple explanation is that trying more initial partitions better explores the search space.

To show that the parallelization of each phase is important, we consider running time shares of the phases of Mt – KaHIP with $p = 79$ (when all phases are parallel). On average, the coarsening, initial partitioning, and refinement phases take 30.4, 51.0, and 18.6 percent of the running time, respectively. We observe that each phase has the largest running time ratio for at least one instance. The *coarsening phase* takes 83.6 percent of the running time on the graph rgg_2_27 and $k = 16$. The *initial partitioning phase* takes 98.2 percent of the running time on the graph hollywood-2011 and $k = 64$. The *refinement* takes 71 percent of the running time on the graph er_2_22_2_23 and $k = 16$.

Fig. 7 shows speed-ups and times per edges for each of the phases of `Mt-KaHIP`. We can see that the coarsening and refinement phases have good scalability and times per edges. For refinement, this is a quite positive surprise – apparently, the sequential application of moves is not a big bottleneck. Initial partitioning turns out to be the main scalability bottleneck since it only uses trivial parallelization of the repetitions. This is a particular problem for small graphs, large k, or complex networks. This can be mitigated using a smaller base case size but this measure involves a difficult speed/quality trade-off.

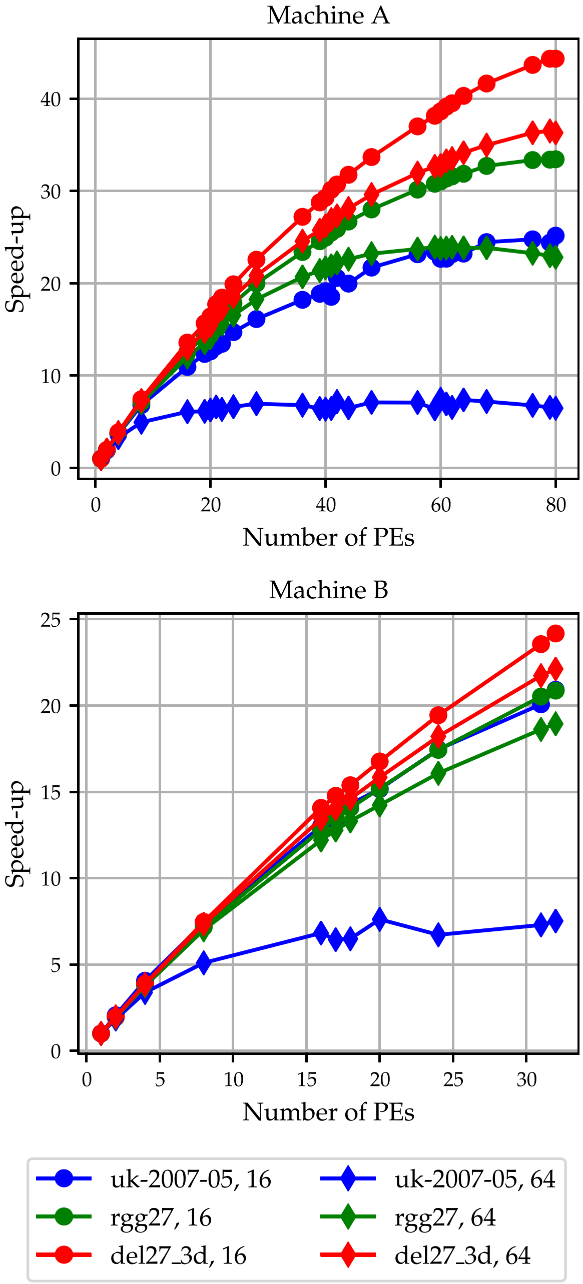

Fig. 6. Speedups for large instances (instance name, # of partitions k).

