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 partitions 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 speedups for a variety of large graphs and yields very good quality independently of the number of cores used. For example, our algorithm partitions a graph with 2 billions edges using 16 cores in 53 seconds producing a solution that cuts two times less edges than our main competitor which runs 33 seconds. Important ingredients include parallel label propagation for both coarsening and improvement, 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

I. INTRODUCTION

The graph partitioning problem arises in almost all parallel memory-distributed problems where there is an implicit (or explicit) graph $G$ that represents inter-connections within the data to process. A good example is the distribution of the Facebook social graph among multiple servers [38]. The less vertices that share an edge reside on different servers the smaller query time as well as the load of the network and the servers. In particular, when you process a graph in parallel on $k$ PEs (processing elements), you often want to partition the graph into $k$ blocks of about equal size. In this paper we focus on a version of the problem that constrains the maximum block size to $(1 + \epsilon)$ times the average block size and tries to minimize the total cut size; i.e., the number of edges that run between blocks.

The graph partitioning problem is NP-hard [16], [13] and there is no approximation algorithms with a constant ratio factor for general graphs [7]. 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 depicted in Figure 1. The main idea is to contract the graph in the coarse-grain phase until it 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 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 corresponded 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 the C++14 multi-threading library. Roughly speaking, our approach uses a parallel label propagation algorithm that is able to shrink large complex networks fast during coarsening, and then uses a parallelization of a localized local search approach which is able to obtain high-quality solutions.

The rest of the paper is organized in the following way. After presenting preliminaries and related work in Section II, we explain details of the multi-level graph partitioning approach and the algorithms that we parallelize in Section III. Section IV presents our approach to the parallelization of the multi-graph partitioning phases. More precisely, we present a parallelization of the label propagation algorithm with size-constraint by Meyerhenke et al. [30], as well as a parallelization of $k$-way multi-try local search by Sanders and Schulz [35] that is able to provide high-quality solutions. In Section V we describe further optimizations to achieve better speed-ups and running times. Extensive experiments to...
evaluate the performance of different algorithms are presented in Section [VI]. Our experiments show that our approach has better or similar scalability and always produces balanced partitions of better quality than competing shared-memory and distributed memory parallel approaches. Finally, we conclude in Section [VII].

II. PRELIMINARIES

A. Basic concepts

Let \( G = (V = \{0, \ldots, n - 1\}, E, c, \omega) \) be an undirected graph with edge weights \( \omega : E \rightarrow \mathbb{R}_{\geq 0} \), vertex weights \( c : V \rightarrow \mathbb{R}_{\geq 0}, n = |V|, \) and \( m = |E| \). We extend \( c \) and \( \omega \) to sets, i.e., \( c(V') := \sum_{v \in V'} c(v) \) and \( \omega(E') := \sum_{e \in E'} \omega(e) \). \( N(v) := \{ u : \{v, u\} \in E \} \) denotes the neighbors of \( v \). The degree of a vertex \( v \) is defined as \( |N(v)| \). We denote the maximum vertex degree as \( \Delta \). A vertex \( v \in V_i \) that has a neighbor \( w \in V_j, i \neq j \), is a boundary vertex. We are looking for blocks of vertices \( V_1, \ldots, V_k \) that partition \( V \); i.e., \( V_1 \cup \cdots \cup V_k = V \) and \( V_i \cap V_j = \emptyset \) for \( i \neq j \). The balancing constraint demands that \( \forall i \in \{1, k\} : c(V_i) \leq L_{\max} := (1 + \epsilon) \cdot \left[ \frac{c(V)}{k} \right] \) for some imbalance parameter \( \epsilon \). The objective is to minimize the total cut \( \sum_{i<j} w(E_{ij}) \) where \( E_{ij} := \{ u, v \in E : u \in V_i, v \in V_j \} \). We call a block \( V_i \) underloaded [overloaded] if \( c(V_i) < L_{\max} \) [if \( c(V_i) > L_{\max} \)]. We define the gain of a vertex \( v \in V_i \) as \( \max_{u \neq i} \omega(\{(v, w) : w \in N(v) \cap V_i\}) - \omega(\{(v, w) : w \in N(v \cap V_i)\}) \). 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 \( U \) such that \( c(V_i) \leq U \). Note that by adjusting the upper bound one can somehow control the number of blocks of a feasible clustering.

