(Semi-)External Algorithms for
Graph Partitioning and Clustering

Yaroslav Akhremtsev, Peter Sanders and Christian Schulz
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
Email: {*yaroslav. akhremtsev*, *sanders*, *christian. schulz*}*@kit. edu*

Abstract

In this paper, we develop semi-external and external memory algorithms for graph partitioning and clustering problems. Graph partitioning and clustering are key tools for processing and analyzing large complex networks. We address both problems in the (semi-)external model by adapting the size-constrained *label propagation* technique. Our (semi-)external size-constrained label propagation algorithm can be used to compute graph clusterings and is a prerequisite for the (semi-)external graph partitioning algorithm. The algorithm is then used for both the coarsening and the refinement phase of a multilevel algorithm to compute graph partitions. Our algorithm is able to partition and cluster huge complex networks with billions of edges on cheap commodity machines. Experiments demonstrate that the semi-external graph partitioning algorithm is scalable and can compute high quality partitions in time that is comparable to the running time of an efficient internal memory implementation. A parallelization of the algorithm in the semi-external model further reduces running time.

1 Introduction

Graph partitioning and clustering problems are often solved to analyse or process large graphs in various contexts such as social networks, web graphs, or in scientific numeric simulations. To be able to process huge unstructured networks on cheap commodity machines one can rely on graph partitioning and partition the graph under consideration into a number of blocks such that each block fits into the internal memory of the machine while edges running between blocks are minimized (see for example [1]). However, to do so the partitioning algorithm itself has to be able to partition networks that do not fit into the internal memory of a machine. In this paper, we present semi-external and external algorithms for the graph partitioning problem that are able to compute high quality solutions.

It is well known that graph partitioning is NP-complete [2] and that there is no constant factor approximation algorithm for general graphs [3]. Hence, mostly heuristics are used in practice. Probably the most-commonly used heuristic is the *multilevel graph partitioning* (MGP) scheme. Here, the graph is recursively *contracted* to obtain a sequence of smaller graphs with similar properties as the input graph. As soon as the graph is small enough, an *initial partitioning algorithm* partitions the coarsest graph. Afterwards the contraction is undone and on each level a local search algorithm is used to improve the quality of the partition.

The paper is organized as follows. After introducing basic concepts and related work in Sections 2, we present the key concepts and a rough overview of our technique in Section 3. Section 4 describes different algorithms to compute clusterings with or without a size-constraint in both computational models and a parallelization in the semi-external model. Moreover, it presents the *first* external memory algorithm to tackle the graph partitioning problem. Subsequently, Section 5 explains how clusterings can be

used to build a graph hierarchy to be used in a multi-level algorithm in the (semi-)external model. Experiments to evaluate the performance of our algorithms are presented in Section 6. Finally, we conclude in Section 7.

2 Preliminaries

2.1 Basic Concepts. We consider an undirected weighted graph $G = (V, E, c, w)$, with node weights $c : V \mapsto \mathbb{R}_{>0}$ and edge weights $w : E \mapsto \mathbb{R}_{>0}$. To keep our analysis simple, we assume that $|E| \geq |V|$. If the graphs are unweighted, we assume unit edge and node weights. The set $N(v) := \{u \mid \{v, u\} \in E\}$ denotes the neighbors of a node v.

Given a number $k > 1$, the *graph partitioning* problem asks to find blocks of nodes $V_1, \ldots, V_k$ such that $V_1 \cup \ldots \cup V_k = V$ and $\forall i \neq j : V_i \cap V_j = \emptyset$. A *balance* constraint demands that $|V_i| \leq L_{max} := (1 + \varepsilon)\lceil |V|/k \rceil \ \forall i \in \{1, \ldots, k\}$, for some imbalance parameter $\varepsilon \geq 0$. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u, v\} \in E \mid u \in V_i, v \in V_j\}$. Throughout the paper cluster$_G[v]$ denotes the block/cluster of a node v. We omit the subscript G if the context allows it. A *graph clustering* is also a partition of the nodes, however, k is not given in advance and there is no size-constraint. A size-constrained clustering constrains the size of the clusters by a given upper bound U such that $|V_i| \leq U$. We say that a block/cluster V_i is *underloaded* if $|V_i| < L_{\max}$ and *overloaded* if $|V_i| > L_{\max}$. Given a graph clustering $V_1, \ldots, V_k$, the *quotient graph* is defined as $\mathcal{Q} = (V_q, E_q, c_q, w_q)$, where $V_q = \{1, \ldots, k\}$ and $E_q = \{(i, j) \mid E_{ij} \neq \emptyset\}$, $w_q(i, j) = \sum_{e \in E_{ij}} w(e)$, $c_q(i) = \sum_{v \in V_i} c(v)$, $i, j \in V_q$.

2.1.1 Computational Models. We look at two models: the external and the semi-external model [4]. In both models, one wants to minimize the number of I/O operations. In the external model it is assumed that the graph does not fit into internal memory whereas the semi-external memory assumes that there is enough memory for the nodes of the graph to fit into internal memory, but not enough for the edges. We will use the following notations: M is the size of internal memory, B is the size of a disk block, $\mathcal{O}(\frac{N}{B}) = \mathrm{Scan}(N)$ is the number of I/O operations needed for reading or writing an array of size N and $\mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}) = \mathrm{Sort}(N)$ is the number of I/O operations needed for sorting an array of size N.

2.1.2 Graph Data Structure. To store a graph in external memory, we use a data structure similar to an adjacency array. This data structure allows us to inspect all edges using Scan($|E|$) I/O operations. An external array of the edges contains the adjacency lists of each node in increasing order of their IDs. Each element of the adjacency list of a node u is a pair (v, w), where v is the target of the edge (u, v) and $w = w(u, v)$ is the weight of the edge. We mark the end of each adjacency list by using a sentinel pair. This allows us to determine easily if we reached the end of the adjacency list of the node that we currently process. The second external array stores node offsets, i.e. for each node we store a pointer to the beginning of its adjacency list in the edge array. The third external array contains the weights of the nodes.