6.5 Memory Consumption

We now look at the memory consumption of Mt – KaHIP on the eight largest graphs from our benchmark for $k = 16$ (for $k = 64$ they are comparable) on Machine A. The geometric mean memory consumptions of Mt – KaHIP, Mt – Metis, and ParHIP are 44.8 GB, 53.2 GB, and 102.7 GB, respectively for $p = 1$ and 47.3 GB, 62.0 GB, and 109.1 GB for $p = 79$. Note that Mt – Metis was not able to partition sk-2005 and uk-2007. We observe only small memory overheads of Mt – KaHIP when increasing the number of PEs. We explain this by the fact that all data structures created by each PE are either of small (copy of a coarsened graph) or the data is distributed between them approximately uniformly (a hash table in LMLS). Note that all codes have relatively little memory overhead for parallelization. However, on average Mt – KaHIP with 79 PEs consumes 33.1 percent less memory

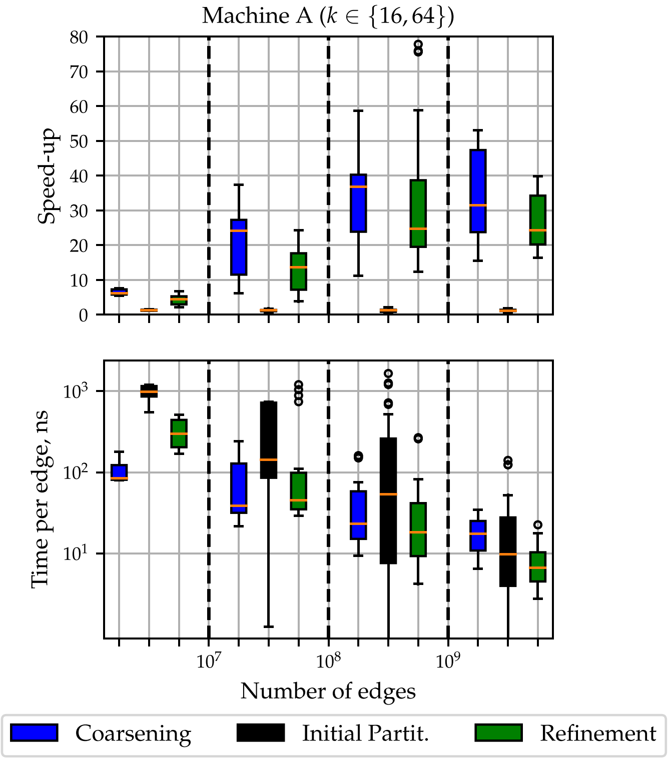

Fig. 7. Box plots with speedups and times per edge (ns) of coarsening, initial partitioning, and refinement phases of Mt-KaHIP for Machine A, $p = 79$ for $k \in \{16, 64\}$.

than Mt − Metis and 56.2 percent less memory than ParHIP on these graphs.

7 Conclusion and Future Work

Graph partitioning is a key prerequisite for efficient large-scale parallel graph algorithms and many other applications. We presented an approach to multi-level shared-memory parallel graph partitioning that achieves balanced solutions, shows high speedups for a variety of large graphs and yields very good quality independently of the number of cores used. Compared to previous approaches, ours produces smaller cuts and is better at keeping the balance constraints. Important ingredients of our algorithm include parallel label propagation for both coarsening and refinement, parallel initial partitioning, a simple yet effective approach to parallel localized local search, and fast locality preserving hash tables.

On the speed versus quality Pareto curve of state-of-the-art graph partitioners, Mt − KaHIP achieves much higher speed but somewhat lower quality than the high-quality settings of KaHIP (using flow-based refinement and/or evolutionary algorithms). Mt − Metis is faster than Mt − KaHIP but achieves considerably lower quality and cannot reliably guarantee balanced solutions. Systems like ParMetis [11] or XtraPuLP [22] are even faster and more scalable but with even lower quality.

