Buffered Streaming Graph Partitioning

MARCELO FONSECA FARAJ and CHRISTIAN SCHULZ, Heidelberg University

Partitioning graphs into blocks of roughly equal size is a widely used tool when processing large graphs. Currently, there is a gap observed in the space of available partitioning algorithms. On the one hand, there are streaming algorithms that have been adopted to partition massive graph data on small machines. In the streaming model, vertices arrive one at a time including their neighborhood, and then have to be assigned directly to a block. These algorithms can partition huge graphs quickly with little memory, but they produce partitions with low solution quality. On the other hand, there are offline (shared-memory) multilevel algorithms that produce partitions with high-quality but also need a machine with enough memory to partition huge networks. In this work, we make a first step to close this gap by presenting an algorithm that computes significantly improved partitions of huge graphs using a single machine with little memory in a streaming setting. First, we adopt the buffered streaming model which is a more reasonable approach in practice. In this model, a processing element can store a buffer of nodes alongside with their edges before making assignment decisions. When our algorithm receives a batch of nodes, we build a model graph that represents the nodes of the batch and the already present partition structure. This model enables us to apply multilevel algorithms and in turn, on cheap machines, compute much higher quality solutions of huge graphs than previously possible. To partition the model graph, we develop a multilevel algorithm that optimizes an objective function that has previously been shown to be effective for the streaming setting. Surprisingly, this also removes the dependency on the number of blocks k from the running time compared to the previous state-of-the-art. Overall, our algorithm computes, on average, 75.9% better solutions than Fennel [35] using a very small buffer size. In addition, for large values of k our algorithm becomes faster than Fennel.

CCS Concepts: • **Theory of computation → Streaming, sublinear and near linear time algorithms**; *Graph algorithms analysis;*

Additional Key Words and Phrases: Graph partitioning, streaming algorithms, multilevel algorithms

ACM Reference format:
Marcelo Fonseca Faraj and Christian Schulz. 2022. Buffered Streaming Graph Partitioning. *J. Exp. Algorithmics* 27, 1, Article 1.10 (October 2022), 26 pages.
https://doi.org/10.1145/3546911

1 INTRODUCTION

Complex graphs are increasingly being used to model phenomena such as social networks, data dependency in applications, citations of articles, and biological systems like the human brain. Often these graphs are composed of billions of entities that give rise to specific properties and structures. As a concrete example to cope with such graphs, graph databases [12] and graph processing

Partially supported by DFG grant SCHU 2567/1-2.
Authors' address: M. F. Faraj and C. Schulz, Heidelberg University, Im Neuenheimer Feld 205, 69120 Heidelberg, Germany; emails: {marcelofaraj, christian.schulz}@informatik.uni-heidelberg.de.

1084-6654/2022/10-ART1.10 $15.00
https://doi.org/10.1145/3546911

frameworks [10, 15, 21] can be used to store a graph, query it, and provide other operations. If the graphs become too large, then they have to be distributed over many machines in order for the system to provide scalable operations. A key operation for scalable computations on huge graphs is to partition its components among k **processing elements** (**PEs**) such that each PE receives roughly the same amount of components and the communication between PEs in the underlying application is minimized. In the distributed setup, each PE operates on some portion of the graph and communicates with other PEs through message-passing. This operation is naturally modeled by the graph partitioning problem, which computes a partition of the graph into k blocks such that the blocks have roughly the same size and the number of edges crossing blocks, i.e., communication, is minimized. Graph partitioning is NP-hard [14] even for unweighted trees of maximum degree 3 [1] and no constant-factor approximation algorithms exist [2]. Thus, heuristic algorithms are used in practice.

There has been an extensive body of work in the area of graph partitioning. Roughly speaking, there are streaming algorithms, internal memory (shared-memory parallel) algorithms, and distributed memory parallel algorithms. However, currently, there is a gap observed in the design space of available partitioning algorithms. First of all, the most popular streaming approach in the literature is the one-pass model, in which vertices arrive one at a time including their neighborhood and then have to be permanently assigned to blocks. Algorithms based on this model can partition huge graphs quickly with little memory, but compute rather low-quality partitions. To improve partition quality, the graph can be restreamed while the one-pass algorithm updates block assignment, but this is still far behind offline approaches. Offline multilevel algorithms such as KaHIP [32] or Metis [19] are widely known and can produce partitions with high-quality compared to streaming algorithms. Nevertheless, they cannot partition huge graphs unless a machine with sufficient memory is used and hence can not be used for example for preprocessing in out-of-core algorithms. Lastly, distributed algorithms are able to partition huge graphs successfully and compute high-quality solutions. However, they require a large amount of computational resources and typically access to a supercomputer, which can be infeasible depending on the application. Moreover, a distributed partitioning algorithm needs to split the input graph among different machines before actually partitioning it. When the graph is too large, this preliminary partition has to be generated on the fly by a stream partitioning algorithm while loading the graph. Other natural applications for stream partitioning include distributed graph processing systems based on a load-compute-store logic such as MapReduce [11] and Giraph [10], and systems, which support native graph-as-a-stream computations such as Kineograph [9], and Apache Flink [8].

Contribution. In this work, we start to fill the gap currently observed for the existing graph partitioning algorithms. We propose an algorithm that can compute significantly better partitions of huge graphs than the currently available streaming algorithms while using a single machine without a lot of memory. We adopt the buffered streaming model which allows a buffer of nodes to be received and stored before making assignment decisions. Our algorithm is carefully engineered to produce partitions of improved quality by using a sophisticated multilevel scheme on a compressed model of the buffer *and* the already assigned vertices. Our multilevel algorithm optimizes for the same objective as the previous state-of-the-art Fennel [35]. However, due to the multilevel scheme used on the compressed model, our local search algorithms have a global view on the optimization problem and hence compute better solutions overall. Lastly, using the multilevel scheme reduces the time complexity from $O(nk + m)$ of Fennel to $O(n + m)$, where k is the number of blocks a graph has to be partitioned in. To this end, experiments indicate that our algorithm can partition huge networks on machines with small memory while computing *better* solutions than the previous state-of-the-art in the streaming setting. At the same time our algorithm is *faster* than the previous state-of-the-art for larger values of blocks k.

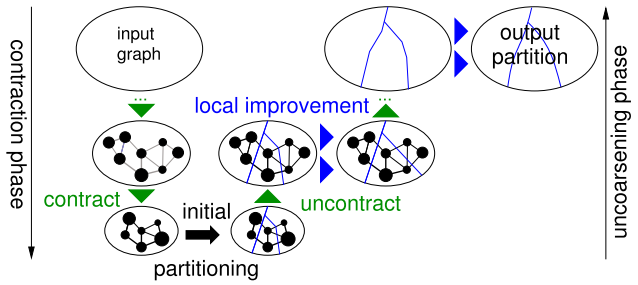


Fig. 1. Multilevel graph partitioning. The graph is recursively contracted to achieve smaller graphs. After the coarsest graph is initially partitioned, a local search method is used on each level to improve the partitioning induced by the coarser level. Thicker edges have a higher weight.

2 PRELIMINARIES

2.1 Basic Concepts

Let $G = (V = \{0, \ldots, n-1\}, E)$ be an *undirected graph* with no multiple or self edges allowed, such that $n = |V|$ and $m = |E|$. Let $c : V \to \mathbb{R}_{\geq 0}$ be a node-weight function, and let $\omega : E \to \mathbb{R}_{>0}$ be an edge-weight function. We generalize c and ω functions to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. Let $N(v) = \{u : \{v, u\} \in E\}$ denote the neighbors of v. A graph $S = (V', E')$ is said to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. When $E' = E \cap (V' \times V')$, S is an *induced* subgraph. Let $d(v)$ be the degree of node v, Δ be the maximum degree of G, and $\Delta_{V'}$ be the maximum degree of the subgraph induced by $V' \subseteq V$. The ***graph partitioning* problem (GPP)** consists of assigning each node of G to exactly one of k distinct *blocks* respecting a balancing constraint in order to minimize the edge-cut. More precisely, GPP partitions V into k blocks $V_1, \ldots, V_k$ (i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$), which is called a *k-partition* of G. The *balancing constraint* demands that the sum of node weights in each block does not exceed a threshold associated with some allowed *imbalance* ϵ. More specifically, $\forall i \in \{1, \ldots, k\} : c(V_i) \leq L_{\max} := \lceil (1 + \epsilon) \frac{c(V)}{k} \rceil$.

The *edge-cut* of a k-partition consists of the total weight of the edges crossing blocks, i.e., $\sum_{i<j} \omega(E_{ij})$, where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. Let a *clustering* be any partition of V, i.e., a set of *blocks* or *clusters* $V_1, \ldots, V_t \subset V$ such that $V_1 \cup \cdots \cup V_t = V$ and $V_i \cap V_j = \emptyset$, where $t \in [1, n]$. An abstract view of the partitioned graph is a *quotient graph* $\mathcal{Q}$, in which nodes represent blocks and edges are induced by the connectivity between blocks. More precisely, each node i of $\mathcal{Q}$ has weight $c(V_i)$ and there is an edge between i and j if there is at least one edge in the original partitioned graph that runs between the blocks V_i and V_j. We call *neighboring blocks* a pair of blocks that is connected by an edge in the quotient graph. A vertex $v \in V_i$ that has a neighbor $w \in V_j$, $i \neq j$, is a boundary vertex. A successful heuristic for partitioning large graphs is the ***multilevel graph partitioning* (MGP)** approach depicted in Figure 1, where the graph is recursively *contracted* to achieve smaller graphs which should reflect the same basic structure as the input graph. After applying an *initial partitioning* algorithm to the smallest graph, the contraction is undone and, at each level, a *local search* method is used to improve the partitioning induced by the coarser level. *Contracting* a cluster of nodes $C = \{u_1, \ldots, u_\ell\}$ consists of replacing these nodes by a new node v. The weight of this node is set to the sum of the weight of the cluster vertices. Moreover, the new node is connected to all elements $w \in \bigcup_{i=1}^{\ell} N(u_i)$, $\omega(\{v, w\}) = \sum_{i=1}^{\ell} \omega(\{u_i, w\})$. This ensures that a partition from a coarser level that is transferred to a finer level maintains the edge cut and the balance of the partition. Figure 2 gives an example. The *uncontraction* of a node undoes the

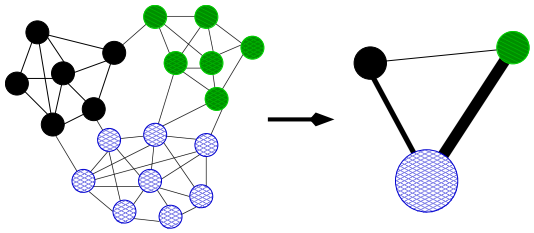


Fig. 2. Contraction of a clustering. Each cluster is represented by a different color on the lef-hand side graph. Each cluster on the left-hand side is contracted to a single vertex on the right-hand side.