An abstract view of the partitioned graph is the so-called 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 which 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 multilevel algorithm. In order to avoid tedious notation, \( G \) will denote the current state of the graph before and after a (un)contraction in the multilevel scheme throughout this paper.

B. Related Work

There has been a huge amount of research on graph partitioning so that we refer the reader to [37], [5], [8] for most of the material. 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 multilevel principle. Well-known software packages based on this approach include Jostle [45], KaHIP [35], Metis [18] and Scotch [10].

Most probably the fastest available distributed memory parallel code is the parallel version of Metis, ParMetis [17]. This parallelization has problems maintaining the balance of the partitions 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 [25], [24] is a shared-memory parallel version of the ParMetis graph partitioning framework. LaSalle and Karypis use a hill-climbing technique during the refinement. The local search method is a simplification of the multi-try search [35] in order to make it fast. The idea is to find a set of vertices (hill) whose move to other partition 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 [29] is a multilevel 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. [34]. A single round of simple label propagation can be interpreted as the randomized agglomerative clustering approach proposed by Catalyurek and Aykanat [9]. Moreover, the label propagation algorithm has been used to partition networks by Ugander and Backstrom [42]. The authors do not use a multilevel scheme and rely on a given or random partition which is improved by combining the unconstrained label propagation approach with linear programming. The approach does not yield high quality partitions.

Meyerhenke et al. [31] 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 state-of-the-art systems like ParMetis or PT-Scotch.

III. 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. The sequential implementations of these algorithms are contained in the KaHIP [35] graph partitioning framework.
A. 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. This coarsening procedure is repeated recursively until the coarsest graph is small enough. Contracting the 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 block $A$ and block $B$ 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 algorithm that we parallelize.

Clustering: We denote the set of all clusters as $C$ and the cluster ID of a vertex $v$ as $C[v]$. There are a variety of clustering algorithms. Some of them build clusters of size two (matching algorithms) and other build clusters with size less than preset upper bound. In our framework, we use the label propagation algorithm by Meyerhenke et al. [30] that builds clustering adhering to a preset size-constraint.

The size constrained label propagation algorithm works in iterations; i.e., the algorithm is repeated $\ell$ times, where $\ell$ 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 $C$ of a vertex $v$ is called eligible if $C$ will not become overloaded once $v$ is moved to $C$. 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 $C$ that maximizes $\omega(\{(v, u) \mid u \in N(v) \cap C\})$. 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 iteration, $Q$ and $Q'$ are swaped, and the algorithm proceeds with the next iteration. The sequential running time of one iteration the algorithm is $O(m + n)$.

B. Initial Partitioning

After constructing the hierarchy of coarse graphs, the coarsest one is partitioned into $k$ blocks using a recursive bisection algorithm [20] implemented in KaHIP. More precisely, it is partitioned into two blocks and then the subgraphs induced by these two blocks are recursively partitioned into $\left\lfloor \frac{k}{2} \right\rfloor$ and $\left\lceil \frac{k}{2} \right\rceil$ blocks each. To get a better solution, the coarsest graph is partitioned into $k$ blocks $T$ times and the best solution is returned.

C. Uncoarsening/Local Search

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; i.e., a vertex in the finer graph is assigned the block of its coarse representative. This process is then repeated for each level of the hierarchy.

There are a variety of local search algorithms: size-constraint label propagation, Fiduccia-Mattheyses $k$-way local search [12], [19], max-flow min-cut based local search [35], $k$-way multi-try local search [35] . . . . Sequential versions of KaHIP use a combinations of those. Since $k$-way local search is P-complete [50], our algorithm uses size-constraint label propagation in combination with $k$-way multi-try local search. More precisely, the size-constraint 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 a 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 multi-try $k$-way local search (MLS). In contrast to previous $k$-way local search methods MLS is not initialized with all boundary vertices; that is, all boundary vertices are eligible for movement at the beginning. Instead, the method is repeatedly initialized with a single boundary vertex. Intuitively, this introduces a larger amount of diversification compared to the classical method that is restricted to move the boundary vertices having the largest gain.

The algorithm is organized in iterations. An iteration works as follows. 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 the 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. 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 [32] decides to stop or when the priority queue is empty. More precisely, if the overall gain is negative then the stop 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 iteration, this is repeated until the todo list is empty.

The running time of one iteration of the algorithm is $O(n + \sum_{v \in V} d(v) \log d(v)) = O(n + m \log \Delta)$. Because each vertex can be moved only once during an iteration and we update the gains of its neighbors using a bucket heap [12]. The $\log \Delta$ term is the cost to update the gain of a vertex that is already
in the priority queue. 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.