At the same quality point as current Mt − KaHIP, we can probably achieve better scalability by looking for more scalable algorithms for initial partitioning. These also need to be revised in order to actually guarantee the balance constraint. Also the other components can perhaps be made more scalable. In particular, it would be interesting to get rid of the sequential approach to applying moves in Section 4.3 while avoiding the imbalances produced by the parallel approach of Mt − Metis. A scaleable distributed-memory parallelization achieving similar quality and efficiency as Mt − KaHIP would also be interesting.

Moving along the Pareto curve to higher quality solutions makes it interesting to parallelize flow-based refinement. Moving to faster codes while keeping at least some of the quality advantages of Mt − KaHIP seems to be a particularly important but challenging area for future research. Part of this can probably be achieved by devising high speed two-level algorithms for graph contraction/refinement that apply a high quality algorithm like Mt − KaHIP to the coarser level.

Acknowledgments

This work was supported in part by DFG Grants SA 933/10-2 and SCHU 2567/1-2.

References

[1] 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.

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

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

[4] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, no. 3, pp. 85–97, 1996.

[5] 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.

[6] C. Walshaw and M. Cross, "JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview," in *Mesh Partitioning Techniques and Domain Decomposition Techniques*, 2007, pp. 27–58.

[7] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*, L. Kliemann and P. Sanders, Eds. Berlin, Germany: Springer, 2014, vol. 9220, pp. 117–158.

[8] C. Schulz and D. Strash, "Graph partitioning: Formulations and applications to big data," in *Encyclopedia of Big Data Technologies*, Berlin, Germany: Springer, 2019. [Online]. Available: https://doi.org/10.1007/978–3-319-63962-8_312-2

[9] 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.

[10] C. Chevalier and F. Pellegrini, "PT-Scotch," *Parallel Comput.*, vol. 34, pp. 318–331, 2008.

[11] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proc. ACM/IEEE Conf. Supercomputing*, 1996, Art. no 35.

[12] D. LaSalle and G. Karypis, "A parallel hill-climbing refinement algorithm for graph partitioning," in *Proc. 45th Int. Conf. Parallel Process.*, 2016, pp. 236–241.

[13] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. 27th IEEE Int. Parallel Distrib. Process. Symp.*, 2013, pp. 225–236.

[14] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," *Proc. 24th IPDPS*, 2010, pp. 1–12.

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

[16] 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.

[17] 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.

[18] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. 6th WSDM*, 2013, pp. 507–516.

[19] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2625–2638, Sep. 2017.

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

[21] G. M. Slota, K. Madduri, and S. Rajamanickam, "Complex network partitioning using label propagation," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S620—S645, 2016. [Online]. Available: https://doi.org/10.1137/15M1026183

[22] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *Proc. 31st IEEE Int. Par. Distrib. Process. Symp.*, 2017, pp. 646–655.

[23] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. 18th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2012, pp. 1222–1230.

[24] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Conf. Web Search Data Mining*, 2014, pp. 333–342.

[25] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," *PVLDB*, vol. 11, no. 11, pp. 1590–1603, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1590-abbas.pdf

[26] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *Proc. 33rd IEEE Int. Conf. Data Eng.* , 2017, pp. 1083–1094.

[27] J. Nishimura and J. Ugander, "Restreaming graph partitioning: Simple versatile algorithms for advanced balancing," in *Proc. 19th ACM SIGKDD Conf. Knowl. Discov. Data Mining*, 2013, pp. 1106–1114.

[28] R. Glantz, H. Meyerhenke, and C. Schulz, "Tree-based coarsening and partitioning of complex networks," *ACM J. Exp. Algorithmics*, vol. 21, no. 1, pp. 1.6:1–1.6:20, 2016. [Online]. Available: https://doi.org/10.1145/2851496

[29] H. Meyerhenke, P. Sanders, and C. Schulz, "Partitioning (hierarchically clustered) complex networks via size-constrained graph clustering," *J. Heuristics*, vol. 22, no. 5, pp. 759–782, 2016. [Online]. Available: https://doi.org/10.1007/s10732-016-9315-8

[30] B. W. Kernighan, "Some graph partitioning problems related to program segmentation," Ph.D. dissertation, Princeton, 1969.