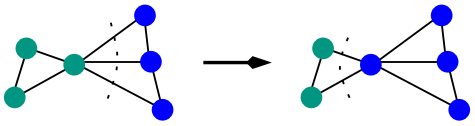


Fig. 3. Typical step of a local search algorithm. Block assignments are indicated by colors. In this example, a single vertex is moved to another block in order to decrease the overall edge-cut.

contraction. Local search moves vertices between the blocks in order to reduce the objective, which is exemplified in Figure 3.

Computational Models. The focus of this article is to engineer a graph partitioning algorithm for a streaming input. In particular, the input is a stream of nodes alongside with their respective adjacency lists. The classic streaming model is the one-pass model, in which the nodes have to be permanently assigned to a block as soon as they are loaded from the input. As soon as assignment decisions of an algorithm for the current node depend on the previous decisions, an algorithm in the model has to store the assignment of the previously loaded nodes and hence needs $\Omega(n)$ space. We use an extended version of this model, which is called the *buffered streaming* model. More precisely, a δ-sized *buffer* or *batch* of input nodes with their neighborhood is repeatedly loaded. Partition/block assignment decisions have to be made after the whole buffer is loaded. While we investigate the dependence of our algorithm on this parameter, in practice the parameter will depend on the amount of available memory on a machine. The parameter can be dynamically chosen such that the buffer is "full" if $\Theta(n)$ space has been loaded from the disk. In particular, if there are nodes that have degree $\Omega(n)$ then algorithms will still work as it only loads the node and its neighbors, but not their edges. Hence the buffered streaming model asymptotically does not need more space than a one-pass streaming algorithm if this setting is used. This holds true even in the worst case: when a node has a degree close to n. In this article, we use a constant δ throughout the run of an algorithm. For a predefined batch size δ, the total amount of $\lceil n/\delta \rceil$ batches are consecutively loaded and assigned to blocks.

2.2 Related Work

There has been a large amount of research on graph partitioning. We refer the reader to [5, 7, 33] for extensive material and references. Here, we focus on results close to our main contribution. The most successful general-purpose offline algorithms to solve the graph partitioning problem for huge real-world graphs are based on the multilevel approach. The most commonly used formulation of the multilevel scheme for graph partitioning was proposed in [16]. However, these algorithms require the graph to fit into the main memory of a single machine or into the memory of a distributed machine if a distributed memory parallel partitioning algorithm is used.

Stanton and Kliot [34] introduced graph partitioning in the streaming model and proposed many natural heuristics to solve it. These heuristics include one-pass methods such as Hashing, Chunking, and **linear deterministic greedy** (**LDG**), and some buffered methods such as greedy evocut. The evocut buffered model is different from our model as it is an extended one-pass model in which a buffer of fixed size is kept and the algorithm can assign any node from the buffer to a block, rather than the one that has been received most recently. However, all methods that use a buffer perform significantly worse than random partitionings—hence we do not include them in our experiments. In their experiments, LDG had the best overall results in terms of total edge-cut. In this algorithm, node assignments prioritize blocks containing more neighbors while using a penalty multiplier to control imbalance. In particular, it assigns a node v to the block V_i that maximizes $|V_i \cap N(v)| * \lambda(i)$ with $\lambda(i)$ being a multiplicative penalty defined as $(1 - \frac{|V_i|}{L_{\max}})$. The intuition here is that the penalty avoids to overload blocks that are already very heavy. Tsourakakis et al. [35] proposed a one-pass partitioning heuristic named Fennel, which is an adaptation of the widely-known clustering objective *modularity* [6]. Roughly speaking, Fennel assigns a node v to a block V_i, respecting a balancing threshold, in order to maximize an expression of type

$$|V_i \cap N(v)| - f(|V_i|), \tag{1}$$

i.e., with an additive penalty. The assignment decision of Fennel is based on an interpolation of two properties: attraction to blocks with more neighbors and repulsion from blocks with more non-neighbors. When $f(|V_i|)$ is a constant, the resulting objective function coincides with the first property. If $f(|V_i|) = |V_i|$, the objective function coincides with the second property. More specifically, the authors defined the Fennel objective function by using $f(|V_i|) = \alpha \times \gamma \times |V_i|^{\gamma-1}$, in which γ is a free parameter and $\alpha = m\frac{k^{\gamma-1}}{n^{\gamma}}$. After a parameter tuning made by the authors, Fennel uses $\gamma = \frac{3}{2}$, which provides $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$. Note that in the original article, the authors assume k to be constant and hence derive a complexity of $O(n + m)$. However, since one has to iterate overall blocks k for each node the complexity of the algorithm depends on k and is given by $O(nk + m)$.

A restreaming approach has been introduced by Nishimura and Ugander [24]. In this model, multiple passes through the entire input are allowed, to allow iterative improvements. The authors proposed easily implementable restreaming versions of LDG and Fennel: ReLDG and ReFennel. Awadelkarim and Ugander [3] studied the effect of node ordering for streaming graph partitioning. The authors introduced the notion of *prioritized streaming*, in which (re)streamed nodes are statically or dynamically reordered based on some priority. Patwary et al. [26] proposed WStream, a greedy stream graph partitioning algorithm that keeps a sliding stream window. This sliding window contains a few hundred nodes such that it gives more information about the next node to be allocated to a block. However, the code is not available and the algorithm has only been evaluated on graphs with a few thousand vertices. Jafari et al. [18] proposed a shared-memory partitioning algorithm based on a buffered streaming computational model similar to the one we use here. Their algorithm uses the idea of the multilevel algorithm but with a simplified structure in which the LDG one-pass algorithm constitutes the coarsening step, the initial partitioning, and the local search during uncoarsening. Our work differs from theirs in the fact that we focus on single-threaded execution, our running time complexity is $O(n + m)$ while theirs is $O(nk + m)$, we construct a sophisticated model instead of processing the nodes of a batch directly, and our algorithm is inspired by the Fennel one-pass algorithm, which outperformed LDG in previously published studies. There are also a wide range of algorithms that focus on (buffered) streaming edge partitioning [22, 28, 31]. However, as our focus is on edge cut-based algorithms they are beyond the scope of this work.

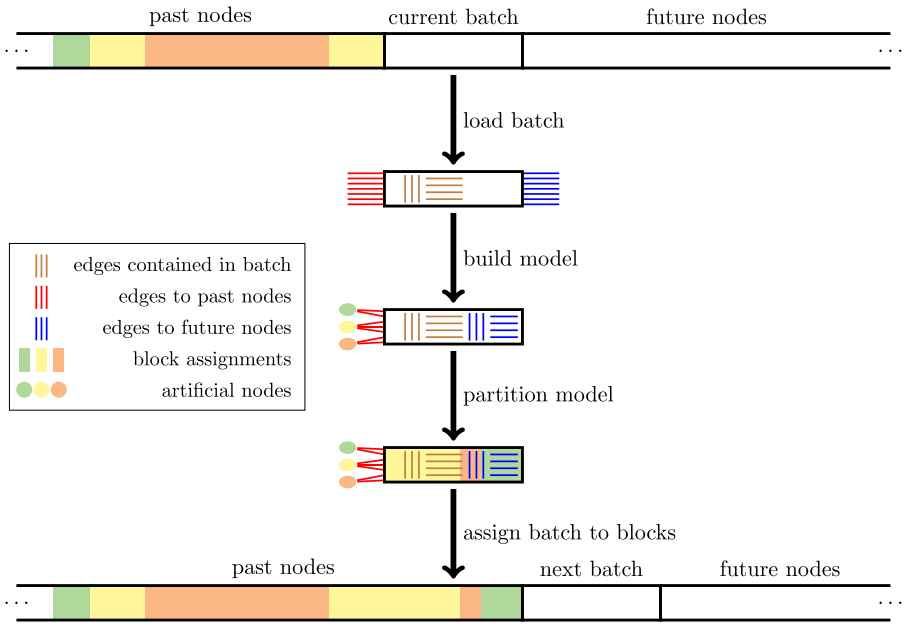


Fig. 4. Detailed structure of HeiStream. The algorithm starts by loading a batch of nodes alongside with their edges. Next, it builds a meaningful model based on the loaded nodes and edges. In this model, edges to past nodes are connected to artificial representing blocks. On the other hand, edges to future nodes are inserted in the model by contracting each future node to one of its neighbors in the current batch. This model is then partitioned using a multilevel algorithm. Based on the partition of the model, the nodes of the current batch are permanently assigned to blocks. This whole process is repeated for the next batch until the whole graph is partitioned.

3 BUFFERED GRAPH PARTITIONING

We now present our main contribution, namely HeiStream: a novel algorithm to solve graph partitioning in the buffered streaming model. We start by first outlining the overall structure behind HeiStream and then we present each of the algorithm components.

3.1 Overall Structure

We now explain the overall structure of HeiStream. We slide through the streamed graph G by repeating the following successive operations until all the nodes of G are assigned to blocks. First, we load a batch containing δ nodes alongside with their adjacency lists. Second, we build a model $\mathcal{B}$ to be partitioned. This model represents the already partitioned vertices as well as the nodes of the current batch. Third, we partition $\mathcal{B}$ with a multilevel partitioning algorithm to optimize for the Fennel objective function. And finally, we permanently assign the nodes from the current batch to blocks. Algorithm 1 summarizes the general structure of HeiStream and Figure 4 shows the detailed structure of HeiStream.