2.2 Related Work. There has been a *huge* amount of research on graph partitioning so that we refer the reader to [5,6] for most of the material. Here, we focus on issues closely related to our main contributions. All general-purpose methods that work well on large real-world graphs are based on the multilevel principle. The basic idea can be traced back to multigrid solvers for systems of linear equations. Recent practical methods are mostly based on graph theoretic aspects, in particular edge contraction and local search. There are different ways to create graph hierarchies such as matching-based schemes (e.g. [7]) or variations thereof [8] and techniques similar to algebraic multigrid (e.g. [9]). Well-known MGP software packages include Chaco [10], Jostle [11], Metis [12], Party [13] and Scotch [14].

Graph clustering with the label propagation algorithm (LPA) has originally been described by Raghavan et al. [15]. Meyerhenke et al. [16] introduced a size-constrained LPA. Doing so made it possible to use this algorithm during coarsening and uncoarsening of a multilevel scheme to compute graph partitions of large complex networks. In this paper, we present a semi-external and an external variant of this algorithm. There are other semi-external algorithms to tackle the graph partitioning problem which are based on streaming [17]. However, they do not use a multilevel approach and do not achieve high solution quality. To the best of our knowledge, our algorithm is the first that tackles the graph partitioning problem in the external memory model.

2.2.1 KaHIP (Karlsruhe High Quality Partitioning) is a family of graph partitioning algorithms that tackle the balanced graph partitioning problem [18]. It includes several multilevel algorithms and meta-heuristics to compute high quality partitions. In particular, the algorithms of Meyerhenke et al. [16] to partition large complex networks are included. We use KaHIP to partition the coarse graphs as soon as they fit into the internal memory of the machine.

3 Basic Cluster Contraction

We now present the basic idea and explain the main ingredients which are used to obtain a (semi-)external graph partitioning algorithm. First of all, we define a (semi-)external algorithm to create graph hierarchies for the multilevel graph partition approach. The approach of Meyerhenke et al. [16] to create graph hierarchies in internal memory iteratively contracts size-constraint graph clusterings that are obtained using a *label propagation algorithm*. *Contracting a clustering* works as follows: each block of the clustering is contracted into a single node. The weight of the node is set to the sum of the weight of all nodes in the original block. There is an edge between two nodes u and v in the contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G, i.e. block u and block v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weight of edges that run between block A and block B of the clustering. Due to the way contraction is defined, a partition of the coarse graph corresponds to a partition of the finer graph with the same cut and balance. Note that the contracted graph corresponds to the quotient graph.

Cluster contraction is an aggressive coarsening strategy. In contrast to most previous approaches, it can drastically shrink the size of irregular networks. The intuition behind this technique is that a clustering of the graph (one hopes) contains many edges running inside the clusters and only a few edges running between clusters, which is favorable for the edge cut objective. Regarding complexity, experiments in [16] indicate that already one contraction step can shrink the graph size by orders of magnitude and that the average degree of the contracted graph is smaller than the average degree of the input network. Thus it is very likely that the graph will fit into internal memory after the first contraction step. On the other hand, by using a different size-constraint ($|V_i| \leq L_{\max}$), the LPA can also be used as a simple strategy to improve a solution on the current level.

Our *main idea* to obtain a (semi-)external mul-

tilevel graph partitioning algorithm is to engineer (semi-)external variants of the size-constrained LPA and to externalize the contraction as well as solution transfer component of the algorithm. By doing this, we have a (semi-)external algorithm to build graph hierarchies and to transfer solutions to finer levels. Once the graph is small enough to fit into internal memory, we use the KaHIP framework to compute a partition of the graph. Additionally, we use a (semi-)external size-constrained LPA as a local search algorithm to improve the solution on the finer levels that do not fit into internal memory. We proceed by explaining the (semi-)external size-constrained LPA as well as its parallelization in Section 4 and explain the external contraction and solution transfer algorithm in Section 5.

4 (Semi-)External Graph Clustering

We now explain how graph clusterings can be obtained in both, the semi-external and the external memory model. We present multiple algorithms: a semi-external LPA that can deal with size-constraints, an external LPA that does not use size-constraints, as well as a coloring-based graph clustering algorithm inspired by label propagation that can also maintain size-constraints in the external model.

4.1 Label Propagation (LP) was first presented by Raghavan et al. [15]. In the beginning each node belongs to its own cluster. Afterwards the algorithm works in rounds. In each round, the algorithm visits each node in increasing order of their IDs. When a node v is visited, it is *moved* to the block that has the strongest connection to v, i.e. it is moved to the cluster V_i that maximizes $\omega(\{(v, u) \mid u \in N(v) \cap V_i\})$. Ties are broken randomly. If the algorithm is used to compute a size-constrained clustering, the selection rule is modified such that only moves are eligible that do not result in overloaded blocks.

Suppose the algorithm is currently processing a node v. Now we scan the adjacency list of the node v in the respective memory model and compute the new cluster of this node. If a size-constraint is present, we also need a scheme to manage the sizes of each cluster/block.

4.1.1 Semi-External Label Propagation. This is the simple case: since we have $\mathcal{O}(|V|)$ internal memory, we can afford to store the cluster IDs in internal memory. Additionally, we maintain an array of size $|V|$ in internal memory that stores the cluster sizes. Hence, one iteration of the semi-

external LP algorithm can be done using Scan($|E|$) I/O operations.