IV. PARALLEL MULTI-LEVEL GRAPH PARTITIONING

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 uncoarsening.

A. Coarsening

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

Parallel Size-Constraint Label Propagation: To parallelize the size-constrained label propagation algorithm, we employ ideas by Staudt and Meyerhenke [41]. Their algorithm iterates in parallel over a set of active vertices; i.e., vertices whose neighbors changed their label on previous iteration, and use guided scheduling that assigns shrinking whole subsets of active vertices to PEs. We improve their idea by balancing too large subsets work wise and yields better load balancing.

Now, two more problems have to be solved: concurrent updates of cluster sizes and the maintenance of the queue that contains the sets of active vertices. To concurrently update the sizes of the clusters, we use an atomic compare and swap operation \( \text{CAS}(x, y, z) \) operation. This operation works as follows: if \( x = y \) then this operation assigns \( x \leftarrow z \) and returns True; otherwise it returns False.

To solve the second problem, we use the TBB concurrent queue [1] that allows concurrent push and pop operations. Each element in the queue is a set of active vertices with total degree not greater than \( O(\sqrt{m}, 1000)) \).

In the beginning of the algorithm, we use the parallel sorting algorithm by Axtmann et al. [2] to sort vertices in increasing order of their degrees. Next, we build sets/blocks as described above and insert those into the queue \( Q \). Additionally, we have an empty queue \( Q' \) that stores blocks 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 block of active vertices from the queue. A PE then chooses a new cluster for each vertex in the currently processed block. During this process, neighbors of the processed vertices are inserted into a block for the next iteration if the vertex changed its cluster. If the sum of vertex degrees in the block for the next iteration exceeds \( O(\sqrt{m}, 1000)) \) then it is inserted into the queue \( Q' \). When the queue \( Q \) is empty, the main PE exchanges \( Q \) and \( Q' \) and we proceed with the next iteration. A pseudocode description of the parallel label propagation is given in Algorithm 1. One iteration of the algorithm can be done with \( O(n + m) \) work and \( O(\frac{n + m}{p}) \) span.

Algorithm 1: Parallel Size-Constraint Label Propagation

```
Input: graph \( G = (V, E) \); cluster-size upper bound \( U \); iterations \( I \)
Output: clustering \( C \);

1. \( V_s = \text{parallel_sort}(V) \)
2. \( Q = V_s; \) // concurrent queue
3. \( Q' = \emptyset; \) // queue for the next iteration
4. \( C \leftarrow \{0, \ldots, |V| - 1\}; \) // init clustering
5. for \( i \leftarrow 1 \) to \( I \)
6. while \( Q \) is not empty do // parallel
7. Block \( B = Q\text{.pop}(); \)
8. Block \( B_n; \) // new active block
9. foreach \( v \in B \) do
10. Map map; // connection strengths
11. foreach \( w \in N(v) \) do
12. \( \text{map}[C[w]] \leftarrow \text{map}[C[w]] + w(v, w); \)
13. \( c \leftarrow \text{arg max}\{\text{map}[c] : size[c] + w(v) < U\}; \)
14. if \( \text{map}[C[v]] > \text{map}[c] \) then
15. continue; // nothing to be done
16. // move \( v \) to \( c \) and update size
17. move \( v \rightarrow \text{true}; \)
18. do
19. \( S \leftarrow \text{size}[c]; \)
20. if \( S + w(v) > U \) then
21. move \( v \rightarrow \text{false}; \) break;
22. while \( \text{CAS(size}[c], S, S + w(v)); \)
23. if not move then
24. continue; // atomically:
25. \( \text{size}[C[v]] \leftarrow \text{size}[C[v]] - w(v); \)
26. \( C[v] \leftarrow c; \) \( B_n = B_n \cup N(v); \)
27. if \( \sum_{w \in B_n} \text{deg}(w) > \max(\sqrt{m}, 1000) \) then
28. \( Q'.\text{push}((w : w \in B_n \text{ and } w \notin Q')) \);
exchange \( Q \) and \( Q' \);
```

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 where \(|V'|\) is the number of cluster 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 \( O(n) \) time when done sequentially. Sequentially, 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 \((C[v], C[u])\) 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 $O(|E'| + m)$. To construct the coarse graph we iterate over all edges $E'$ contained in the hash table. This takes time $O(|V'| + |E'|)$. Hence, the total expected running time to compute the coarse graph is $O(m + n + |E'|)$ when run sequentially.