3.2 Model Construction

We build two different models, which yield a running time–quality tradeoff. We start by describing the *basic model* and then extend this later. When a batch is loaded, we conceptually build the model $\mathcal{B}$ as follows. We initialize $\mathcal{B}$ as the subgraph of G induced by the nodes of the current batch. We

ALGORITHM 1: Structure of HeiStream

```
while G has not been completely streamed do
    Load batch of vertices
    Build model B
    Run multilevel partitioning on model B
    Assign nodes of batch to permanent blocks
```

ALGORITHM 2: Basic Model Construction

Input: Buffer B with δ vertices; Edges of the vertices in B
Output: Graph model $\mathcal{B}$

```
B ← Graph with δ unity-weighted nodes, k properly-weighted artificial nodes, and no edges
for u ∈ B do
    u' ← getBatchNodeID(u)                    // map node ID from graph domain to batch domain
    for v ∈ N(u) do
        if v ∈ B then
            v' ← getBatchNodeID(v)            // map node ID from graph domain to batch domain
            B.InsertEdge(u', v')              // directly insert edge (u', v') into B
        else if v is past node then
            a ← getArtificialNodeID(v)        // map node ID based on the assignment of v
            B.InsertOrIncrementEdge(u', a)    // insert or increment weight of edge (u', a)
```

add k artificial nodes to the model. These represent the k preliminary blocks in their current state, i.e., filled with the nodes from the previous batches, which were already assigned. The weight of an artificial node i is set to the weight of block V_i. A node of the current batch is connected to an artificial node i if it has a neighbor from the previous batch that has been assigned to block V_i. If this creates parallel edges, we replace them by a single edge with its weight set to the sum of the weight of the parallel edges. Note that the basic model does ignore edges towards nodes that will be streamed in future batches, i.e., batches that have not been streamed yet. Algorithm 2 shows a practical procedure to build our basic model. Our *extended model* incorporates edges towards nodes from future, not yet loaded, batches—if the stream contains such edges. We call edges towards nodes of future batches ghost edges and the corresponding endpoint in the future batch ghost node. Ghost nodes and ghost edges provide partial information about parts of G that have not yet been streamed. Hence, representing them in model $\mathcal{B}$ enhances the partitioning process. Note though that simply inserting all ghost nodes and edges can overload memory in case there is an excessive amount of them. Thus our approach consists of randomly contracting the ghost nodes with one of their neighboring nodes from the current batch. Note that this contraction increments the weight of a node within our model and ensures that if there is more than one node from the current batch connected to the same future node, then there will be edges between those nodes in our model. Also, note that the contraction ensures that the number of nodes in all models throughout the batched streaming process is constant. This prevents memory from being overloaded and makes it unnecessary to reallocate memory for $\mathcal{B}$ between successive batches. In order to give a lower priority to ghost edges in comparison to the other edges, we divide the weight of each ghost edge by 2 in our model. Our extended model is conceptually illustrated in Figure 4 and Algorithm 3 shows a practical procedure to build it.

3.3 Multilevel Weighted Fennel

Our approach to partition the model $\mathcal{B}$ is a multilevel version of the algorithm Fennel. Recall that the multilevel scheme consists of three successive phases: coarsening, initial partitioning, and

ALGORITHM 3: Extended Model Construction

Input: Buffer B with δ vertices; Edges of the vertices in B
Output: Graph model $\mathcal{B}$

$\mathcal{B} \leftarrow$ Graph with δ unity-weighted nodes, k properly-weighted artificial nodes, and no edges
$S \leftarrow \emptyset$ // set of ghost node IDs
$M \leftarrow \emptyset$ // map: ghost node ID → list of node IDs in the batch
for $u \in B$ **do**
  $u' \leftarrow getNonArtificialNodeID(u)$ // map node ID from graph domain to batch domain
  for $v \in N(u)$ **do**
    if $v \in B$ **then**
      $v' \leftarrow getNonArtificialNodeID(v)$ // map node ID from graph domain to batch domain
      $\mathcal{B}.InsertEdge(u', v')$ // directly insert edge (u', v') into $\mathcal{B}$
    else if v is past node **then**
      $a \leftarrow getArtificialNodeID(v)$ // map node ID based on the assignment of v
      $\mathcal{B}.InsertOrIncrementEdge(u', a)$ // insert or increment weight of edge (u', a)
    else if v is future node **then**
      $S \leftarrow S \cup \{v\}$ // record ghost node ID
      $M[v] \leftarrow M[v] \cup \{u'\}$ // add u' to the list of neighbors of ghost node v
for $w \in S$ **do**
  $u' \leftarrow PickRandom(M[w])$ // randomly pick neighbor u' of ghost node w
  $\mathcal{B}.IncrementNodeWeight(u')$ // increment weight of u' by 1 (due to contraction)
  for $v' \in M[w] \setminus \{u'\}$ **do** // go through remaining neighborhood of w
    $\mathcal{B}.InsertOrIncrementEdge(u', v')$ // insert or increment weight of edge (u', v')

uncoarsening, as depicted in Figure 1. Note, however, that the artificial nodes in our model can become very heavy and are not allowed to be contracted or change their block. As a consequence, these nodes need special handling in our multilevel algorithm. Moreover, the Fennel algorithm is designed for unweighted graphs, hence it needs some adaptation to be used in HeiStream. We introduce a generalization of Fennel for weighted graphs that can be directly employed in a multilevel algorithm. In this section, we present this generalized Fennel objective and explain the details of our multilevel algorithm to partition the model.

3.3.1 Generalized Fennel. As already mentioned, adaptations are necessary to implement Fennel for weighted graphs, in particular in a multilevel scheme. However, our model $\mathcal{B}$ has nodes and edges that are weighted—due to connections to the artificial nodes as well as future nodes that may be contracted into the model. Moreover, the multilevel scheme creates a sequence of weighted graphs. Note that a generalization of the Fennel gain function (1) has to ensure that the gain of a node on a coarser level corresponds to the sum of the gains of the nodes that it represents on the finest level. This way the algorithm gets a global view onto the optimization problem on the coarser levels and a very local view on the finer levels. More specifically, on coarser levels each vertex movement corresponds to the movement of a whole set of vertices from the original graph, which is not the case on the finest level. Moreover, it is ensured that on each level of the hierarchy the algorithm works towards optimizing the Fennel gain function (1) on the finest graph.

Our generalization of the gain function of Fennel is as follows. Let u be the node that should be assigned to a block. Our generalized Fennel assigns u to a block i that maximizes

$$\sum_{v \in V_i \cap N(u)} \omega(u, v) - c(u) f(c(V_i)), \tag{2}$$

such that $f(c(V_i)) = \alpha * \gamma * c(V_i)^{\gamma-1}$. Note that this is a direct generalization of the unweighted case. First, if the graph does not have edge weights, then the first part of the equation becomes $|V_i \cap N(u)|$ which is the first part of the Fennel objective. Second, if the graph also does not have node weights, then the second part of the equation is the same as the second part of the equation in the Fennel objective. Moreover, observe that the penalty term $f(c(V_i))$ in our objective is multiplied by $c(u)$. This is done to have the property stated above and formalized in Theorem 3.1. Finally, we keep the original value of the parameters α and γ in order to keep consistency.

Theorem 3.1. *If a set of nodes $S \subseteq V$ is contracted into a node w, the generalized* Fennel *gain function of w is equal to the sum of the generalized* Fennel *gain functions of the nodes in S.*

Proof. On the one hand, the generalized Fennel gain of assigning a node w to a block i is computed as

$$\sum_{v \in V_i \cap N(w)} \omega(w, v) - c(w) f(c(V_i)). \tag{3}$$

On the other hand, the sum of the generalized Fennel gains of assigning all nodes in S to a block i consists of

$$\sum_{u \in S} \sum_{v \in V_i \cap N(u)} \omega(u, v) - \sum_{u \in S} c(u) f(c(V_i)). \tag{4}$$

Since the factor $f(c(V_i))$ is identical in Equations (3) and (4), the two of them are equivalent if equalities (5) and (6) hold:

$$\sum_{v \in V_i \cap N(w)} \omega(w, v) := \sum_{u \in S} \sum_{v \in V_i \cap N(u)} \omega(u, v), \tag{5}$$

$$c(w) := \sum_{u \in S} c(u). \tag{6}$$

But equalities (5) and (6) are trivially true as a property of the contraction process. □

3.3.2 Multilevel Fennel. In principle, our model $\mathcal{B}$ could be partitioned by any multilevel partitioning algorithm which allows fixed nodes. Nevertheless, our preliminary tests using an adaptation of KaHIP [32] generated poor edge-cuts. This is the case because internal memory algorithms such as KaHIP and Metis [19] are designed to minimize edge-cut as much as possible while ensuring a low balancing constraint. This leads to two possibilities, both of which are bad for the overall edge-cut within the buffered streaming model: (i) using a large balancing constraint for the first buffers, the internal memory partitioner will tend to assign all their nodes to a single block, (ii) using a low balancing constraint for each batch, the overall partitioning problem will be aggressively over-restricted. Hence, we design a multilevel algorithm based on the Fennel function, which is a function that successively deals with the tradeoff between edge-cut and imbalance associated with graph streaming. We now explain our multilevel algorithm. Our *coarsening* phase is based on an adapted version of the size-constraint label propagation approach [23]. To be self-contained, we briefly outline the coarsening approach and then show how to modify it to be able to handle artificial nodes. To compute a graph hierarchy, the algorithm computes a size-constrained clustering on each level and contracts that to obtain the next level. The clustering is contracted by replacing each of its clusters by a single node (as exemplified in Figure 2), and the process is repeated recursively until the graph becomes small enough. This hierarchy is then used by the algorithm. Due to the way we define contraction, it ensures that a partition of a coarse graph corresponds to a partition of all the finer graphs in the hierarchy with the same edge-cut and balance. Note that cluster contraction is an aggressive coarsening strategy. In contrast to matching-based approaches, it enables us to 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.