4.1.2 Parallel Semi-External Label Propagation.

Recall that the LP algorithm iterates through the external array of edges. In attempt to accelerate semi-external LPA and to get closer to the I/O bound, we parallelize the processing of a disk block of edges. Since, the LP algorithm processes nodes interdependently, we divide the disk block into equal ranges and process them in parallel. We now explain how we process nodes which have adjacency lists that belong to different disk blocks (see Figure 1 for an example). Each thread t begins to process its range $[begin_t; end_t)$ of the disk block. Afterwards, the range is shifted such that each adjacency list in the block is processed by precisely one thread. Consider the example depicted in Figure 1. Here, the thread finds the end of the adjacency list 1 in $[begin_t; end_t)$ and iterates through the elements until the end of adjacency list 2 is reached. The colored area in Figure 1 represents the range that will be actually processed by thread t.
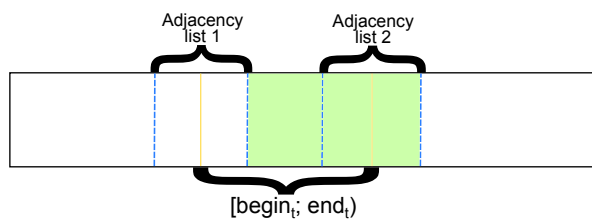

Figure 1: Range processed by a thread

To maintain up-to-date cluster sizes, we do not move a node immediately, i.e. we store the moves which were generated by the threads during the processing of the disk block. Afterwards, all moves are processed sequentially and we make a move if it does not violate the size constraint.

4.1.3 External Label Propagation.

To propagate the cluster IDs of adjacent nodes, we use time forward processing [19]. More precisely, we maintain two external priority queues [20]: one for the current and one for the next round. Initially, the current queue contains triples (v, c, w) for each edge $(u, v) \in E : v < u$ where v is the key value, w denotes the weight of the edge and c is the current cluster ID of u. When the algorithm scans a node u, all triples (u, c, w) are on top of the priority queue since the nodes are processed in increasing order of their ID. The tuples are then extracted using the operations POP and TOP. This means, we know the current

cluster ID of all adjacent nodes of u and can calculate the new cluster ID. When the new cluster ID is computed, the algorithm pushes triples with the new cluster ID for all adjacent nodes into the next and current priority queue depending on the node ID of the neighbor v: if $u < v$ we push $(v, \text{cluster}[u], w(u, v))$ to the current priority queue and if $v < u$ we push it to the next priority queue. At the end of a round we swap the priority queues.

Each operation of the priority queue POP, PUSH and TOP is called $\mathcal{O}(|E|)$ times. Each of the operations can be done using $\mathcal{O}(\frac{1}{B} \log_{\frac{M}{B}} \frac{|E|}{B})$ I/O operations amortized [20, 21]. Thus, the overall algorithm uses $\mathcal{O}(\frac{|E|}{B} \log_{\frac{M}{B}} \frac{|E|}{B}) = \text{Sort}(|E|)$ I/O operations.

4.2 Coloring-based Graph Clustering.

We now present another approach to compute a graph clustering in the external model. The algorithm is able to maintain the sizes of all clusters in the external memory model. The *main idea* of the algorithm is to process *independent sets*. Due to the definition of an independent set, a change of the cluster ID of a node will not have an effect on the other nodes within the set. However, the changes of all adjacent nodes of v need to be taken into account.

Assume that we have a node coloring $C = \{C_1, C_2, \ldots, C_\ell\}$ of the graph, where C_i is the set of vertices with the same color i. Note that each set C_i forms an independent set. For each set C_i, we maintain an external array (bucket) of tuples $\mathcal{T}_i$ and allocate a buffer of size B internal memory for each bucket $\mathcal{T}_i$. Hence, we assume that the number of colors $|C|$ is smaller than $\frac{M}{B}$.

The bucket clustering algorithm also works in rounds. Roughly speaking, in each round it processes the buckets in increasing order of their color and updates the cluster IDs of all nodes. When we process a bucket we need the cluster IDs of all adjacent nodes. Hence, we define the content of a bucket as follows: we store tuples so that we know the current cluster ID of each adjacent neighbor, its color, ID of the neighbor and the weight of the edge. More precisely, initially for a node u, we store the following tuples for all adjacent nodes v with color$[v] <$ color$[u]$ in the corresponding buckets $\mathcal{T}_{\text{color}[v]}$: $(v, \text{cluster}[u], u, w(u, v), \text{color}[u])$. To do this efficiently, we augment the array of edges by adding the color of the target node v to each edge (u, v) before the initialization step. Note that these tuples contain the complete information about the graph structure and that the information suffices to update the cluster of a node.

When the algorithm processes a bucket $\mathcal{T}_i$, it sorts the elements of the bucket lexicographically by the first and second component. Afterwards, it scans the tuples of the bucket and calculates a new cluster ID for each node in C_i in the same manner as the LPA. When the bucket is processed, we push tuples with the new cluster IDs to the corresponding bucket, i.e. for each tuple $(v, \text{cluster}[u], u, w(v, u), \text{color}[u])$ in bucket $\mathcal{T}_i$, we push the tuple $(u, \text{cluster}[v], v, w(u, v), \text{color}[v])$ into the bucket $\mathcal{T}_{\text{color}[u]}$.

LEMMA 4.1. *Processing a bucket $\mathcal{T}$ can be done in $Sort(|\mathcal{T}|)$ I/O-operations.*

Proof. For sorting the bucket, we need $\text{Sort}(|\mathcal{T}|)$ I/O operations. We need $\text{Scan}(|\mathcal{T}|)$ I/O operations for scanning the bucket. Hence, the algorithm uses $\text{Sort}(|\mathcal{T}|)$ I/O operations.

