High-Quality Shared-Memory Graph Partitioning

Yaroslav Akhremtsev[1(✉)], Peter Sanders[1], and Christian Schulz[2]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{yaroslav.akhremtsev,peter.sanders}@kit.edu
[2] University of Vienna, Vienna, Austria
christian.schulz@univie.ac.at

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 guarantees balanced solutions, shows high speed-ups for a variety of large graphs and yields very good quality independently of the number of cores used. For example, on 31 cores, our algorithm partitions our largest test instance into 16 blocks cutting *less than half* the number of edges than our main competitor when both algorithms are given the same amount of time. Important ingredients include parallel label propagation, parallel initial partitioning, a simple yet effective approach to parallel localized local search, and cache-aware hash tables.

1 Introduction

Partitioning a graph into k blocks of similar size such that few 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. 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 *uncoarsening/local search* phase, 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 due to the well-known fact that CPU technology increasingly provides more cores with relatively low clock rates

This is the short version of the technical report [2].

© Springer International Publishing AG, part of Springer Nature 2018
M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 659–671, 2018.
https://doi.org/10.1007/978-3-319-96983-1_47

in the last years since these are cheaper to produce and run. Moreover, shared-memory parallel algorithms implemented without message-passing libraries (e.g. MPI) usually give better speed-ups and running times than its MPI-based counterparts. Shared-memory parallel graph partitioning algorithms can in turn 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 a 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++14` multi-threading. Our approach uses a parallel label propagation algorithm that is able to shrink large complex networks fast during coarsening. Our parallelization of localized local search [10] is able to obtain high-quality solutions and guarantee 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. Our approach scales comparatively better than other parallel partitioners and has considerably higher quality which does not degrade with increasing number of processors.

After presenting preliminaries and related work in Sect. 2, we explain details of the multi-level graph partitioning approach and the algorithms that we parallelize in Sect. 3. Section 4 presents our approach to parallelization of the multi-level graph partitioning phases. Extensive experiments are presented in Sect. 5.

2 Preliminaries

2.1 Basic Concepts

Let $G = (V = \{0, \dots, n-1\}, E)$ 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 disjoint *blocks* of vertices $V_1,\dots,V_k$ that partition V; i.e., $V_1 \cup \dots \cup V_k = V$. The *balancing constraint* demands that all blocks have weight $c(V_i) \le (1+\epsilon)\lceil \frac{c(V)}{k} \rceil =: L_{\max}$ 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 the *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 which 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.

In general, 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.

We analyze algorithms using the concept of total *work* (the time spent by one processor) and *span*; i.e., the time spent using an unlimited number of PEs.

2.2 Related Work

There has been intensive research on graph partitioning so that we refer the reader to the full version of the paper and a recent overview [2,4]. 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, KaHIP, Metis and Scotch.

Probably the fastest available distributed memory parallel code is the parallel version of Metis, ParMetis [5]. 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 [6,7] is a shared-memory parallel version of the `ParMetis` graph partitioning framework. LaSalle and Karypis use a hill-climbing technique during refinement. The local search method is a simplification of k-way multi-try local search [10] 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.

Meyerhenke et al. [9] propose ParHIP, to partition large complex networks on distributed memory parallel machines using *label propagation*. The resulting system is more scalable and achieves higher quality than state-of-the-art systems like ParMetis or PT-Scotch. There are other parallel graph partitioners: PT-Scotch, KaPPa, and PDiBaP. See details in the full version of the paper [2].

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 uncoarsening/local search. In particular, we give a description of the sequential algorithms that we parallelize in the following sections. Our starting point here is the KaHIP framework [10]. For the development of the parallel algorithm, we add the k-way

multi-try local search scheme which gives higher quality, and improve it to perform less work than the original sequential version.

Coarsening. To create a new level of a graph hierarchy, we compute a clustering and build a coarse graph G'. Each original cluster corresponds to a single vertex in G'. The weight of this vertex is set to the sum of the weights of all vertices (in the finer graph) in the cluster. There is an edge between two vertices of G' if the corresponding clusters are connected by at least one edge. The weight of this edge is set to the sum of all edges (in the finer graph) that connect these clusters. The hierarchy created in this recursive manner is then used by the partitioner. Note 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 algorithm that we parallelize.