The algorithm to compute clusters is based on *label propagation* [29] and avoids large clusters by using a *size constraint*, as described in [23]. For a graph with n nodes and m edges, one round of size-constrained label propagation can be implemented to run in $O(n+m)$ time. Initially, each node is in its own cluster/block, i.e., the initial block ID of a node is set to its node ID. The algorithm then works in rounds. In each round, all the nodes of the graph are traversed. 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. We perform at most L rounds, where L is a tuning parameter.

In HeiStream, we have to ensure that two artificial nodes are not contracted together since each of them should remain in its previously assigned block. We achieve this by ignoring artificial nodes and artificial edges during the label propagation, i.e., artificial nodes can not change their label and nodes from the batch can not change their label to become a label of an artificial node. As a consequence, artificial nodes are not contracted during coarsening. Overall, we repeat the process of computing a size-constrained clustering and contracting it, recursively. As soon as the graph is small enough, i.e., it has fewer nodes than an $O(\max(|\mathcal{B}|/k, k))$ threshold, it is initially partitioned by an initial partitioning algorithm. More precisely, we use the threshold $\max(\frac{|\mathcal{B}|}{2xk}, xk)$, in which x is a tuning parameter. Note that, for large enough buffer sizes, this threshold will be $O(|\mathcal{B}|/k)$.

When the coarsening phase ends, we run an *initial partitioning algorithm* to compute an initial k-partition for the coarsest version of $\mathcal{B}$. That means that all nodes other than the artificial nodes, which are already assigned, will be assigned to blocks. To assign the nodes, we run our generalized Fennel algorithm with explicit balancing constraint $L_{\max}$, i.e., the weight of no block will exceed $L_{\max}$. To be precise, a node u will be assigned to a block i that maximizes $\sum_{v \in V_i \cap N(u)} \omega(u,v) - c(u)f(c(V_i))$, such that $f(c(V_i)) = \alpha * \gamma * c(V_i)^{\gamma-1}$ *and* $c(V_i \cup u) \leq L_{\max}$. Note that the algorithm at this point considers all possible blocks $i \in \{1,\ldots,k\}$ and hence has complexity proportional to k. However, as the coarsest graph has $O(|\mathcal{B}|/k)$ nodes, overall the initial partitioning needs time which is linear in the size of the input model. When initial partitioning is done, we transfer the current solution to the next finer level by assigning each node of the finer level to the block of its coarse representative. At each level of the hierarchy, we apply a *local search* algorithm. Our local search algorithm is the same size-constraint label propagation algorithm we used in the contraction phase but with a different objective function. Namely, when visiting a non-artificial node, we remove it from its current block and then we assign it to the neighboring block which maximizes the generalized Fennel gain function defined above. Note that, in contrast to the initial partitioning, only blocks of adjacent nodes are considered here. Hence, one round of the algorithm can still be implemented to run in linear time in the size the current level. As in the coarsening phase, artificial nodes cannot be moved between blocks. Differently though, we do not exclude the artificial nodes from the label propagation here. This is the case because the artificial nodes and their edges are used to compute the generalized Fennel gain function of the other nodes. As in the initial partitioning, we use the explicit size constraint $L_{\max}$ of G. As a side note, we also tried to use high-quality offline algorithms as initial partitioning algorithms, however, in preliminary experiments this results in very unbalanced blocks (even with adaptively configured balance constraints) and overall in reduced quality throughout the process. Hence, we did not consider this further.

Assuming geometrically shrinking graphs throughout the hierarchy and assuming that the buffer size δ is larger than the number of blocks k, then the overall running time to partition a batch is linear in the size of the batch. This is due to the fact that the overall running time of

coarsening and local search sums up to be linear in the size of the batch, while the overall running time of the initial partitioning depends linearly on the size of the input model. Summing this up overall batches yields an overall linear running time $O(n + m)$.

3.4 Restreaming

We now extend HeiStream to operate in a restreaming setting. During restreaming, the overall structure of the algorithm is roughly the same. Nevertheless, we need to implement some adaptations which we explain in this section. The first adaptations concern model construction. Recall that the nodes from the current batch are already assigned to blocks during the previous pass of the input. We explicitly assign these nodes to their respective blocks in $\mathcal{B}$. Furthermore, ghost nodes and edges are not needed to construct $\mathcal{B}$. This is the case since all nodes from future batches are already known and assigned to blocks, i.e., these nodes will be represented in the artificial nodes. More precisely, we adapt the artificial nodes to represent the nodes from all batches except the current one. Since a partition of the graph is already given, we do not allow the contraction of cut edges during restreaming in the coarsening phase of our multilevel scheme. That means that clusters are only allowed to grow inside blocks. As a consequence, we can directly use the partition computed in the previous pass as initial partitioning for $\mathcal{B}$ so we do not need to run an initial partitioning algorithm.

3.5 Implementation Details

Our implementation of $\mathcal{B}$ is based on an adjacency array and consecutive node IDs. We reserve the first δ IDs for the nodes from the current batch, which keep their global order. This means that, when we process the ith batch, nodes IDs can be easily converted from our model $\mathcal{B}$ to G and the other way around by, respectively, summing or subtracting $(i - 1) * \delta$ on their ID. Similarly, we reserve the last k IDs of $\mathcal{B}$ for the artificial nodes and keep their relative order for all batches. Note that this configuration separates *mutable* nodes (nodes from the current batch) and *immutable* nodes (artificial nodes). This allows us to efficiently control which nodes are allowed to move during coarsening, initial partitioning, and local search. We keep an array of the size n store the permanent block assignment of the nodes of G. To improve running time, we use approximate computation of powering in our Fennel function.

4 EXPERIMENTAL EVALUATION

Methodology. We performed the implementation of HeiStream and competing algorithms inside the KaHIP framework (using C++) and compiled them using gcc 9.3 with full optimization turned on (-O3 flag). Since no official versions of the one-pass streaming and restreaming algorithms are available in public repositories, we implemented them in our framework. Our implementations of these algorithms reproduce the results presented in the respective articles and are optimized for running time as much as possible. To this end, we implemented Hashing, LDG, Fennel, and ReFennel. Multilevel LDG [18] is also not publicly available. We sent a message to the authors requesting an executable version of their algorithm for our tests but we have not received any response up to the moment this work was submitted. Hence, we compare HeiStream against Multilevel LDG based on the results explicitly reported in [18]. We have used two machines. Machine A has a two six-core Intel Xeon E5-2630 processors running at 2.8 GHz, 64 GB of main memory, and 3 MB of L2-Cache. It runs Ubuntu GNU/Linux 20.04.1 and Linux kernel version 5.4.0-48. Machine B has a four-core Intel Xeon E5420 processor running at 2.5 GHz, 16 GB of main memory, and 24 MB of L2-Cache. The machine runs Ubuntu GNU/Linux 20.04.1 and Linux kernel version 5.4.0-65. Most of our experiments were run on a single core of Machine A. The only exceptions are the experiments with huge graphs, which were run on a single core of Machine B. When using

machine A, we stream the input directly from the internal memory, and when using machine B, that only has 16 GB of main memory, we stream the input from the hard disk.

We use $k \in \{2, 3, \ldots, 128\}$ for most experiments. We allow an imbalance of 3% for all experiments (and all algorithms). All partitions computed by all algorithms have been balanced. Depending on the focus of the experiment, we measure running time and/or edge-cut. In general, we perform ten repetitions per algorithm and instance using different random seeds for initialization, and we compute the arithmetic average of the computed objective functions and running time per instance. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*. Unless explicitly mentioned otherwise, we average all results of each algorithm grouped by k. For a k_o-partition generated by an algorithm A, we express its score σ_A (which can be edge-cut or running time) using one or more of the following tools: *improvement* over an algorithm B, computed as $(\frac{\sigma_B}{\sigma_A} - 1) * 100\%$; *ratio*, computed as $(\frac{\sigma_A}{\sigma_{max}})$ with σ_{max} being the maximum score for k_o among all competitors including A; *relative* value over an algorithm B, computed as $(\frac{\sigma_A}{\sigma_B})$. We also present *performance profiles* which relate the running time (respectively, solution quality) of a group of algorithms to the fastest (respectively, best) one on a per-instance basis (rather than grouped by k). Their x-axis shows a factor τ while their y-axis shows the percentage of instances for which A has up to τ times the running time (respectively, solution quality) of the fastest (respectively, best) algorithm.

Instances. We get graphs from various sources to test our algorithm [4, 13, 17, 20, 30]. Most of the considered graphs were used for benchmark in previous works on graph partitioning. The graphs wiki-Talk and web-Google, as well as most networks of co-purchasing, roads, social, web, autonomous systems, citations, circuits, similarity, meshes, and miscellaneous are publicly available either in [20] or in [30]. Prior to our experiments, we converted these graphs to a vertex-stream format while removing parallel edges, self-loops, and directions, and assigning unitary weight to all nodes and edges. We also use graphs such as eu-2005, in-2004, uk-2002, and uk-2007-05, which are available at the 10th DIMACS Implementation Challenge website [4]. Finally, we include some artificial random graphs. We use the name `rggX` for a *random geometric graph* with 2^X nodes where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. We use the name `delX` for a graph based on a Delaunay triangulation of 2^X random points in the unit square [17]. We use the name `RHGX` for random hyperbolic graphs [13, 27] with 10^8 nodes and $X \times 10^9$ edges. The basic properties of the graphs under consideration can be found in Table 1. For our experiments, we split the graphs in three disjoint sets. A *tuning* set for the parameter study experiments, a *test* set for the comparisons against the state-of-the-art, and a set of *huge graphs* for special larger-scale tests. In any case, when streaming the graphs we use the natural given order of the nodes.

4.1 Parameter Study

We now present experiments to tune HeiStream and explore its behavior. We do this on the tuning set of our instance set. In our strategy, each experiment focuses on a single parameter of the algorithm while all the other ones are kept invariable. We start with a baseline configuration consisting of the following parameters: 5 rounds in the coarsening label propagation, 1 round in the local search label propagation, and $x = 64$ in the expression of the coarsest model size. After each tuning experiment, we update the baseline to integrate the best-found parameter. Unless explicitly mentioned otherwise, we run the experiments of this section overall tuning graphs from Table 1 based on the *extended* model construction, i.e., including ghost nodes and edges, for a buffer size of 32,768.