THEOREM 4.1. *The bucket algorithm requires $Sort(|E|)$ I/O-operations for one iteration of label propagation.*

Proof. Adding the information about the colors of target nodes to the edges requires $\text{Sort}(|E|)$ I/O operations. Our bucket initialization uses $\text{Scan}(|E|)$ I/O operations. Suppose we have the buckets $\mathcal{T}_1, \ldots, \mathcal{T}_\ell$. All buckets can be processed using $\text{Sort}(|\mathcal{T}_1|) + \ldots + \text{Sort}(|\mathcal{T}_\ell|) = \text{Sort}(|E|)$ I/O operations. Overall, we use $\text{Scan}(|E|)$ I/O operations for pushing tuples with the new cluster IDs into the respective buckets. Hence, the algorithm can be implemented using $\text{Sort}(|E|)$ I/O operations.

4.2.1 Graph Coloring. Computing the graph coloring is a very important part of the bucket graph clustering algorithm. Note that the number of colors is equal to the number of buckets and we want to maintain as few buckets as possible. This is due to the fact that we need an amount of B space for each bucket in internal memory. Moreover, the size of each bucket must be smaller than an upper bound, since each bucket has to fit into internal memory during our experiments. To compute a coloring, we use the time forward processing technique [19] with an additional size-constraint on the color classes that can be maintained in internal memory. This allows us to build a coloring using $\text{Sort}(|E|)$ I/O operations. Note that the coloring is computed only once so that the cost for computing the coloring can be amortized over many iterations of label propagation.

4.2.2 External Graph Clustering Algorithm with Size-Constraints. In this section we describe how we modify the coloring-based clustering algorithm, so that it can handle a size-constraint. The main advantage of the coloring-based clustering algorithm is as follows. When we process a bucket, the cluster IDs of all adjacent nodes will not change. This allows us to maintain a data structure with up-to-date sizes of the clusters of the nodes of the independent set and their neighbors. In the following, we consider two different data structures depending on if each of the buckets fits into internal memory or not. In both cases, we use an external array that stores the sizes of all clusters. We start by explaining the case where each bucket fits into internal memory.

Case A: each bucket fits into internal memory. In this case, we can use a hash table $\mathcal{H}$ to maintain the cluster sizes of the current bucket. The key of $\mathcal{H}$ is the cluster ID and the value is the current size of the cluster. When we process a bucket $\mathcal{T}_i$, the hash table $\mathcal{H}$ can be built as follows. We collect all cluster IDs of the nodes of the current independent set as well as their neighbors, sort them and then iterate through the external array to get the current cluster sizes. After finishing to calculate a new cluster ID for each node in C_i, we write the updated cluster sizes to the external array. Hence, the cluster sizes are up-to-date after we processed the current bucket.

THEOREM 4.2. *The coloring-based clustering algorithm with size-constraints uses $t \cdot Scan(|V|) + Sort(|E|)$ I/O operations, where $t = \max(\frac{|E|}{M}, |C|)$ is the amount of buckets such that each bucket fits into internal memory.*

Proof. First, we prove the complexity to create the data structure containing the clusters sizes. In the worst case, each tuple $(v, \text{cluster}[u], u, w(v, u), \text{color}[u])$ of the bucket $\mathcal{T}$ has a unique cluster ID and also the cluster IDs of each node v is unique. Hence, we need an additional amount of $\mathcal{O}(|\mathcal{T}|)$ internal memory for the hash table. For the saving and sorting the cluster IDs from the bucket we use $\mathcal{O}(1)$ I/O operations since we have the bucket in internal memory. Reading and writing the sizes of the clusters to and from the external array uses $\text{Scan}(|V|)$ I/O-operations.

Now we estimate the amount of buckets that fit into internal memory. There are two cases. If a bucket does not fit into internal memory, we need to divide it into multiple buckets. Since the

overall size of all buckets is $\mathcal{O}(|E|)$ the minimum amount of buckets (such that each fits into internal memory) is $\mathcal{O}(\frac{|E|}{M})$. Otherwise, if all buckets fit into internal memory, we have $|C|$ buckets. Since we want each bucket to fit into internal memory, we have $\max(\frac{|E|}{M}, |C|)$ buckets.

The overall I/O-volume is estimated as follows: for each bucket we need $\text{Scan}(|V|)$ additional I/O-operations to create the data structure containing the sizes of the clusters. There are at most $\max(\frac{|E|}{M}, |C|)$ buckets. Hence, the total number of I/O-operations is $\max(\frac{|E|}{M}, |C|) \cdot \text{Scan}(|V|) + \text{Sort}(|E|)$ (to perform the main part of the bucket clustering algorithm).

Case B: there is at least one bucket that does not fit into internal memory. This case is somewhat more complicated, since we cannot afford to store the hash table in internal memory. Basically, when we process a bucket $\mathcal{T}_i$, we do not use a hash table but an external priority queue and additional data structures which contain enough information to manage the cluster sizes. More precisely, we define a structure $\mathcal{M}$ that tells us for each node which nodes need the updated cluster size information. Nodes from C_i are still processed in increasing order of their IDs. We now explain the structures in detail.

For a node v in the current independent set C_i, let $C(v) := \{\text{cluster}[u] \mid (v, u) \in E\} \cup \{\text{cluster}[v]\}$ denote the set of adjacent clusters. An example is shown in Figure 2. These are the clusters that can possibly change their size if v changes its cluster. We now need to find all nodes from the independent set that are adjacent to the clusters or are contained in this cluster because they need to receive the updated cluster size.