The parallel contraction algorithm works as follows. First, we remap the cluster IDs using the parallel prefix sum algorithm by Singler et al. \cite{Singler2006}. Next, we iterate over the edges of the original graph in parallel constructing the set of edges of the coarse graph and aggregating the weights of parallel edges of the coarse graph using a concurrent hash table by Maier and Sanders \cite{Maier2011}. Finally, we construct the coarse graph from the edges in the hash table and their corresponding weights in parallel as follows. We put each edge $(C[v], C[u])$ into the buckets for $C[u]$ and $C[v]$ and then construct an array of edges. Note that the construction of the coarse graph is much faster than aforementioned phases and its effect on the total speed-up is insignificant. Thus, we use the sequential version in our algorithm. The expected total work of the algorithm is $O(m + n)$ and the expected span is $O\left(\frac{m+n}{p}\right)$.

### B. 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 $T$ time. Then $\max(p, I)$ partitions can be built with $O(\max(p, I) \cdot T + p \cdot (|E'| + |V'|))$ work and $O(\frac{\max(p, I) \cdot T}{p} + |E'| + |V'|)$ span, where the additional terms $|V'|$ and $|E'|$ account for the time each PE copies the coarsest graph.

### C. Uncoarsening/Local Search

Our parallel algorithm first uses the size-constraint parallel label propagation to improve the current partition and afterwards applies our parallel MLS. When using the label propagation algorithm to improve a partition we set the upper bound $U$ to the size-constraint of the partitioning problem $L_{\max}$. Note that this greedy local search algorithm makes all the easy improvements. Afterwards only few opportunities for local improvements remain so that there is a lower chance for conflicts when we apply a high-quality scheme to improve the partition even further.

Parallel MLS works in iterations. During an iteration, each PE extracts a random vertex from a concurrent producer queue $Q$ that contains all randomly shuffled boundary vertices. 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 continues with the next iteration. Pseudocode of one iteration of the algorithm can be found in Algorithm \ref{algorithm:parallel-mls}. In the paragraphs that follow, we describe how to perform the local moves in $\text{PerformMoves}$ and then how to update the global data structures in $\text{ApplyMoves}$.

#### Performing moves ($\text{PerformMoves}$): Starting from a single boundary vertex, each PE moves vertices to find a sequence of moves that decreases the cut. However, all moves are local; that is, they do not affect 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 the maximum gain and marks it so that other PEs can not 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 $\text{PerformMoves}$: In attempt to achieve better speed-ups, PEs access only one global array to mark vertices and all other accesses occur to local data structures of each PE. Note, that this is the only interaction between PEs. Additionally, we set a bit only once during an iteration and never unset it. Hence, the operation can be seen as priority update operation (see Shun et al. \cite{Shun2011}) and causes only little contention \cite{Shun2011}.

To move a vertex, a PE extracts a vertex $v$ from a local bucket heap (BH), moves it to a block yielding the highest gain and updates the sizes of the affected blocks using a local array. The algorithm then inserts the new block ID of the vertex into a local hash table so that subsequent gain calculations take the move into account. Afterwards, unmarked neighbors of $v$ are inserted into to the BH and marked, and the gain values of the neighbors that are already in the BH are updated. Recall, to mark a vertex, we use a global array. Here, we use a CAS operation to guarantee that a vertex is not marked yet. When the call of $\text{PerformMoves}$ finishes,
we use a global flag to signal to other PEs to stop the computation of PerformMoves. We use an empty local hash table in the beginning of each iteration in order improve the performance of PerformMoves. Also note that since the local hash table is small it often fits into cache which is crucial for parallelization due to memory bandwidth limits.

If one of the PEs finished the PerformMoves routine a global flag is set to stop the other PEs. This ensures that performed work is balanced; our experiments show that this does not affect the quality of partition. Let each PE process $|\mathcal{E}|$ edges and $|\mathcal{V}|$ vertices. 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 $O(\sum_{v \in \mathcal{V}} \sum_{u \in N(v)} d(u) + |\mathcal{V}|) = O(\sum_{v \in \mathcal{V}} d^2(v) + |\mathcal{V}|)$ 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 $O(|\mathcal{E}| \log \Delta + |\mathcal{V}|)$ span. In our experiments, we use the implementation with the former running time since it requires less memory and the worst case; i.e., the gain of a vertex $v$ is updated $d(v)$ times, is quite unlikely.