Tuning. We begin by evaluating how the number of label propagation rounds during a local search affects running time and solution quality. In particular, we run configurations of HeiStream with 1, 5, and 25 rounds and report results in Figure 5(a) and (b). Observe that the results of the

Table 1. Graphs for Experiments

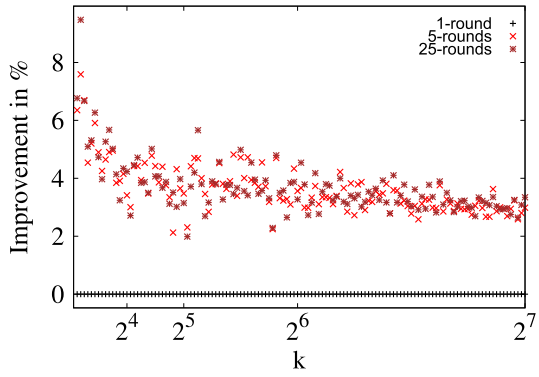
Graph	n	m	Type
Tuning Set			
coAuthorsCiteseer	227,320	814,134	Citations
citationCiteseer	268,495	1,156,647	Citations
amazon0312	400,727	2,349,869	Co-Purch.
amazon0601	403,364	2,443,311	Co-Purch.
amazon0505	410,236	2,439,437	Co-Purch.
roadNet-PA	1,087,562	1541514	Roads
com-Youtube	1,134,890	2,987,624	Social
soc-lastfm	1,191,805	4,519,330	Social
roadNet-TX	1,351,137	1,879,201	Roads
in-2004	1,382,908	1,3591,473	Web
G3_circuit	1,585,478	3,037,674	Circuit
soc-pokec	1,632,803	2,2301,964	Social
as-Skitter	1,694,616	11,094,209	Aut.Syst.
wiki-topcats	1,791,489	28,511,807	Social
roadNet-CA	1,957,027	2,760,388	Roads
wiki-Talk	2,388,953	4,656,682	Web
soc-flixster	2,523,386	7,918,801	Social
del22	4,194,304	12582,869	Artificial
rgg22	4,194,304	30359,198	Artificial
del23	8,388,608	25,165,784	Artificial
rgg23	8,388,608	63,501,393	Artificial
Huge Graphs			
uk-2005	39,459,923	783,027,125	Web
twitter7	41,652,230	1,202,513,046	Social
sk-2005	50,636,154	1,810,063,330	Web
soc-friendster	65,608,366	1,806,067,135	Social
er-fact1.5s26	67,108,864	907,090,182	Artificial
RHG1	100,000,000	1,000,913,106	Artificial
RHG2	100,000,000	1,999,544,833	Artificial
uk-2007-05	105,896,555	3,301,876,564	Web

Graph	n	m	Type
Test Set			
Dubcova1	16,129	118,440	Meshes
hcircuit	105,676	203,734	Circuit
coAuthorsDBLP	299,067	977,676	Citations
Web-NotreDame	325,729	1,090,108	Web
Dblp-2010	326,186	807,700	Citations
ML_Laplace	377,002	13,656,485	Meshes
coPapersCiteseer	434,102	16,036,720	Citations
coPapersDBLP	540,486	15,245,729	Citations
Amazon-2008	735,323	3,523,472	Similarity
eu-2005	862,664	16,138,468	Web
web-Google	916,428	4,322,051	Web
ca-hollywood-2009	1,087,562	1,541,514	Roads
Flan_1565	1,564,794	57,920,625	Meshes
Ljournal-2008	1,957,027	2,760,388	Social
HV15R	2,017,169	162,357,569	Meshes
Bump_2911	2,911,419	62,409,240	Meshes
del21	2,097,152	6,291,408	Artificial
rgg21	2,097,152	14,487,995	Artificial
FullChip	2,987,012	11,817,567	Circuit
soc-orkut-dir	3,072,441	117,185,083	Social
patents	3,750,822	14,970,766	Citations
cit-Patents	3,774,768	16,518,947	Citations
soc-LiveJournal1	4,847,571	42,851,237	Social
circuit5M	5,558,326	26,983,926	Circuit
italy-osm	6,686,493	7,013,978	Roads
great-britain-osm	7,733,822	8,156,517	Roads

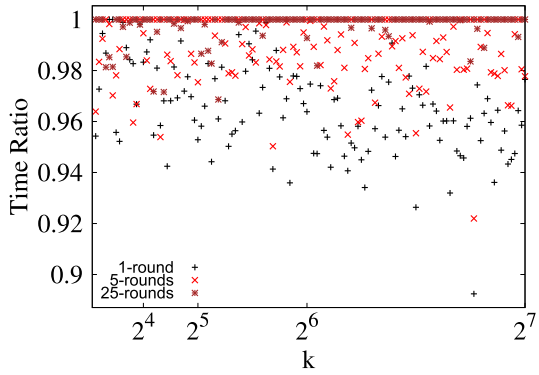
baseline have considerably lower solution quality than the other ones overall. On the other hand, the results of the configurations with 5 and 25 rounds differ slightly to each other. On average, they respectively improve solution quality 3.6% and 3.7% over the baseline. Regarding running time, they respectively increase 2.2% and 3.4% on average over the baseline. Since the variation of quality for these two configurations is not significant, we decided to integrate the fastest one among them in the algorithm, namely the 5-round configuration.

Next, we look at the parameter x associated with the expression $\max(|\mathcal{B}|/2xk, xk)$, which determines the size of the coarsest model. We run experiments for $x = 2^i$, with $i \in \{0, \ldots, 6\}$, and report results in Figure 5(c). We omit running time charts for this experiment since the tested configurations present comparable behavior in this regard. Figure 5(c) shows that the baseline presents the overall worst solution quality while $x = 2$ and $x = 4$ present the overall best solution quality. Averaging overall instances, these two latter configurations produce results, respectively, 10.3% and 11.2% better than the baseline. In light of that, we decide in favor of $x = 4$ to compose HeiStream.

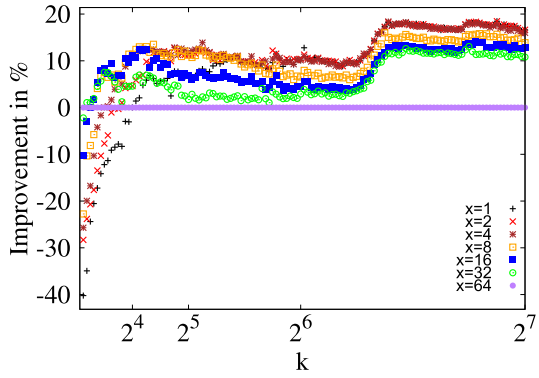
Exploration. We start the exploration of open parameters by investigating how the buffer size affects solution quality and running time. We use as a baseline a buffer of 8,192 nodes and successively double its capacity until any graph from the tuning set in Table 1 fits in a single buffer. We plot our results in Figure 6(b) and (c). Note that solution quality and running time increase regularly as the buffer size becomes larger. This behavior occurs because larger buffers enable



(a) Quality improvement plot for label propagation rounds during uncoarsening.

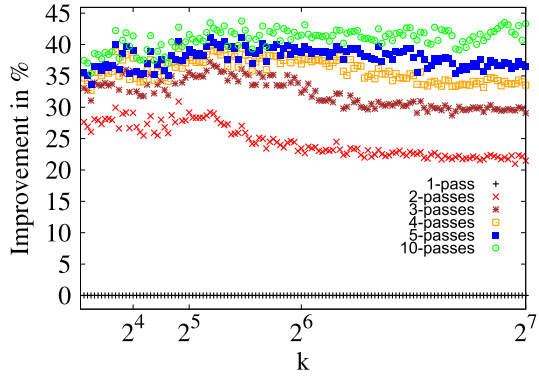


(b) Running time ratio plot for label propagation rounds during uncoarsening.

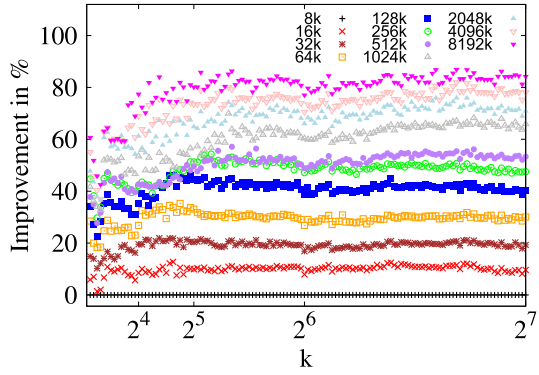


(c) Quality improvement plot for parameter x from expression $\max(|\mathcal{B}|/2xk, xk)$.

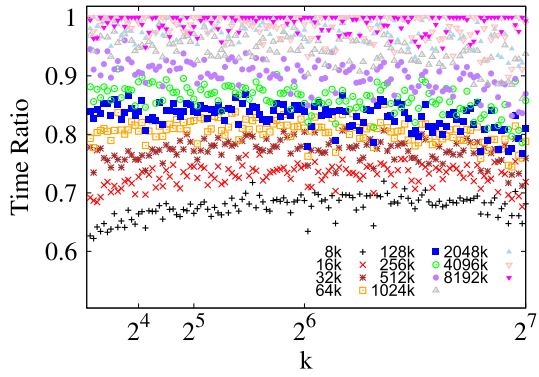
Fig. 5. Results for tuning and exploration experiments. Higher is better for quality improvement plots. Lower is better for running time ratio plots.



(a) Quality improvement plot for restreaming.

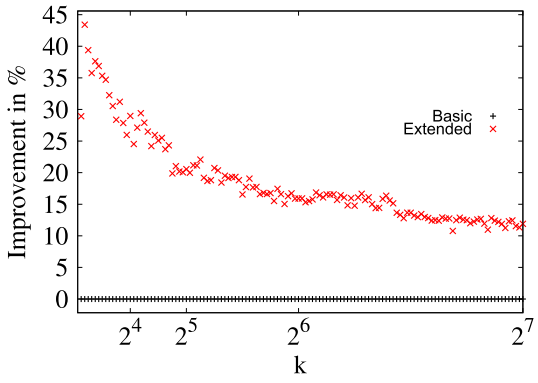


(b) Quality improvement plot for buffer size.