Clustering. We denote the set of all clusters as C and the cluster ID of a vertex v as $C[v]$. In our framework, we use the label propagation algorithm by Meyerhenke et al. [8] that creates clusters with constrained size. The size constrained label propagation algorithm works in iterations; i.e., the algorithm is repeated ℓ times (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 $\mathcal{C}$ that maximizes $\omega(\{(v, u) \mid u \in N(v) \cap \mathcal{C}\})$. If a vertex changes its cluster 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. The sequential running time of one iteration of the algorithm is $\mathcal{O}(m + n)$.

Initial Partitioning. After we have built the coarsest graph G', we partition it into k blocks using the algorithms from KaHIP [10]. To get a better solution, the graph G' is partitioned into k blocks I times and the best solution is returned.

Uncoarsening/Local Search. After initial partitioning, a local search algorithm is applied on each level of the multi-level hierarchy to decrease the cut size. There are a variety of local search algorithms: size-constraint label propagation, Fiduccia-Mattheyses k-way local search (FM), max-flow min-cut based local search, k-way multi-try local search (MLS) [10] .... Sequential versions of KaHIP use combinations of those. Since k-way local search is P-complete, we use a combination of the size-constrained label propagation algorithm and MLS. MLS achieves higher quality than FM [10] and decomposes the optimization into many small local searches which is a good basis for parallelization.

We now describe MLS that performs a k-way local searches around a single boundary vertices. This gives better chances of finding a nontrivial improvements [10]. The algorithm is organized in a nested loop of global and local

iterations. In the beginning of a global iteration, we put *all* boundary vertices into a todo list T. Initially, all vertices are unmarked. Afterwards, the algorithm repeatedly chooses and removes a random vertex $v \in T$. If v is not marked then it performs a k-way local search around v. It marks v and $N(v)$ and inserts them into the priority queue PQ using gain values as keys. Next, the algorithm extracts a vertex w with the maximum key in the PQ. If the corresponding move of w does not produce an overloaded block then it performs the move and inserts all unmarked neighbors of w into the PQ. The algorithm stops when the priority queue is empty or an adaptive stopping rule decides to stop. 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. Afterwards, the algorithm reinserts moved vertices into T in a random order. If the achieved gain improvement is larger than a certain percentage (currently 10 %) of the total improvement during the current global iteration, it continues to perform moves around the vertices currently in the todo list (next local iteration). This allows to further decrease the cut size without significant impact to the running time. When improvements fall below this threshold, the next global iteration is started that initializes the todo list with all boundary vertices. After a fixed number of global iterations (currently 3), the MLS algorithm stops. Our experiments show that 3 global iterations is a fair trade-off between the running time and quality of the partition. This nested loop of local and global iterations is an improvement over the original MLS search from [10] since they allow for a better control of the running time 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.

4 Parallel Multi-level Graph Partitioning

Profiling the sequential algorithm shows that each of the components of the multi-level scheme has a significant contribution to the overall algorithm. Our general approach is to avoid bottlenecks as well as performing independent work as much as possible.

4.1 Coarsening: Parallel Size-Constraint Label Propagation

To parallelize the size-constraint label propagation algorithm, we adapt a clustering technique by Staudt and Meyerhenke [12]. First, we sort the vertices in increasing order of their degrees using a parallel sorting Algorithm [3]. Then we form work packets of vertices and put them into a concurrent queue. We constraint each packet to contain vertices with a total number of at most $\sqrt{m}$ neighbors. Additionally, we have an empty queue Q' that stores packets for the

next iteration. During an iteration, each PE tries to extract a packet from the queue Q. It 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 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 $\sqrt{m}$ 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 $\mathcal{O}(n+m)$ work and $\mathcal{O}(\frac{n+m}{p})$ span.