**Applying moves**: 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 $j$-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 increases 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 than 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. Let $M$ be the set of all moved vertices during this procedure. The overall running time is then given by $O(\sum_{v \in M} d(v))$.

**D. Differences to Mt-Metis**

We now emphasize the differences between our local search algorithms and the hill-climbing technique by Karypis and LaSalle. First, 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 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 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.

**V. Further Optimization**

In this section, we describe 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 \cite{11, 22} for concurrent memory allocations and 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.

**A. 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 MLS 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 guarantees that a pair of similar integer keys is in 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 the tabular hash function \cite{33}. Linear probing typically outperforms other collision resolution techniques in practice and the computing the tabular hash function can be done with small computational overhead. The tabular hash function works as follows. Let $x = x_1 \ldots x_k$ 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 follows:

$$h(x) = T_1[x_1] \oplus \cdots \oplus T_k[x_k].$$

**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 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 PerformMoves procedure.

However, even if the arrays fit into memory this would be problematic. To see this let $|L_2|$ and $|L_3|$ be the sizes of $L_2$...
and L3 caches of a given system, respectively, and let $p'$ be the number of PEs used per a socket. For large graphs, the array may not fit into $\max(\left\lfloor \frac{L \cdot 128}{p'} \right\rfloor)$ memory. In this case, each PE will access its own array in main memory which affects the running time due to the available memory bandwidth. Thus, we want a compact data structure that fits into $\max(\left\lfloor \frac{L}{p} \right\rfloor)$ memory most of the time and preserve the locality guarantees of an array.

For this, we modify the tabular hash function from above according to Mehlhorn and Sanders [28]. More precisely, let $x = x_1 \ldots x_{k-1} x_k$, where $x_k$ are the $t'$ least significant bits of $x$ and $x_1, \ldots, x_{k-1}$ are $t$ bits each. Then we compute the tabular hash function as follows:

$$h(x) = T_1[x_1] \oplus \cdots \oplus T_{k-1}[x_{k-1}] \oplus x_k.$$  

This guarantees that if two keys $x$ and $x'$ differ only in first $t'$ bits and, hence, $|x - x'| < 2^t$ then $|h(x) - h(x')| < 2^t$. Thus, if $t' = O(\log c)$, 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 $k = 3$, $t' = 5$, $t = 10$.

VI. EXPERIMENTS

A. Methodology

We implemented our algorithm Mt-KaHIP within the KaHIP [35] framework using C++ and C++14 multi-threading library. We plan to make our program available in the KaHIP 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 an Intel Xeon E5-2650v2 (2 sockets, 8 cores with Hyper-Threading, 32 threads) running at 2.6 GHz with 128GB 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 with fast social configuration (KaHIP) and ParHIP 2.0 [31] using the fast social configuration (ParHIP). According to LaSalle and Karypis [24] Mt-Metis has better speed-ups and running times compare to ParMetis and similar quality of partitioning. Hence, we do not perform additional experiments with ParMetis.

Our default value for the allowed imbalance is 3% – this is one of the values used in [44]. 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 $k$) basis. When further averaging over multiple instances, we use the geometric mean for quality and time per edge quantities and harmonic mean for relative speed-up in order to give every instance a comparable influence on the final score. 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 $p \in \{1, 8, 16\}$ to save running time and to keep the experimental evaluation simple.