The first additional data structure contains only nodes of the independent set. It is needed to build the next data structure $\mathcal{M}$. Let $N_c := \{u \in C_i \mid \exists (u, v) \in E : \text{cluster}[v] = c\} \cup \{u \in C_i \mid \text{cluster}[u] = c\}$ be the set of adjacent nodes for a cluster c that are in the current independent set (including the nodes that are in the cluster). We sort N_c in increasing order of node IDs and remove repeated elements. For a cluster c, the set N_c contains all nodes from the independent set that are adjacent to the cluster. Moreover, the order in N_c is similar to the processing order of the independent set. We denote the j-th node of N_c as N_c^j. The second additional data structure uses the first one and is defined as the set $M_v := \{(u, c) \mid c \in C(v), N_c^j = v, N_c^{j+1} = u\}$. It contains the nodes to which the node v must forward information about the changes in the cluster sizes. Roughly speaking, for each cluster in the neighborhood of v (including
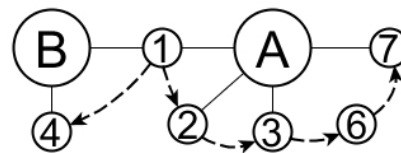

Figure 2: In this example, we have $C(1) = \{A, B, \text{cluster}[1]\}$ and $C(4) = \{B, \text{cluster}[4]\}$. Dotted lines denote forwarding cluster size changes. A, B are cluster IDs, 1-7 denote node ID. Node 6 belongs to cluster A. The sets of adjacent nodes for the cluster A and B are $N_A = \{1, 2, 3, 6, 7\}$ and $N_B = \{1, 4\}$. Moreover, $M_1 = \{(2, A), (4, B)\}$, $M_2 = \{(3, A)\}$.

the cluster of v), M_v contains the adjacent node of the cluster that will be processed next. This way the information can be propagated easily. An example is shown in Figure 2. We now explain the details of the algorithm when processing one bucket. First, we compute the sets N_c. To do so, we build a list N of pairs (c, v) that are sorted lexicographically by their first and second component, where c is the ID of the cluster adjacent to v. For building this list, we iterate through the bucket and add pairs (c, v) for each tuple $(v, \ldots) \in \mathcal{T}_i$ to the list and also add the pair $(\text{cluster}[v], v) \; \forall v \in C_i$. Then we sort these pairs and we are done. Note that $|N| = |C_i| + |\mathcal{T}_i| = \mathcal{O}(|\mathcal{T}_i|)$. To compute the sets M_v, we build a list $\mathcal{M}$ of triples (v, c, u), where v is the node ID and $(u, c) \in M_v$. For each $N_c^j = v$ and $N_c^{j+1} = u$ the triple (v, c, u) is added to the list. Afterwards, the triples are sorted by the first component. Note that the size of the list is at most $\mathcal{O}(|\mathcal{T}_i|)$.

Recall, that the set M_v contains the nodes that have to receive the changes in the cluster sizes. To forward the information, we use an external priority queue. The priority queue contains triples (v, c, sz), where v is the node ID which also serves as key value, $c \in C(v)$ is the cluster and sz the size of the cluster. We initialize the priority queue as follows: we iterate through the sets N_c and put the tuples (v_1, c, sz) in the priority queue, where v_1 is the first node in N_c. The sizes of the clusters are obtained from the external array containing the cluster sizes. Then the nodes are processed. After node v is processed, we put (u, c, sz) in the priority queue for each pair $(u, c) \in M_v$. After we processed a bucket, we update the cluster sizes in the external array.

LEMMA 4.2. *When node v is processed there is a triple (v, c, sz) for each adjacent cluster on the top of priority queue with up-to-date cluster sizes.*

Proof. Consider a cluster ID c and let the list N_c be $\{v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_k\}$. In the beginning, there is triple $(v_1, c, size)$ in the priority queue with the actual size of the cluster c due way the priority queue is initialized. When we process v_i, we put the triple with the updated size into the priority queue for *the* pair (v_{i+1}, c) in the list M_{v_i}. Hence, when we process v_{i+1} the size of the cluster will also be up-to-date (since it is the next adjacent node of cluster c being processed). When v is processed the triples with key value v are on top of the priority queue, due to the increasing processing order.

LEMMA 4.3. *Processing a bucket and maintaining cluster sizes costs Sort($|\mathcal{T}|$) I/O-operations.*

Proof. We need Sort($|\mathcal{T}|$) I/O-operations to sort the bucket and to build the lists N and $\mathcal{M}$. The operations POP and PUSH of the priority queue have amortized $\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}$ cost. The number of PUSH (or POP) operations is equal to $|\mathcal{M}|$. This is due to the fact that each element of this list means that two nodes in the bucket have to take the size of the same cluster into account. Thus, we need to forward the information from the first to the second node. This means that the total cost of all operations is Sort($|\mathcal{T}|$). Iterating through the bucket costs Scan($|\mathcal{T}|$). Hence, processing a bucket costs Sort($|\mathcal{T}|$) in total.

THEOREM 4.3. *One iteration of the coloring-based clustering algorithm costs Sort($|E|$) + $|C| \cdot$ Scan($|V|$) operations, where $|C|$ is the number of buckets.*

Proof. We have $|C|$ buckets. Processing a bucket costs Sort($|\mathcal{T}|$) I/Os. The overall number of elements in the buckets is $\mathcal{O}(|E|)$. Hence, processing all buckets takes Sort($|E|$) I/O-operations. After we processed a bucket, we update the sizes of the clusters that changed during processing. This takes Scan($|V|$) I/O-operations. Thus, one iteration of the bucket clustering algorithm costs Sort($|E|$) + $|C| \cdot$ Scan($|V|$) I/O-operations.

5 (Semi-)External MGP

We now explain the (semi-)external multilevel graph partitioning algorithm.

5.1 Coarsening/Contraction. We have two different algorithms to create graph hierarchies depending on the memory model that we use. In general to create a graph hierarchy, we compute a clustering with size-constraints of the current graph using some algorithm from Section 4. The next step is to renumber the cluster IDs. The external algorithm sorts the nodes by their cluster ID and scans the sorted array assigning new cluster IDs from $0, \ldots, n'-1$, where n' is the number of the distinct clusters. This step can be done using Sort($|V|$) I/O operations. In contrast, the semi-external algorithm uses an additional array of size $\mathcal{O}(|V|)$ to assign new cluster IDs. Hence, it needs $\mathcal{O}(1)$ I/O operations.