Coarsening: Parallel Contraction

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 by calculating a prefix sum on an array that contains ones in the positions equal to the current cluster IDs. This phase can be done in $\mathcal{O}(n)$ work. Sequentially, the edge weight accumulation step calculates weights of edges in E' using hashing. For each cut edge $(v, u) \in E$, we insert a pair $(C[v], C[u])$ 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 work 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 $\mathcal{O}(|V'| + |E'|)$ work. Hence, the total expected work to compute the coarse graph is $\mathcal{O}(m + n + |E'|)$.

The parallel contraction algorithm works as follows. First, we remap the cluster IDs using parallel prefix sums. Edge weights are accumulated by iterating over the edges of the original graph in parallel. This uses a concurrent hash table. The third phase is performed sequentially in the current implementation since profiling indicates that it is so fast that it is not a bottleneck.

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. Each PE creates a copy of the coarsest graph and runs KaHIP sequentially on it using a random seed. Assume that one partitioning can be done in T time. 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.

4.3 Uncoarsening/Local Search

Our parallel algorithm first uses size-constraint parallel label propagation to improve the current partition and afterwards applies our parallel MLS. The idea

is that label propagation is easy to parallelize and will do all the easy improvements. Subsequent MLS will then invest considerable work to find nontrivial improvements. In this combination, only few nodes actually need be moved globally which makes it easier to parallelize MLS scalably. When using the label propagation algorithm to improve a partition, we set the upper bound U to $L_{\max}$.

Parallel MLS works in a nested loop of local and global iterations as in the sequential version. Initialization of a global iteration uses a simplified parallel shuffling algorithm where each PE shuffles the nodes it considers into a local bucket and then the queue is made up of these buckets in random order. During a local iteration, each PE extracts vertices from a producer queue Q. Afterwards, it performs *local* moves around it; that is, global block IDs and the sizes of the blocks remain *unchanged*. When the producer queue Q is empty, the algorithm applies the best found sequences of moves to the global data structures and proceeds with the next local iteration.

Performing Moves. Each PE performs moves in the function `PerformMoves`. Starting from a single boundary vertex, each PE performs *local* moves of vertices to find a sequence of moves that decreases the cut. That is, moves do not affect the current global partition – they 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 smallest cut is returned.

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 compare and swap instruction) and they are never unset. Hence, the marking operation can be seen as priority update operation (see Shun et al. [11]) 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 queue Q is empty, it sets a global variable to signal the other threads to finish the current call of the function `PerformMoves`. This way, isolated very long MLS searches cannot lead to bad load balance.

Let each PE process a set of edges $\mathcal{E}$ and a set of vertices $\mathcal{V}$. Since a vertex can be moved only by one PE and moving it requires to compute gain for 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 is updated at most $d(v)$ times.

Applying Moves. Let M_i denote the set of sequences of moves performed by PE i. 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 may affect a move done by another PE and the cut size may even increase. To prevent this, we recalculate the gain of each move in a given sequence and apply the subsequence of moves that gives the best cut. Finally, we insert all moved vertices into the queue Q. 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.

4.4 Differences to Mt-Metis

We now discuss differences between our algorithm and Mt-Metis. In the coarsening phase, we use a cluster contraction while Metis is using a matching-based scheme. Our approach is especially well suited for networks that have a pronounced and hierarchical cluster structure. The general initial partitioning scheme is similar in both algorithms. However, the employed sequential techniques differ because different sequential tools (KaHIP and Metis) are used to partition the coarsest graphs. In terms of local search, unlike 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 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 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.

4.5 Further Optimization

In this section, we list further optimization techniques that we use to achieve better speed-ups and overall speed. More precisely, we use cache-aligned arrays to mitigate the problem of false-sharing, the TBB scalable allocator [1] for concurrent memory allocations and pin threads to cores to avoid rescheduling overheads. Additionally, we use a cache-aware hash table that is described in the full version of the paper [2]. 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 Experiments

We implemented our algorithm `Mt-KaHIP` (Multi-threaded KaHIP) within the `KaHIP` [10] framework using `C++` and the `C++14` multi-threading library. We plan to make our program available in the framework. All binaries are built using `g++-5.2.0` with the `-O3` flag and 64-bit index data types. We run our experiments on two machines. Machine A is an Intel Xeon E5-2683v2 (2 sockets, 16 cores with Hyper-Threading, 64 threads) running at 2.1 GHz with 512 GB RAM. Machine B is an Intel Xeon E5-2650v2 (2 sockets, 8 cores with Hyper-Threading, 32 threads) running at 2.6 GHz with 128 GB RAM.