| graph           | $n$  | $m$  | type | ref. |
|-----------------|------|------|------|------|
| amazon          | \(\leq 0.4M\) | \(\leq 2.5M\) | C    | [20] |
| youtube         | \(\approx 1.1M\) | \(\approx 3.0M\) | C    | [20] |
| amazon-2008     | \(\approx 0.7M\) | \(\approx 3.5M\) | C    | [20] |
| in-2004         | \(\approx 1.4M\) | \(\approx 13.6M\) | C    | [20] |
| eu-2005         | \(\approx 0.9M\) | \(\approx 16.1M\) | C    | [20] |
| packing         | \(\approx 2.1M\) | \(\approx 17.5M\) | M    | [20] |
| del23           | \(\approx 8.4M\) | \(\approx 25.2M\) | M    | [20] |
| hugebubbles-00  | \(\approx 18.3M\) | \(\approx 27.5M\) | M    | [20] |
| channel         | \(\approx 4.8M\) | \(\approx 42.7M\) | M    | [20] |
| cage15          | \(\approx 5.2M\) | \(\approx 47.0M\) | M    | [20] |
| europe.osm      | \(\approx 50.9M\) | \(\approx 54.1M\) | M    | [20] |
| enwiki-2013     | \(\approx 4.2M\) | \(\approx 91.9M\) | C    | [20] |
| er-fact1.5-scale23 | \(\approx 8.4M\) | \(\approx 100.3M\) | C    | [20] |
| hollywood-2011  | \(\approx 2.2M\) | \(\approx 114.5M\) | C    | [20] |
| rgg24           | \(\approx 16.8M\) | \(\approx 132.6M\) | M    | [20] |
| rgh             | \(\approx 10.0M\) | \(\approx 199.6M\) | C    | [20] |
| del26           | \(\approx 67.1M\) | \(\approx 201.3M\) | M    | [20] |
| uk-2002         | \(\approx 18.5M\) | \(\approx 261.8M\) | C    | [20] |
| nlpkt240        | \(\approx 28.0M\) | \(\approx 373.2M\) | M    | [20] |
| arabic-2005     | \(\approx 22.7M\) | \(\approx 553.9M\) | C    | [20] |
| rgg26           | \(\approx 67.1M\) | \(\approx 574.6M\) | M    | [20] |
| uk-2005         | \(\approx 39.5M\) | \(\approx 783.0M\) | C    | [20] |
| webbase-2001    | \(\approx 118.1M\) | \(\approx 854.8M\) | C    | [20] |
| it-2004         | \(\approx 41.3M\) | \(\approx 1.0G\) | C    | [20] |

TABLE I

Basic properties of the benchmark set with a rough type classification. C stands for complex networks, M is used for mesh-type networks.

We use performance plots to present the quality comparison 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 $1 - \frac{\text{best alg}}{\text{best cut obtained by any of the considered algorithms and cut is the cut of algorithm X}}$. 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.

Algorithm Configuration: Any multilevel 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, respectively, partitions a coarse graph 10 times in initial partitioning and uses ten iterations of parallel MLS in the refinement phase.

Instances: We evaluate our algorithms on a number of large graphs. These graphs are collected from [3], [11], [6], [26], [43], [4]. Table I summarizes the main properties of the benchmark set. Our benchmark set includes a number of graphs from numeric simulations as well as complex networks (for the latter with a focus on social networks and web graphs).

The $\text{rgg}$ graph is a complex network generated with NetworKit [43] according to the random hyperbolic graph model [21]. 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 $\text{rgg}$ and $\text{del}$ for comparisons. $\text{rgg}$ 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. \(\text{del}X\) is a Delaunay triangulation of \(2^N\) random points in the unit square. The graph \(\text{er-fact1.5-scale23}\) is generated using the Erdős-Rényi \(G(n,p)\) model with \(p = 1.5\ln n/n\).

**B. Quality Comparison**

In this section, we compare our algorithm against competing state-of-the-art algorithms in terms of quality. The performance plot on Figure 2 shows the results of our experiments for all of our benchmark graphs shown in Table I.

Our algorithm gives the best overall quality, usually producing the overall best cut. Even in the small fraction of instance where other algorithms are best, our algorithm is at most 15% off. The overall solution quality does not heavily depend on the number of PEs used. Indeed, more PEs give slightly higher partitioning quality since more initial partition attempts are done in parallel. 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 MLS significantly improves solution quality. Mt-Metis with \(p = 1\) has worse quality than our algorithm almost on all instances. The exceptions are seven mesh type networks and one complex network but our algorithm is at most 15% off. 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. For our algorithm they are 721.1K, 715.9K and 711.8K for \(p = 1, 8, 16\), respectively. For Mt-Metis they are 819.8K, 867.0K and 874.7K for \(p = 1, 8, 16\), respectively. For ParHIP they are 810.4K, 797.9K and 809.5K for \(p = 1, 8, 16\), respectively, and for KaHIP it is 766.2K. Significance tests that we run indicate that the quality advantage of our solver over the other solvers is statistically significant.