The external algorithm builds an array of triples (cluster[u], cluster[v], $w(u,v)$) for each edge $(u,v) \in E$ to build the contracted graph. This array is sorted lexicographically by the first two entries using Sort($|E|$) I/Os. Then we merge parallel edges and build the edges of the quotient graph by iterating through the sorted array using Scan($|E|$) I/Os. The total I/O volume of this step is Sort($|E|$). The semi-external algorithm stores pairs (cluster[u], cluster[v]) for each edge (u,v) in a hash table and uses it to build the contracted graph. This can be done using Scan($|E|$) I/Os. If the number of edges of the contracted graphs decreases geometrically and a constant number of label propagation iterations is assumed, the complete hierarchy can be built using Scan($|E|$) I/Os using the semi-external algorithm or Sort($|E|$) I/Os using the external algorithm.

5.2 Uncoarsening/Solution Transfer. In this step, we want to transfer a solution of a coarse level to the next finer level in the hierarchy and perform some local search. Let $\mathcal{Q} = (V_\mathcal{Q}, E_\mathcal{Q})$ be a contracted graph of the next finer level $G = (V, E)$ and let cluster$_\mathcal{Q}$ be a partition of the contracted graph. Recall that the contracted graph has been built according to a clustering of the graph G. Also note that cluster i of G corresponds to node i in the contracted graph. Hence, for a node $v \in V$ the transferred cluster ID of the coarse level is cluster$'_G[v]$:= cluster$_\mathcal{Q}$[cluster$_G[v]$].

To transfer the solution in external algorithm, we build an array of pairs (cluster$_G[v]$, v) and sort it by the first component using Sort($|V|$) I/O operations. Now we iterate through both of the arrays cluster$_\mathcal{Q}$ and $\{(\text{cluster}_G[v], v)\}$ at the same time and generate

the array $\{(\text{cluster}_{\mathcal{Q}}[\text{cluster}_G[v]], v)\}$ which contains the transferred solution. We sort the resulting array by the second component and apply the clustering to our graph. Overall, we need $\text{Sort}(|V|)$ I/O operations. The semi-external algorithm iterates through all nodes of graph G and updates the cluster IDs of each node. This can be done using $\mathcal{O}(1)$ I/Os. If the number of nodes of the quotient graphs decreases geometrically then uncoarsening of the complete hierarchy can be done using $\text{Sort}(|V|)$ I/O operations. After each solution transfer step, we apply a size-constrained LPA (using $L_{\max}$) to improve the solution in (semi-)external memory on the current level.

6 Experiments

In this section, we evaluate the performance of our graph clustering and multilevel graph partitioning algorithms. We compare ourselves against kMetis, which is probably the most widely used partitioning algorithm and the results of KaHIP presented in [16].

Methodology. We implemented our algorithms using C++. Our implementation uses the STXXL library [22] to a large extend, i.e. external arrays, sorting algorithms and priority queues. All binaries were built using g++ 4.8.2. We run our algorithm once in order to save running time and report cut size, running time, internal memory consumption and I/O volume. Experiments were run on two machines. Machine A is used in Section 6.1. It has two Intel Xeon X5550 running at 2.66 GHz (4-Core) with 48 GB RAM and 8xSATA 1 TB (read 120 MB/s, write 120 MB/s). Machine B is used in Section 6.2. It has two Intel Xeon E5-2650v2 running at 2.6 GHz (8-Core) with 128Gb RAM and 4xSSD 1 TB (read 1440 MB/s, write 1440 MB/s). The size of a block during the experiments is set to 1 MB. Using a larger block size does not yield an advantage because of the parallel prefetching read/write algorithms that are implemented within the STXXL.

Instances. We evaluate our algorithms on graphs collected from [23–25]. Table 1 summarizes the main properties of the benchmark set. Our benchmark set includes a number of social networks and web graphs. rggX is a *random geometric graph* from [25] with 2^X nodes where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost certainly connected. clueweb12* is the largest connected component of the web graph clueweb12 from [24].

Table 1: Benchmark set

Graph	n	m	Ref.
nlpkkt200	$\approx 16.2M$	$\approx 215M$	[23]
uk-2002	$\approx 18.5M$	$\approx 262M$	[24]
arabic-2005	$\approx 22.7M$	$\approx 553M$	[24]
nlpkkt240	$\approx 27.9M$	$\approx 373M$	[23]
it-2004	$\approx 41.2M$	$\approx 1G$	[23]
twitter	$\approx 41.6M$	$\approx 1.2G$	[24]
sk-2005	$\approx 50.6M$	$\approx 1.8G$	[24]
webbase-2001	$\approx 118M$	$\approx 854M$	[23]
uk-2007	$\approx 106M$	$\approx 3.3G$	[24]
clueweb12*	$\approx 951M$	$\approx 37.3G$	[24]
rgg31	$\approx 2.1G$	$\approx 21.9G$	[25]

6.1 Graph Clustering Algorithms. We now compare different graph clustering algorithms not using a size-constraint. We do this on the nine largest graphs from our collection (excluding rgg31 and clueweb12*). By default, we perform three label propagation iterations. We use the following algorithm abbreviations: LP - label propagation, SE - semi-external, E - external and BT for the coloring-based graph clustering algorithm. We use the variant of the coloring-based clustering algorithm where each bucket fits into the internal memory (this turned out to be true for all instances). The external label propagation algorithm was run with 1 GB of internal memory. Figure 3 summarizes the results. First of all, the semi-external and the external label propagation outperform the coloring-based clustering algorithm. This can be explained by the fact that each tuple in the bucket uses 20 bytes (4 bytes per element of the tuple). Hence, the sorting and scanning operations consume a significantly larger amount of time. However, the coloring-based clustering algorithm would be able to compute a graph clustering fulfilling a size-constraint in the external memory model. We also run LP with the active nodes strategy (see TR [26] for details on the active nodes algorithm). It turns out that the LP with active nodes is faster in both models as soon as enough iterations are performed.