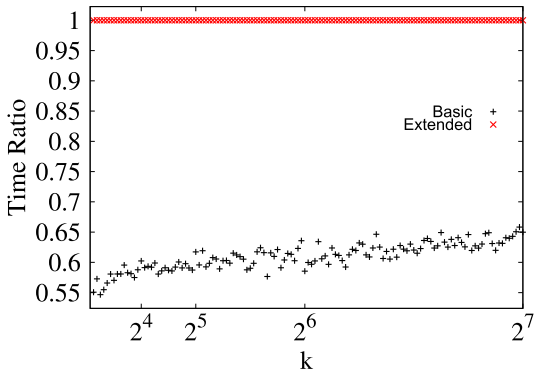


(c) Running time ratio plot for buffer size.

Fig. 6. Results for tuning and exploration experiments. Higher is better for quality improvement plots. Lower is better for running time ratio plots.



(a) Quality improvement plot for model construction.



(b) Running time ratio plot for model construction.

Fig. 7. Results for tuning and exploration experiments. Higher is better for quality improvement plots. Lower is better for running time ratio plots.

more comprehensive and complex graph structures to be exploited by our multilevel algorithm. As a consequence, there is a tradeoff between solution quality and resource consumption. In other words, we can improve partitioning quality at the cost of considerable extra memory and slightly more running time. Otherwise, we can save memory as much as possible and get a faster partitioning process at the cost of lowering solution quality. In practice, it means that HeiStream can be adjusted to produce partitions as refined as possible with the resources available in a specific system. For the extreme case of a single-node buffer, HeiStream behaves exactly as Fennel, while it behaves as an internal memory partitioning algorithm for the opposite extreme case.

Next, we compare the effect of using the *extended* model, which incorporates ghost nodes, over the *basic* model, which ignores ghost nodes. Figure 7(a) and (b) display the results. The results show that the extended model provides improved quality over the basic model, with an 18.3% improvement on average. This happens because the presence of ghost nodes and edges expands the perspective of the partitioning algorithm to future batches. This has a similar effect to increasing the size of the buffer, but at no considerable extra memory cost. Regarding running time, the results show that the extended model is consistently slower than the basic model for all values of k. Averaging overall instances, the extended model costs 63.9% more running time than the basic model. This increase in running time is explained by the higher number of edges to be processed

when ghost nodes are incorporated in the model. As a practical conclusion from the experiment, the extended model can be used for better overall partitions with no significant extra memory but at the cost of extra running time. Otherwise, the basic model can be used for a consistently faster execution at the cost of a lower solution quality.

Finally, we test to what extent solution quality can be improved by restreaming HeiStream multiple times. We investigate this by restreaming each input graph 10 times. We collect results after each pass and plot in Figure 6(a). The first restream generates a considerable quality jump, with an improvement over the baseline of 24.6% on average. Each following pass has a positive impact on solution quality, which converges to be a 40.9%-improvement on average over the baseline after the last pass. On the other hand, the running time has a roughly linear increase for each pass over the graph. In practice, this adds another degree of freedom to configure HeiStream for the needs of real systems.

4.2 Comparison Against State-of-the-Art

In this section, we show experiments in which we compare HeiStream against the current state-of-the-art algorithms. Unless mentioned otherwise, these experiments involve all the graphs from the Test Set group in Table 1 and focus on two particular configurations of HeiStream, which we refer to as HeiStream(32 k) and HeiStream(Int.). The first configuration is based on batches of size 32,768, while the second one has enough batch capacity to operate as an internal memory algorithm—both configurations perform a single pass over the input using the extended model. We also present results for the 2-pass restreaming version of HeiStream(32 k), which we refer to as 2-ReHeiStream(32 k).

Internal memory algorithms such as Metis [19] and KaHIP [32] are beyond the scope of this work since it is common knowledge that internal memory algorithms are better than streaming algorithms regarding partition quality if the instances fits in the memory of a machine. For the sake of reproducibility, we ran Metis and the fast social version of KaHIP over our Test Set group of instances for $k \in \{2, 4, 8, 16, 32, 64, 128\}$. On average, Fennel cuts a factor 7.5 more edges than Metis and the fast social version of KaHIP, while HeiStream(Int.) cuts a factor 2.2 more edges than both. Furthermore, Fennel is respectively a factor 2.2 and a factor 6.0 faster than Metis and the fast social version of KaHIP, while HeiStream(Int.) is 56.3% slower than Metis and 72.9% faster than the fast social version of KaHIP.

Results. We now present a detailed comparison of **HeiStream** (**HS**) against the state-of-the-art. In the results, we refer to the internal memory version of Multilevel LDG as MLDG(Int.). Moreover, we refer to the restreaming version of HeiStream and Fennel that passes over the graph 2 times as 2-ReHS and 2-ReFennel, respectively. First, we focus on $k = 32$ and later on choose a much wider range for the number of blocks. Table 2 shows the percentage of edges cut in the partitions generated by each algorithm for the graphs in the Test Set for $k = 32$. HeiStream(Int.), 2-ReHeiStream(32 k), and HeiStream(32 k) outperform all the other competitors for the majority of instances. First, all of them outperform Hashing for all graphs and LDG for 23 out of the 26 graphs. Next, they also outperform Fennel in 24 instances. The algorithm 2-ReFennel is outperformed by HeiStream(Int.), 2-ReHeiStream(32 k), and HeiStream(32 k) in 24, 25, and 17 instances, respectively. Considering only the 21 instances for which there are results reported for Multi.LDG(Int.) in literature, the algorithms HeiStream(Int.), 2-ReHeiStream(32 k), and HeiStream(32 k) compute better partitions for 18, 15, and 14 instances, respectively.

For a closer comparison against Multi.LDG(Int.), we present Figure 8. We plot edge-cut for Multi.LDG(Int.) is based on results graphically reported in [18]. In this figure, we show of HeiStream(Int.) and HeiStream(32 k) for five particular graphs with $k = 4, 8, 16, 32, 64, 128$. For all

Table 2. Edge-cut Results Against Competitors for $k = 32$

Graph	Cut Edges (%)							
	HS(Int.)	MLDG(Int.)	2-ReHS(32 k)	2-ReFennel	HS(32 k)	Fennel	LDG	Hashing
Dubcova1	**13.68**	14.26	**12.92**	29.19	**13.68**	33.99	33.96	95.62
hcircuit	**2.73**	17.75	**2.04**	21.86	**2.53**	28.97	28.97	90.75
coAuthorsDBLP	**15.99**	24.82	**16.90**	24.28	**17.80**	27.12	27.12	94.80
Web-NotreDame	**5.85**	11.01	**6.15**	12.58	**9.20**	19.52	19.56	95.97
Dblp-2010	**11.31**	18.52	**12.20**	22.93	**13.42**	28.82	28.80	92.49
ML_Laplace	**7.93**	13.44	**7.62**	7.82	8.36	7.92	**5.77**	96.37
coPapersCiteseer	**8.23**	11.22	**8.35**	9.63	**10.29**	12.88	12.27	96.52
coPapersDBLP	**14.51**	19.29	**15.12**	16.47	**18.33**	20.65	20.22	96.39
Amazon-2008	**10.09**	19.01	**12.77**	28.92	**15.56**	37.07	37.07	94.68
eu-2005	**11.14**	14.57	**14.82**	25.53	**18.64**	35.88	31.96	96.44
Web-Google	**1.62**	9.66	**3.96**	18.04	**9.48**	30.64	30.64	96.87
ca-hollywood-2009	35.34	**32.51**	**37.86**	41.34	**42.36**	44.54	45.25	96.62
Flan_1565	**7.69**	9.36	**7.30**	10.26	8.12	10.59	**6.70**	96.61
Ljournal-2008	**29.12**	34.76	**33.58**	43.23	**38.58**	51.43	51.36	96.07
HV15R	14.09	**11.48**	15.38	**15.05**	17.48	16.39	**15.49**	96.84
Bump_2911	**8.66**	11.73	**7.65**	8.61	**8.23**	8.65	8.30	96.19
FullChip	**38.20**	48.16	**42.24**	57.39	**45.71**	61.93	64.23	95.06
patents	**15.57**	29.12	**41.75**	52.60	**60.56**	70.98	70.98	96.88
Cit-Patents	**15.75**	28.65	**39.67**	51.62	**60.76**	72.16	72.16	96.88
Soc-LiveJournal1	**29.72**	35.69	**29.39**	34.00	**35.62**	39.03	45.62	96.66
circuit5M	40.02	**34.60**	**39.20**	75.45	**41.00**	78.42	78.47	96.87
del21	**1.38**	-	**5.41**	33.52	**8.53**	40.21	40.21	93.39
rgg21	**1.53**	-	**1.34**	4.09	**1.52**	5.02	4.88	96.89
soc-orkut-dir	**37.86**	-	**43.19**	47.22	**54.76**	55.85	60.27	96.85
italy-osm	**0.13**	-	**1.19**	4.65	**1.34**	4.80	4.80	78.11
great-britain-osm	**0.16**	-	**1.43**	7.18	**1.63**	7.34	7.34	79.94

Internal memory algorithms are on the two left columns, restreaming algorithms are on the two subsequent columns, and streaming algorithms are on the four right columns. We, respectively, refer to setups of 2-ReHeiStream and HeiStream with specific buffer sizes as 2-ReHS(Xk) and HS(Xk), in which each buffer contains $X \times 1{,}024$ nodes. HS(Int.) uses a buffer size of n. We bold the best result for each graph for internal memory approaches, restreaming approaches, and streaming approaches. The results for Multilevel LDG for the five bottom graphs are missing, as the graphs where not part of their benchmark set. Lower is better.