**C. Speed-up and Run Time Comparison**

In this section, we compare the speed-ups and the running times of our algorithm against competing algorithms. We calculate a relative speed-up of an algorithm as a ratio between its running time (averaged over ten repetitions) and its running time with \(p = 1\). Figure 3 show scatter plots with speed-ups and time per edge for a full algorithm execution and local search (for our algorithm it is MLS). Additionally, we calculate the harmonic means only for instances that were partitioned in ten repetitions without imbalance. Note that among top 15 speed-ups of Mt-Metis more than 70% correspond to the imbalanced instances (Mt-Metis 16 imbalanced) thus we believe it is fair to exclude them.

The harmonic mean full speed-up of our algorithm, Mt-Metis and ParHIP for \(p = 16\) are 6.93, 7.67 and 4.98, respectively. The harmonic mean local search speed-up of our algorithm, Mt-Metis and ParHIP are 9.33, 5.21 and 3.52, respectively. As you can see our full speed-ups are similar 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 77.3 nanoseconds (ns), 15.9 [ns] and 192.9 [ns], respectively. The geometric mean local search time per edge of our algorithm, Mt-Metis and ParHIP are 6.7 [ns], 2.9 [ns] and 29.9 [ns], respectively. Note that with the increase of the 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.

**VII. CONCLUSION AND FUTURE WORK**

Graph partitioning is a key prerequisite for efficient large-scale parallel graph algorithms. We presented an approach to multi-level shared-memory parallel graph partitioning that guarantees balanced solutions, shows high speedups for a variety of large graphs and yields very good quality independently of the number of cores used. Previous approaches have problems with recently grown structural complexity of networks that need partitioning – they often show a negative trade-off between speed and quality. Important ingredients of our algorithm include parallel label propagation for both coarsening and improvement, parallel initial partitioning, a simple yet effective approach to parallel localized local search, and fast locality preserving hash tables. Considering the good results of our algorithm, we want to further improve and
release its implementation. More precisely, we are planning to further improve scalability of parallel coarsening and parallel MLS. An interesting problem is how to apply moves in Section IV-C without the gain recalculation. The solution of this problem will increase the performance of parallel MLS. Additionally, we are planning to integrate a high quality parallel matching algorithm for the coarsening phase that allows to receive better quality for mesh-like graphs. Further quality improvements should be possible by integrating a parallel version of the flow based techniques used in KaHIP.

ACKNOWLEDGMENT

We thank Dominique LaSalle for helpful discussions about his hill-climbing local search technique.
REFERENCES

[1] Intel threading building blocks, https://www.threadingbuildingblocks.org/.

[2] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, In-place parallel super scalar samplesort (ipssoso), Proc. of the 25th ESA, 2017, pp. 9:1–9:14.

[3] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, Benchmarking for Graph Clustering and Partitioning, Encyclopedia of Social Network Analysis and Mining, 2014, pp. 73–82.

[4] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner (eds.), 10th dimacs implementation challenge – graph partitioning and graph clustering, 2013.

[5] C. Bichot and P. Siarry (eds.), Graph partitioning, Wiley, 2011.

[6] P. Boldi and S. Vigna, The WebGraph framework I: Compression techniques, Proc. of the 13th Int. World Wide Web Conference, 2004, pp. 595–601.

[7] T. Nguyen Bui and C. Jones, Finding good approximate vertex and edge partitions is np-hard, Information Processing Letters 42 (1992), no. 3, 153–159.

[8] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, Recent Advances in Graph Partitioning, Algorithm Engineering – Selected Topics, to app., ArXiv:1311.3144, 2014.

[9] U. V. Catalyurek and A. Aykanat, Hypergraph-partitioning based Decomposition for Parallel Sparse-Matrix Vector Multiplication, IEEE Transactions on Parallel and Distributed Systems 10 (1999), no. 7, 673–690.

[10] H. Meyerhenke, Shape optimizing load balancing for mpi-parallel datasets

[11] T. Davis, The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, 2008.

[12] C. M. Fiduccia and R. M. Mattheyses, A Linear-Time Heuristic for Improving Network Partitions, 19th Conference on Design Automation, 1982, pp. 175–181.

[13] M. R. Garey, D. S. Johnson, and L. Stockmeyer, Some simplified np-complete problems, Proc. of the 6th ACM STOC, 1974, pp. 47–63.

[14] M. Holtgrewe, P. Sanders, and C. Schulz, Engineering a Scalable High Quality Graph Partitioner, Proc. of the 24th IPDS (2010), 1–12.

[15] J. D. Hunter, Matplotlib: A 2d graphics environment, Computing In Science & Engineering (2007), 90–95.