[31] 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.

[32] J. E. Savage and M. G. Wloka, "Parallelism in graph-partitioning," *J. Parallel Distrib. Comput.*, vol. 13, pp. 257–272, 1991.

[33] V. Osipov and P. Sanders, "n-level graph partitioning," in *Proc. 18th Eur. Symp. Algorithms*, vol. 6346, 2010, pp. 278–289.

[34] Y. Akhremtsev, "Parallel and external high quality graph partitioning," Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), 2019.

[35] C. L. Staudt and H. Meyerhenke, "Engineering parallel algorithms for community detection in massive networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 1, pp. 171–184, 2016.

[36] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-place parallel super scalar samplesort (IPSSSSo)," in *Proc. 25th Eur. Symp. Algorithms*, 2017, pp. 9:1–9:14.

[37] "Intel threading building blocks," [Online]. Available: https://www.threadingbuildingblocks.org/.

[38] J. Singler, P. Sanders, and F. Putze, "MCSTL: The multi-core standard template library," *Proc. 13th Euro-Par*, 2007, pp. 682–694.

[39] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general?(!)," *ACM Trans. Parallel Comput. (TOPC)*, vol. 5, 2019, Art. no. 16.

[40] J. Shun, G. Blelloch, J. T. Fineman, and P. B. Gibbons, "Reducing contention through priority updates," in *Proc. 25th ACM Symp. Parallelism Algorithms Architectures*, 2013, pp. 152–163.

[41] M. Patrascu and M. Thorup, "The power of simple tabulation hashing," in *Proc. 43rd ACM Symp. Theory Comp.*, 2011, Art. no. 14.

[42] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev, *Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox*. Berlin, Germany: Springer, 2019.

[43] Multi-threaded KaHIP. [Online]. Available: http://algo2.iti.kit.edu/kahip/ and https://github.com/yarchi/KaHIP/tree/add_parallel_local_search/

[44] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

[45] U. O. M. Laboratory of Web Algorithms, "Datasets," [Online]. Available: http://law.di.unimi.it/datasets.php

[46] D. Funke, S. Lamm, P. Sanders, C. Schulz, D. Strash, and M. von Looz, "Communication-free massively distributed graph generation," in *Proc. IEEE Int. Parallel Distrib. Symp.*, 2018, pp. 336–347.

[47] 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, 2014, pp. 73–82.

[48] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering*, Providence, RI, USA: American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013.

[49] 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.

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

[51] T. Davis, "The university of florida sparse matrix collection," [Online]. Available: www.cise.ufl.edu/research/sparse/matrices/.

[52] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945. [Online]. Available: http://www.jstor.org/stable/3001968

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

[54] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá, "Hyperbolic geometry of complex networks," *Phys. Rev. E*, vol. 82, 2010, Art. no. 036106.

Yaroslav Akhremtsev received the master's degrees in mathematics and computer science from the Moscow Power Engineering Institute, and the PhD degree from the Karlsruhe Institute of Technology. He is a software engineer with Google. He has several papers about graph partitioning, search trees, and randomized data structures. His research interests include sequential and parallel graph algorithms, randomized and non-randomized algorithms, and data structures.

Peter Sanders received the PhD degree in computer science from Universität Karlsruhe, Germany, in 1996. After 7 years at the Max-Planck-Institute for Informatics in 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, etc.) memory hierarchies, graph algorithms (route planning, graph partitioning, etc.), randomized algorithms, full text indices, etc. 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.

Christian Schulz received the master's degree in mathematics and computer science, and the PhD degree with summa cum laude from the Karlsruhe Institute of Technology. He is a postdoctoral researcher with the University of Vienna. He recently received the Heinz Billing Prize. His research interests include graph partitioning and clustering, parallel algorithms and combinatorial optimization in the context of big data.