We compare ourselves to Mt-Metis 0.6.0 using the default configuration with hill-climbing being enabled (*Mt-Metis*) as well as sequential KaHIP 2.0 using the `fast social` configuration (*KaHIP*) and ParHIP 2.0 [9] using the `fast social` configuration (*ParHIP*). According to LaSalle and Karypis [6] `Mt-Metis` has better speed-ups and running times compared to `ParMetis` and `Pt-Scotch`. At the same time, it yields similar solution quality. Hence, we do not perform experiments with `ParMetis` and `Pt-Scotch`. Our algorithm consumes 44.3% less memory than `Mt-Metis` on the largest graph from our benchmark set for $p = 31$ on machine A. For more details, we refer the reader to [2].

Our default value of allowed imbalance is 3%. We call a solution imbalanced if at least one block exceeds this amount. 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. If at least one repetition returns an imbalanced partition of an instance then we mark this instance imbalanced. Our experiments focus on the cases $k \in \{16, 64\}$ and $p \in \{1, 16, 31\}$ to save running time.

We use performance plots to present quality comparisons and scatter plots to present the speed-up and the running time comparisons. A curve in a performance plot for algorithm X is obtained as follows: For each instance (graph and k), we calculate the normalized value $1 - \frac{\texttt{best}}{\texttt{cut}}$, where `best` is the best cut obtained by any of the considered algorithms and `cut` is the cut of algorithm X. These values are then sorted. Thus, the result of the best algorithm is in the bottom of the plot. We set the value for the instance above 1 if an algorithm builds an imbalanced partition. Hence, it is in the top of the plot.

Any multi-level algorithm has a considerable number of tuning parameters. We adopt parameters from the coarsening and initial partitioning phases of `KaHIP`. `Mt-KaHIP` uses 10 and 25 label propagation iterations during coarsening and refinement, respectively, partitions a coarse graph $\max(p, 4)$ times in initial partitioning and uses 3 global iterations of parallel MLS in the refinement phase.

Instances. We evaluate all algorithms on a benchmark of 24 large graphs and for $k \in \{16, 64\}$. This collection consist of different kinds of graphs: numeric simulations, complex networks (focused on social networks and web graphs), and random graphs (*random geometric graphs*, *delaunay graphs*, and *random*

hyperbolic graphs). Details of the benchmark can be found in the full version of the paper [2].

5.1 Quality Comparison

In this section, we compare our algorithm against competing 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.

Our algorithm gives the best overall quality, usually producing the overall best cut. Even in the small fraction of instances where other algorithms are best, our algorithm is at most 7% off. The overall solution quality does not heavily depend on the number of PEs used. Indeed, more PEs give slightly higher partitioning quality. The original fast social configuration of KaHIP as well as ParHIP produce worse quality than Mt-KaHIP, since parallel MLS significantly improves solution quality. Mt-Metis with $p = 1$ has worse quality than our algorithm on almost all instances. 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 can also be seen from the geometric means of the cut sizes over all instances, including the imbalanced solutions. For our algorithm they are 727.2K, 713.4K and 710.8K for $p = 1, 16, 31$, respectively. For Mt-Metis they are 819.8K, 873.1K and 874.8K for $p = 1, 16, 31$, respectively. For ParHIP they are 809.9K, 809.4K and 809.71K for $p = 1, 16, 31$, respectively, and for KaHIP it is 766.2K. For $p = 31$, the geometric mean cut size of Mt-KaHIP is 18.7% smaller than that of Mt-Metis, 12.2% smaller than that of ParHIP and 7.2% smaller than that of KaHIP.