[16] L. Hyafil and R. L. Rivest, Graph partitioning and constructing optimal decision trees are polynomial complete problems, IRIA. Laboratoire de Recherche en Informatique et Automatique, 1973.

[17] G. Karypis and V. Kumar, Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs, Proceedings of the ACM/IEEE Conference on Supercomputing, 1996.

[18] ____, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing (1998), 359–392.

[19] ____, Multilevel-k-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed computing (1998), 96–129.

[20] B. W. Kernighan, Some graph partitioning problems related to program segmentation, Ph.D. thesis, Princeton, 1969.

[21] D. Kreouk, F. Papadopoulos, M. Kitsak, A. Valdat, and M. Boguñá, Hyperbolic geometry of complex networks, Physical Review E (2010), 036106.

[22] A. Kukanov and M. J. Voss, The foundations for scalable multi-core software in intel threading building blocks, Intel Technology Journal (2007), 309–322.

[23] University of Milano Laboratory of Web Algorithms, Datasets.

[24] D. LaSalle and G. Karypis, Multi-threaded graph partitioning, Proc. of the 27th IPDPS, 2013.

[25] ____, A parallel hill-climbing refinement algorithm for graph partitioning, Proc. of the 45th ICPP, 2016, pp. 236–241.

[26] J. Leskovec, Stanford Network Analysis Package (SNAP).

[27] T. Maier, P. Sanders, and R. Dementiev, Concurrent hash tables: Fast and general?, Proc. of the 21st PLDI, 2016, p. 34.

[28] K. Mehlhorn and P. Sanders, Algorithms and data structures — the basic toolkit, Springer, 2008.

[29] H. Meyerhenke, Shape optimizing load balancing for mpir-parallel adaptive numerical simulations, Proc. of the 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, Cont. Mathematics, AMS, 2013.

[30] H. Meyerhenke, P. Sanders, and C. Schulz, Partitioning complex networks via size-constrained clustering, Proc. of the 13th SEA, 2014, pp. 351–363.

[31] ____, Parallel graph partitioning for complex networks, IEEE Transactions on Parallel and Distributed Systems (2017), 2625–2638.

[32] V. Osipov and P. Sanders, n-level graph partitioning, Proc. of the 18th ESA, 2010, pp. 278–289.

[33] M. Patrascu and M. Thorup, The power of simple tabulation hashing, Proceedings of the 43rd ACM STOC, 2011, pp. 1–10.

[34] U. N. Raghavan, R. Albert, and S. Kumara, Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks, Physical review E 76 (2007), no. 3, 036106.

[35] P. Sanders and C. Schulz, Engineering Multilevel Graph Partitioning Algorithms, Proc. of the 19th ESA, LNCS, vol. 6942, Springer, 2011, pp. 469–480.

[36] J. E. Savage and M. G. Wikão, Parallelism in graph-partitioning, Journal of Parallel and Distributed Computing (1991), 257–272.

[37] K. Schloegel, G. Karypis, and V. Kumar, Graph Partitioning for High Performance Scientific Simulations, The Sourcebook of Parallel Computing, 2003, pp. 491–541.

[38] A. Shalita, B. Karrer, I. Kabiljo, A. Sharma, A. Presta, A. Adcock, H. Kilapi, and M. Stumm, Social hash: An assignment framework for optimizing distributed systems operations on social networks., NSDI, 2016, pp. 455–468.

[39] J. Shun, G.E. Blelloch, J. T. Fineman, and P. B. Gibbons, Reducing contention through priority updates, Proc. of the 25th SPAA, 2013, pp. 152–163.

[40] J. Singlet, P. Sanders, and F. Putze, Mxtl: The multi-core standard template library, Proc. of the 13th Euro-Par (2007), 682–694.

[41] C. L. Staudt and H. Meyerhenke, Engineering parallel algorithms for community detection in massive networks, IEEE Transactions on Parallel and Distributed Systems 27 (2016), no. 1, 171–184.

[42] J. Ugander and L. Backstrom, Balanced Label Propagation for Partitioning Massive Graphs, Proc. of 6th WSDM, 2013, pp. 507–516.

[43] M. von Looz, H. Meyerhenke, and R. Prutkin, Generating random hyperbolic graphs in subquadratic time, Proc. of the 26th ISAAC, 2015, pp. 467–478.

[44] C. Walshaw and M. Cross, Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm, SIAM Journal on Scientific Computing 22 (2000), no. 1, 63–80.

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