these instances, HeiStream(Int.) outperforms Multi.LDG(Int.) by a considerable margin. Observe that HeiStream(32 k) outperforms the *internal memory* version of Multilevel LDG for the majority of instances. We omit additional comparisons against buffered versions of Multilevel LDG, since they provide lower quality than the internal memory version. We ran wider experiments over our whole Test Set for 127 different values of k. Figure 9 shows a quality improvement plot over Fennel and a relative running time plot. Figure 10 shows performance profiles for solution quality and running time. Observe that HeiStream(Int.) produces solutions with the highest quality overall. In particular, it produces partitions with the smallest edge-cut for almost 86.9% of the instances and improves solution quality over Fennel 195.0% on average. We now provide some results in which we exclude HeiStream(Int.), since it has access to the whole graph. The best algorithm is HeiStream(32 k), which produces the best solution quality for 63.3% of the instances and improves solution quality over Fennel 75.9% on average. It is followed by 2-ReFennel, which is the best-tested algorithm from the previous state-of-the-art. In particular, it computes the best partition for 26.8% of the instances and improves on average 19.2% over Fennel. LDG and Fennel come next. Particularly, LDG finds the best partition for 9.8% of the instances and improves on average 2.4% over Fennel. Fennel does not find the best partition for any instance. Finally, Hashing

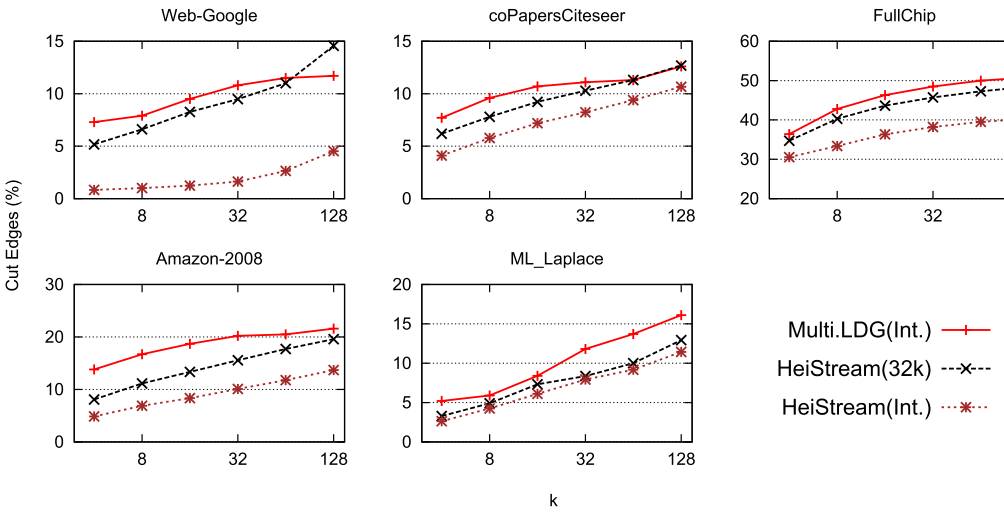
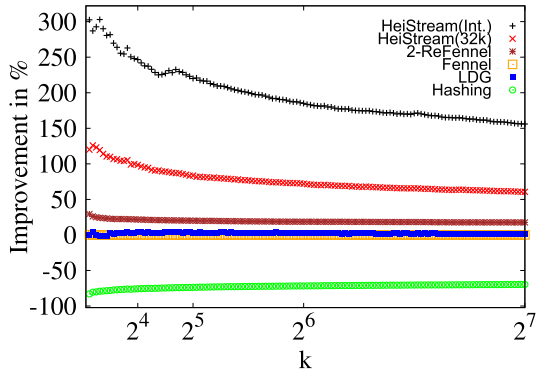


Fig. 8. Comparison of HeiStream against Multilevel LDG for buffer containing the whole graph.

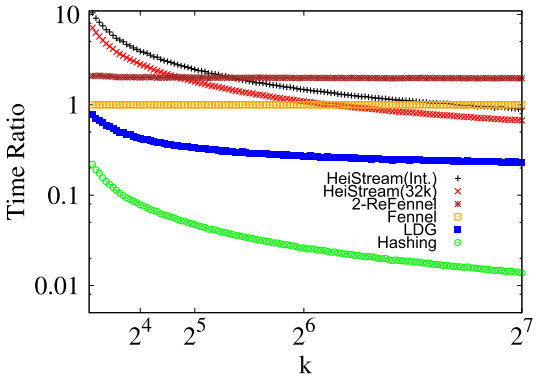
produces the worst solutions, with 72.5% worse quality than Fennel on average. Regarding running time, Hashing is the fastest one for all instances, which is expected since it is the only one with time complexity $O(n)$. The second fastest one is LDG, whose running time is a factor 9.6 higher than Hashing on average. Fennel, HeiStream(32 k) and HeiStream(Int.) come next with factors 32.4, 41.4, and 56.2 slower than Hashing on average, respectively. ReFennel is the slowest algorithm of the test, being a factor 64.61 slower than Hashing on average. In a direct comparison, HeiStream(32 k) and HeiStream(Int.) are, respectively, 27.7% and 73.2% slower than Fennel on average. Note that both configurations of HeiStream are faster than Fennel for larger values of k, which is consistent with the fact the running time of HeiStream is $O(n + m)$ while the running time of Fennel is $O(nk + m)$.

Restreaming. Now we directly evaluate the restreaming version of HeiStream. As shown in Table 2, 2-ReHeiStream(32 k) computes solutions with smaller edge-cut than 2-ReFennel for almost all our Test Set when $k = 32$. Figure 11 shows that this is the case for $k \in \{2, \ldots, 128\}$. In particular, 2-ReHeiStream(32 k) produces the best solution for 98.3% of the instances and improves solution quality over 2-ReFennel by 79.6% on average. The average improvement of 2-ReHeiStream(32 k) over 2-ReFennel is comparable to the average improvement of HeiStream(32 k) over Fennel. Nevertheless, the percentage of instances for which 2-ReHeiStream(32 k) outperforms 2-ReFennel is almost 100%, which means that 2-ReHeiStream(32 k) almost dominates 2-ReFennel with respect to edge-cut. Note that this is considerably higher than the percentage of instances for which HeiStream(32 k) outperforms Fennel (63.3%).

Visualization. As shown, the edge-cut of partitions produced by HeiStream is on average lower than the edge-cut of partitions produced by its competitor streaming algorithms. We shortly look at some visualizations in order to concretely understand why this happens. In Figure 12, we show a visual comparison of some partition layouts generated by Hashing, Fennel, HeiStream and the fast social version of KaHIP for the graph rgg15. Since this graph has only 32,768 nodes, we use a buffer size of 1,024 nodes for HeiStream in order to partition the graph over multiple successive batches. There is a leap of partitioning quality from Hashing to Fennel, i.e., well-delimited clusters associated to a same block can be identified in the partitions generated by Fennel but not in partitions generated by Hashing. Similarly there is a leap of partitioning quality from Fennel to



(a) Quality improvement plot over Fennel.



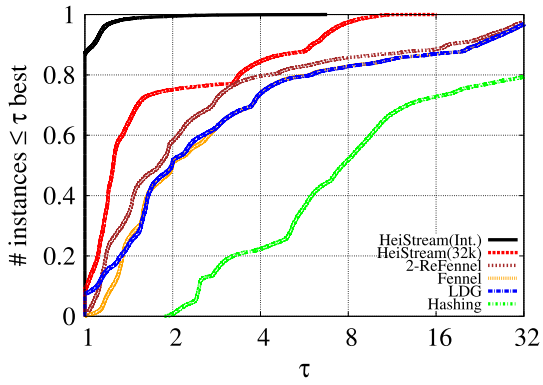
(b) Relative running time plot over Fennel.

Fig. 9. Results for comparison against state-of-the-art one-pass (re)streaming algorithms. Higher is better for quality improvement plots. Lower is better for running time.

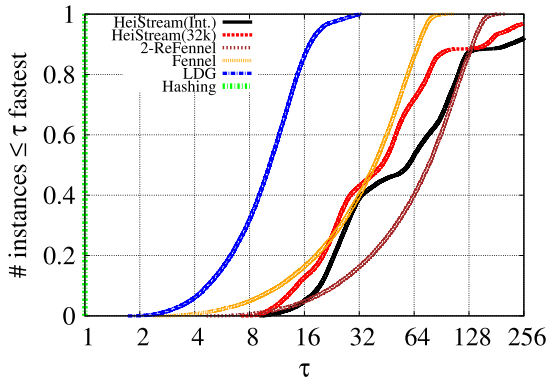
KaHIP, i.e., a block generated by Fennel consists of multiple small clusters that are not mutually connected while a block generated by KaHIP usually consists of a single connected cluster. Note that the partitions produced by HeiStream have intermediary characteristics between the partitions generated by Fennel and KaHIP. More specifically, a block generated by HeiStream consists of fewer and larger clusters than a block generated by Fennel but not as few and as large clusters as those generated by KaHIP. This behavior is a direct consequence of the more or less global view provided by the distinct computational models used by these three algorithms.

4.3 Huge Graphs

We now switch to the main use case of streaming algorithms: computing high-quality partitions for huge graphs on small machines. The experiments in this section are based on the *huge graphs* listed in Table 1 and are run on the relatively small Machine B. Namely, we ran experiments for $k = \{8, 16, 32, 64, 128, 256\}$ and we did not repeat each test multiple times with different seeds as in previous experiments. We also ran Metis and KaHIP on those graphs on this machine, but they fail on all instances since they require more memory than the machine has. For all instances, HeiStream performs a single pass over the input based on the extended model construction. We refer to setups of HeiStream with specific buffer sizes as HeiStream(Xk), in which a buffer contains



(a) Quality performance profile.



(b) Running time performance profile (Hashing is fully located on the far left since it is the fastest algorithm for all instances).

Fig. 10. Results for comparison against state-of-the-art one-pass (re)streaming algorithms using performance profiles. Higher is better for quality improvement plots.