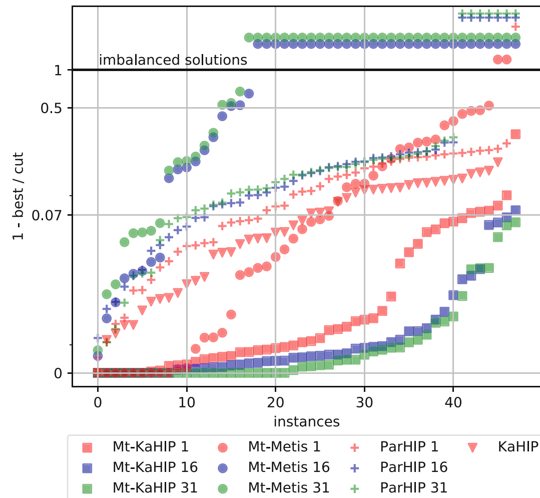

Fig. 1. Performance plot for the cut size. The number behind the algorithm name denotes the number of threads.

Additionally, we compare the effectiveness of our algorithm Mt-KaHIP against competitors. We give the faster algorithm the same amount of time as the slower algorithm for additional repetitions that can lead to improved solutions. The detailed description of these experiments is in the full version of the paper [2]. Still in 80.4% of the tests Mt-KaHIP has better quality than Mt-Metis. In the worst-case, Mt-KaHIP has a 5.5% larger cut than Mt-Metis. In 96.5% of the tests, Mt-KaHIP has better quality than ParHIP. In the worst-case,

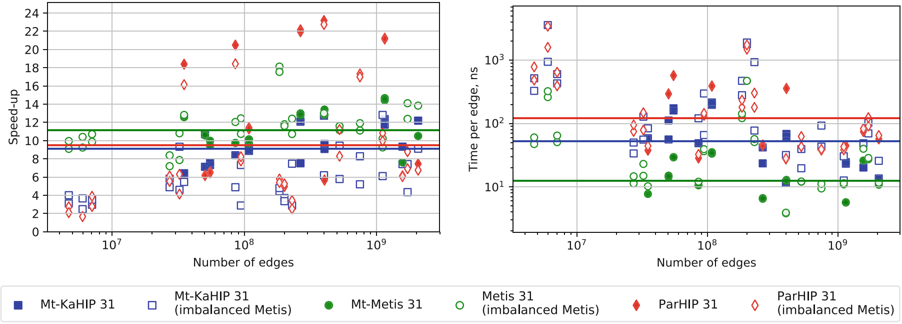

Fig. 2. From left to right for $p = 31$: (a) full speed-up, (b) full running time per edge in nanoseconds. Horizontal lines are harmonic and geometric means.

Mt-KaHIP has a 5.4% larger cut than ParHIP. In 98.9% of the tests, Mt-KaHIP has better quality than KaHIP. In the worst-case, Mt-KaHIP has a 3.5% larger cut than KaHIP.

5.2 Speed-Up and Running Time Comparison

In this section, we compare the speed-ups and the running times of our algorithm against competitors. We calculate a relative speed-up of an algorithm as a ratio between its running time and its running time with $p = 1$. Figure 2 show scatter plots with speed-ups and time per edge for a full algorithm execution on machine A. We calculate the geometric and harmonic means *only* for instances that were partitioned in ten repetitions *without imbalance*. Note that among the top 20 speed-ups of Mt-Metis 60% correspond to imbalanced instances (*Mt-Metis 31 imbalanced*) thus we believe it is fair to exclude them.

The harmonic mean full speed-up of our algorithm, Mt-Metis and ParHIP for $p = 31$ are $9.1, 11.1$ and 9.5, respectively. The harmonic mean local search speed-up of our algorithm, Mt-Metis and ParHIP are 13.5, 6.7 and 7.5, respectively. Our full speed-ups are comparable to that of Mt-Metis but our local search speed-ups are significantly better than that of Mt-Metis. The geometric mean full time per edge of our algorithm, Mt-Metis and ParHIP are 52.3 nanoseconds (ns), 12.4 [ns] and 121.9 [ns], respectively. The geometric mean local search time per edge of our algorithm, Mt-Metis and ParHIP are 3.5 [ns], 2.1 [ns] and 16.8 [ns], respectively. Note that with increasing number of edges, our algorithm has comparable time per edge to Mt-Metis. Superior speed-ups of parallel MLS are due to minimized interactions between PEs and using cache-aware hash tables locally. Although on average, our algorithm is slower than Mt-Metis, we consider this as a fair trade off between the quality and the running time. We also dominate ParHIP in terms of quality and running times.