6.2 Multilevel Graph Partitioning. We now present the results of the semi-external multilevel graph partitioning algorithm. We mainly use the semi-external label propagation algorithm with size-constraints and its parallel version to compute graph clusterings during coarsening and to perform refinement during uncoarsening. Our experiments focus on the four web graphs from our benchmark set that are used in [16] as well as rgg31 and clueweb12*. During

coarsening we use $L_{\max}$ as a size-constraint on the clusters. Note that this is an much weaker restriction on the cluster sizes than the one used in [16]. Using this weaker constraint speeds up the algorithm significantly. The random geometric graph and clueweb12* have been partitioned into two blocks and the other graphs into sixteen blocks using $\epsilon = 0.03$ so that we can compare ourselves with the results presented by Meyerhenke et al. [16]. The values for KaHIP and kMetis have been taken from their paper and the experiments have been performed on different machine with 1TB main memory. Table 2 summarizes the results. The parallel semi-external LPA uses 16 threads (hyperthreading) and the mem column displays how much memory has been used by our algorithm. Our algorithm computes partitions that are almost as good as those computed by KaHIP. In the worst case, our algorithm cuts about 9% more edges than KaHIP. On average we cut about 1% more edges. On the other hand, our algorithm computes much better cuts than kMetis (except on the graph sk-2005). On average our algorithm cuts 73% less edges then kMetis. Note that the running time of our sequential semi-external algorithm is always smaller than the running time of KaHIP. This is partially due to the fact that we use weaker size-constraint during coarsening which makes the contracted even smaller compared to the contracted graphs computed in KaHIP and the fact that we use less label propagation rounds. After the first contraction step we switched to the internal memory implementation of KaHIP to partition the coarser graph. Figure 4 presents the scalability of our shared memory parallel semi-external algorithm on uk-2007. We can see that the algorithm scales well but comes with parallelization overheads. However, using 16 threads the parallelization always speeds up the computations compared to the sequential algorithm. On the other hand the parallelization gets close to the lower bound which is given by the I/O that needs to be performed.
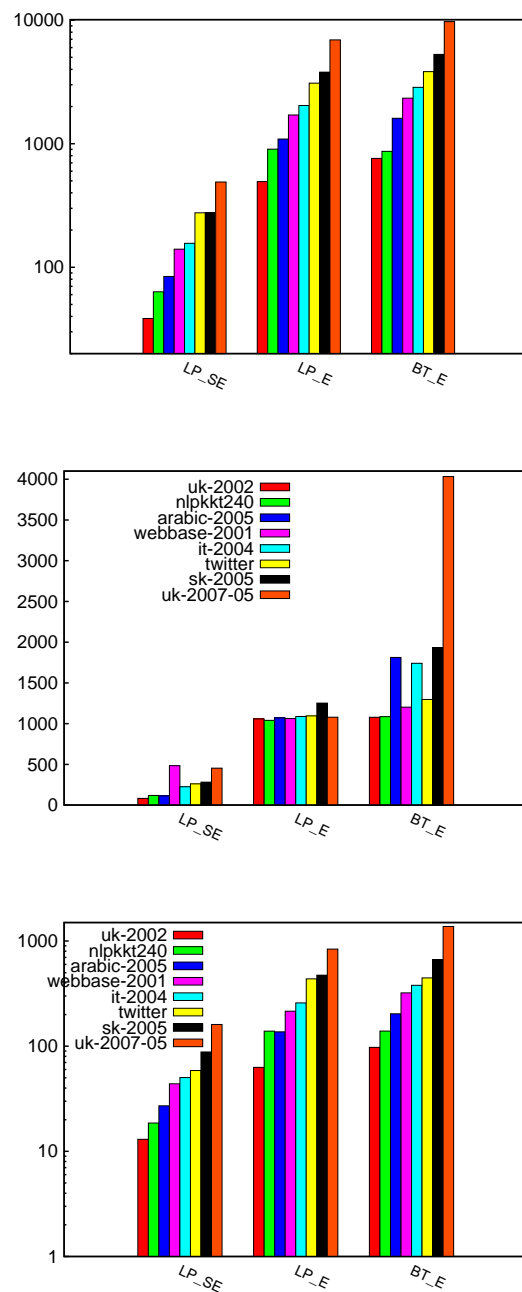

Figure 3: Top: Running time in seconds. Middle: Memory consumption in Mb. Bottom: I/O volume of different graph clustering algorithm in GB.
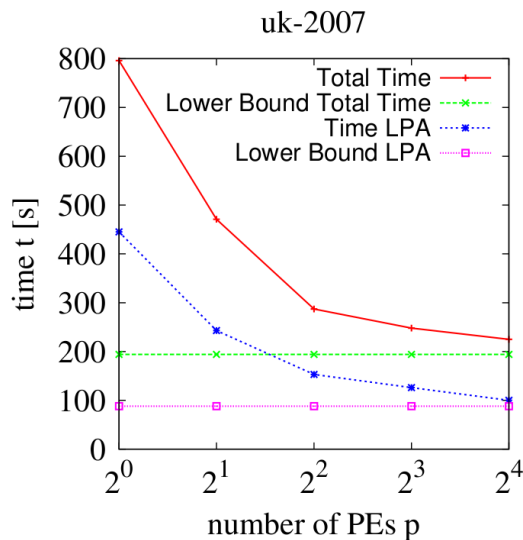
uk-2007

Figure 4: Scalability of the parallel semi-external graph partitioning algorithm on uk-2007.

Table 2: Results

algorithm	time[s]	cut	mem
arabic-2005			
LP_SE	81.3	2.04M	0.48GB
P_LP_SE	36.0	2.28M	1.85GB
KaHIP	111.2	1.87M	
kMetis	99.6	3.50M	
uk-2002			
LP_SE	57.7	1.46M	0.56GB
P_LP_SE	39.7	1.54M	1.53GB
KaHIP	71.7	1.43M	
kMetis	63.7	2.41M	
sk-2005			
LP_SE	257.7	19.00M	1.59 Gb
P_LP_SE	203.4	22.25M	7.76 Gb
KaHIP	387.1	20.34M	
kMetis	405.3	18.56M	
uk-2007			
LP_SE	435.1	4.18M	4.26GB
P_LP_SE	209.1	4.55M	8.58GB
KaHIP	626.6	4.10M	
kMetis	827.6	10.86M	
rgg31			
LP_SE	3803.3	341K	34.40GB
P_LP_SE	3312.3	340K	96.40GB
clueweb12*			
LP_SE	5441.0	179M	21.27GB
P_LP_SE	2649.8	252M	77.12GB

7 Conclusion and Future Work

We presented algorithms that are able to partition and cluster huge complex networks with billions of edges on cheap commodity machines. This has been achieved by using a (semi-)external variant of the size-constrained LPA that can be used for coarsening and as a simple local search algorithm. A shared memory parallelization of the algorithm further reduces the running time of the algorithm. Moreover, we presented the *first* fully external graph clustering/partitioning algorithm that is able to deal with a size-constraint. Our experiments indicate that our semi-external algorithms are able to compute high quality partitions in time is compareble to an efficient internal memory implementation. As a part of future work, it might be interesting to define a (semi-)external clustering algorithm that optimizes modularity. This can be done by using the techniques of this paper but using different update rules to compute the cluster ID of a node.

References

[1] Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: Large-scale Graph Computation on Just a PC. In: Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI). Volume 8. (2012) 31–46

[2] Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some Simplified NP-complete Problems. (1974) 47–63

[3] Bui, T.N., Jones, C.: Finding Good Approximate Vertex and Edge Partitions is NP-Hard. Information Processing Letters **42**(3) (1992) 153–159

[4] Abello, J., Buchsbaum, A.L., Westbrook, J.: A Functional Approach to External Graph Algorithms. Algorithmica **32**(3) (2002) 437–458

[5] Bichot, C., Siarry, P., eds.: Graph Partitioning. Wiley (2011)

[6] Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent Advances in Graph Partitioning. Technical Report ArXiv:1311.3144 (2013)

[7] Diekmann, R., Preis, R., Schlimbach, F., Walshaw, C.: Shape-optimized Mesh Partitioning and Load Balancing for Parallel Adaptive FEM. Parallel Computing **26**(12) (2000) 1555–1581

[8] Abou-Rjeili, A., Karypis, G.: Multilevel Algorithms for Partitioning Power-Law Graphs. In: Proc. of 20th Int. Parallel and Distributed Processing Symp. (2006)

[9] Meyerhenke, H., Monien, B., Schamberger, S.: Accelerating Shape Optimizing Load Balancing for Parallel FEM Simulations by Algebraic Multigrid. In: Proc. of 20th Int. Parallel and Distributed Processing Symp. (2006)

[10] Hendrickson, B.: Chaco: Software for Partitioning Graphs. `http://www.cs.sandia.gov/~bahendr/chaco.html`

[11] Walshaw, C., Cross, M.: Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. SIAM J. on Scientific Computing **22**(1) (2000) 63–80

[12] Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. on Scientific Computing **20**(1) (1998) 359–392

[13] Monien, B., Schamberger, S.: Graph Partitioning with the Party Library: Helpful-Sets in Practice (2004)

[14] Pellegrini, F.: Scotch Home Page. `http://www.labri.fr/pelegrin/scotch`

[15] Raghavan, U.N., Albert, R., Kumara, S.: Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks. Physical Review E **76**(3) (2007)

[16] Meyerhenke, H., Sanders, P., Schulz, C.: Partitioning Complex Networks via Size-constrained Clustering. In: Proc. of the 13th Int. Symp. on Experimental Algorithms (SEA'14). LNCS, Springer (2014)

[17] Stanton, I., Kliot, G.: Streaming Graph Partitioning for Large Distributed Graphs. In: Proc. of the 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. KDD '12, ACM (2012) 1222–1230

[18] Sanders, P., Schulz, C.: Think Locally, Act Globally: Highly Balanced Graph Partitioning. In: Proc. of the 12th Int. Symp. on Experimental Algorithms (SEA'13). LNCS, Springer (2013)

[19] Zeh, N.: I/O-efficient Graph Algorithms. EEF Summer School on Massive Data Sets (2002)

[20] Sanders, P.: Fast priority queues for cached memory. ACM Journal of Experimental Algorithmics **5** (1999) 312–327

[21] Arge, L.: The Buffer Tree: A Technique for Designing Batched External Data Structures. Algorithmica **37**(1) (2003) 1–24

[22] Beckmann, A., Bingmann, T., Dementiev, R., Sanders, P., Singler, J.: STXXL Home Page. `http://stxxl.sourceforge.net/`

[23] University of Florida. Sparse Matrix Collection.

[24] Laboratory of Web Algorithms, University of Milano, Datasets

[25] Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a Scalable High Quality Graph Partitioner. Proc. of the 24th Int. Parallel and Distributed Processing Symp. (2010) 1–12

[26] Akhremtsev, Y., Sanders, P., Schulz, C.: (Semi-)External Algorithms for Graph Partitioning and Clustering). Technical report, Karlsruhe Institute of Technology (2014)