5.3 Influence of Algorithmic Components

We now analyze how the parallelization of the different components affects the cut size and present the speed-ups of each phase. We perform experiments on machine B with configurations of our algorithm in which *only one* of the components (coarsening, initial partitioning, uncoarsening) is parallelized using $p = 16$. Running the algorithm with parallel coarsening decreases the geometric mean of the cut by 0.7%, with parallel initial partitioning decreases the cut by 2.3% and with parallel local search decreases the cut by 0.02%. Compared to the full sequential algorithm, we conclude that running the algorithm with any parallel component either does not affect solution quality or improves the cut slightly on average. The parallelization of initial partitioning gives better cuts since it computes more initial partitions than the sequential version.

To show that the parallelization of each phase is important, we consider instances where one of the phases runs significantly longer than other phases in the experiments on machine A using $p = 31$. The *coarsening phase* may take up to 91% of the running time and its parallelization gives a speed-up of 13.6 for 31 threads and a full speed-up of 12.4. The *initial partitioning phase* may take up to 40% of the running time and its parallelization gives a speed-up of 6.1 and the overall speed-up is 7.4. The *uncoarsening phase* may take up to 57% of the running time and its parallelization gives a speed-up of 13.0 and the overall speed-up is 9.1. The harmonic mean speed-ups of the coarsening phase, the initial partitioning phase and the uncoarsening phase for $p = 31$ are 10.6, 2.0 and 8.6, respectively.

6 Conclusion and Future Work

We presented a shared-memory parallel graph partitioner that is able to partition graphs with very good quality as well as guaranteed balance which also shows good speed-up on a variety of large graphs. Previous approaches show a negative trade-off between quality and speed. The important parts of our algorithm are parallel label propagation, a simple yet effective approach to parallel MLS, parallel initial partitioning, and cache-aware hash tables. Considering the good results of our algorithm, we want to further improve it and release its implementation. An interesting problem is how to apply moves in Sect. 4.3 in parallel whose solution will increase the performance of parallel MLS.

References

1. Intel threading building blocks. https://www.threadingbuildingblocks.org/
2. Akhremtsev, Y., Sanders, P., Schulz, C.: High-quality shared-memory graph partitioning. CoRR abs/1710.08231 (2017)
3. Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: In-place parallel super scalar samplesort (IPSSSSo). In: Proceedings of the 25th ESA, pp. 9:1–9:14 (2017)

4. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent advances in graph partitioning. In: Kliemann, L., Sanders, P. (eds.) Algorithm Engineering. LNCS, vol. 9220, pp. 117–158. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49487-6_4
5. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In: Proceedings of the ACM/IEEE Conference on Supercomputing (1996)
6. LaSalle, D., Karypis, G.: Multi-threaded graph partitioning. In: Proceedings of the 27th IPDPS, pp. 225–236 (2013)
7. LaSalle, D., Karypis, G.: A parallel hill-climbing refinement algorithm for graph partitioning. In: Proceedings of the 45th ICPP, pp. 236–241 (2016)
8. Meyerhenke, H., Sanders, P., Schulz, C.: Partitioning complex networks via size-constrained clustering. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 351–363. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07959-2_30
9. Meyerhenke, H., Sanders, P., Schulz, C.: Parallel graph partitioning for complex networks. In: IEEE Transactions on Parallel and Distributed Systems, pp. 2625–2638 (2017)
10. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 469–480. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23719-5_40
11. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B.: Reducing contention through priority updates. In: Proceedings of the 25th SPAA, pp. 152–163 (2013)
12. Staudt, C.L., Meyerhenke, H.: Engineering parallel algorithms for community detection in massive networks. IEEE Trans. Parallel Distrib. Syst. **27**(1), 171–184 (2016)