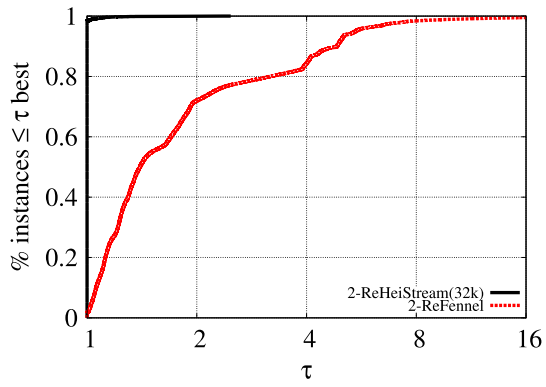


Fig. 11. Quality performance profile of 2-ReHeiStream(32 k) against 2-ReFennel.

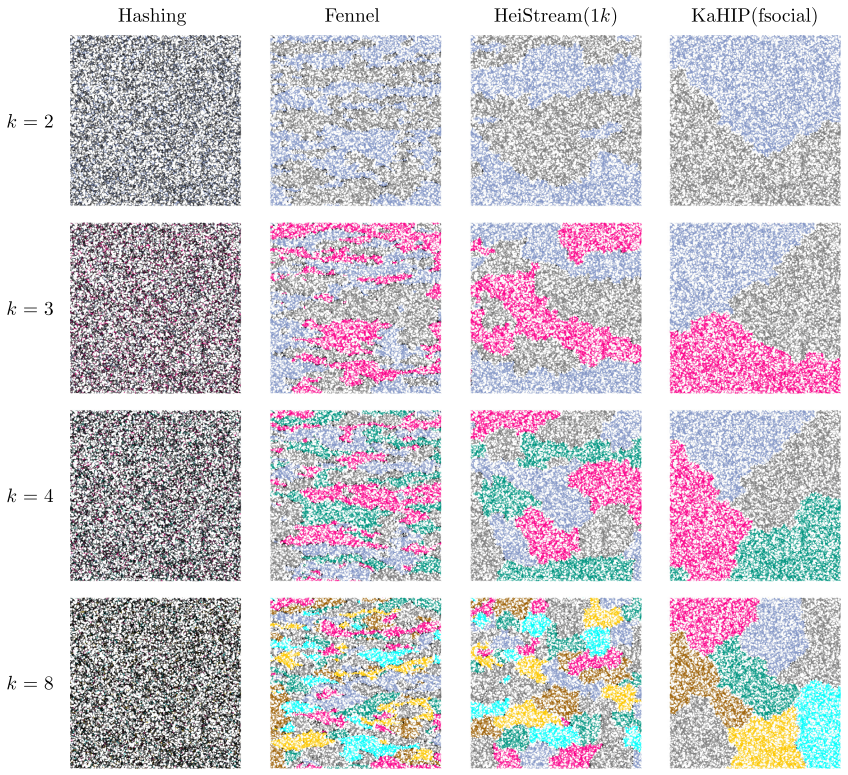


Fig. 12. Visualization of partitions generated by Hashing, Fennel, HeiStream($1k$) and the fast social version of KaHIP for the graph rgg15, which has 32,768 nodes and 160,240 edges.

$X \times 1{,}024$ nodes. In Table 3, we report detailed per-instance results with large buffer sizes able to run on Machine B. In particular, we partition all graphs with $X = 1,024$ except for twitter7, which we were not able to partition with a buffer larger than $X = 512$. We exclude from Table 3 the IO delay to load the input graph from the disk, since it depends on the disk and is roughly the same independently of the used partitioning algorithm. For completeness, we report this delay (in seconds) for the huge graphs listed in Table 1 following their respective order: 131.3, 203.2, 313.2, 294.0, 164.9, 186.1, 340.7, 551.5.

The results show that HeiStream outperforms all the competitor algorithms regarding solution quality for most instances. Notably, HeiStream computes partitions with considerably lower edge-cut in comparison to the one-pass algorithms for four out of the tested graphs: uk-2005, sk-2005, uk-2007-05, and RHG1. For the social networks soc-friendster and twitter7, HeiStream is the best for all instances, but the improvement over Fennel and LDG is not as large as in the other instances. One outlier can be seen on the network RHG2. While HeiStream produces fairly small edge-cut values, which are all below 0.7%, Fennel does outperform both and LDG improves solution quality even further on this instance. Furthermore, note that the running time of Fennel increases with increasing k to the point in which it becomes higher than the running time of HeiStream for five out of the eight huge graphs tested.

Memory Consumption. We now shortly review the amount of memory needed by the streaming algorithms under consideration. First of all note that the memory of HeiStream depends on the size of the buffer that is used. If the buffer only contains one node, then the memory requirements

Table 3. Results of Experiments with the Huge Graphs from Table 1

Graph	k	HeiStream(Xk)			Fennel		LDG		Hashing	
		X	CE(%)	RT(s)	CE(%)	RT(s)	CE(%)	RT(s)	CE(%)	RT(s)
uk-2005	8	1,024	**4.03**	290.23	19.93	37.19	19.97	19.36	73.70	3.28
	16	1,024	**6.01**	300.04	22.76	58.05	22.72	22.58	78.86	3.38
	32	1,024	**7.65**	310.72	25.24	98.78	25.19	29.19	81.39	3.30
	64	1,024	**8.99**	322.14	26.88	183.15	26.81	42.90	82.60	3.28
	128	1,024	**9.94**	346.73	27.89	342.74	27.76	61.87	83.18	3.27
	256	1,024	**10.68**	386.64	28.78	666.22	28.65	109.20	83.46	3.31
twitter7	8	512	**41.64**	1,727.13	45.18	184.17	56.11	180.85	71.66	3.46
	16	512	**47.04**	1,774.92	53.49	213.17	61.73	186.27	76.78	3.57
	32	512	**52.59**	1,884.16	58.15	244.18	66.84	184.90	79.34	3.49
	64	512	**57.53**	1,988.11	62.95	330.46	68.68	197.86	80.62	3.50
	128	512	**61.87**	2,113.34	66.68	504.53	69.94	219.65	81.26	3.93
	256	512	**65.47**	2,357.92	78.32	846.20	71.26	280.99	81.57	3.51
sk-2005	8	1,024	**3.23**	634.79	21.95	55.39	21.13	30.98	81.50	4.17
	16	1,024	**4.11**	648.48	26.36	82.41	25.33	35.44	87.26	4.20
	32	1,024	**5.32**	667.84	29.59	137.42	27.97	43.76	90.11	4.23
	64	1,024	**7.55**	695.60	32.52	238.54	30.18	59.19	91.50	4.21
	128	1,024	**8.95**	733.05	35.87	449.19	32.44	91.36	92.19	4.20
	256	1,024	**12.02**	798.73	40.06	857.76	35.69	150.64	92.55	4.26
soc-friendster	8	1,024	**27.36**	4,099.35	30.57	405.68	45.60	381.78	87.53	5.45
	16	1,024	**34.50**	4,202.04	45.74	440.11	58.98	361.54	93.77	5.62
	32	1,024	**39.52**	4,345.96	54.87	503.90	61.00	408.56	96.89	5.49
	64	1,024	**46.35**	4,546.98	59.27	649.34	64.02	422.87	98.45	5.44
	128	1,024	**52.41**	4,796.56	60.82	888.14	68.17	475.16	99.22	5.72
	256	1,024	**57.79**	5,323.08	64.25	1,426.16	71.90	523.66	99.61	5.53
er-fact1.5s26	8	1,024	**73.27**	2,216.99	73.44	259.98	73.44	208.81	87.50	5.57
	16	1,024	**80.18**	2,292.12	80.40	288.90	80.40	226.01	93.75	5.57
	32	1,024	**84.36**	2,400.35	84.63	357.21	84.63	255.60	96.87	5.60
	64	1,024	**86.99**	2,534.09	87.31	506.23	87.31	270.58	98.44	5.58
	128	1,024	**88.72**	2,725.81	89.10	769.61	89.10	407.59	99.22	5.57
	256	1,024	**89.99**	2,913.95	90.45	1,329.57	90.45	408.35	99.61	5.65
RHG1	8	1,024	**0.04**	380.04	2.02	86.95	2.02	44.42	91.91	8.37
	16	1,024	**0.06**	391.63	2.12	143.47	2.12	52.11	97.39	8.57
	32	1,024	**0.09**	406.56	2.16	252.15	2.16	65.65	99.19	8.38
	64	1,024	**0.15**	435.71	2.17	450.73	2.17	95.22	99.74	8.33
	128	1,024	**0.22**	482.06	2.18	877.69	2.18	147.22	99.90	8.31
	256	1,024	**0.34**	569.77	2.19	1,708.33	2.18	273.76	99.95	8.45
RHG2	8	1,024	0.09	621.56	0.05	103.83	**0.04**	56.23	89.71	8.31
	16	1,024	0.13	632.61	0.07	153.29	**0.04**	60.14	96.15	8.57
	32	1,024	0.19	648.68	0.12	262.91	**0.05**	77.34	98.73	9.85
	64	1,024	0.29	674.36	0.18	468.04	**0.05**	108.39	99.57	8.32
	128	1,024	0.44	727.66	0.28	872.68	**0.07**	157.75	99.85	8.31
	256	1,024	0.68	816.60	0.44	1,686.18	**0.09**	278.51	99.92	8.47
uk-2007-05	8	1,024	**0.54**	1,024.26	25.23	107.46	25.21	58.19	87.91	8.80
	16	1,024	**0.60**	1,045.36	28.02	166.06	28.19	70.72	94.08	8.79
	32	1,024	**0.70**	1,058.73	29.40	278.38	29.32	85.42	97.12	8.99
	64	1,024	**0.92**	1,099.64	29.94	517.20	29.85	115.18	98.61	9.32
	128	1,024	**1.31**	1,163.45	30.73	935.98	30.18	175.01	99.33	8.79
	256	1,024	**1.95**	1,280.48	31.65	1,808.52	30.70	324.71	99.68	8.92

For each instance, we present the percentage of **cut edges** (**CE**) and the **running time** (**RT**) in seconds. We bold the best result for each instance. We refer to setups of HeiStream with specific buffer sizes as HeiStream(Xk), in which each buffer contains $X \times 1{,}024$ nodes.

match those of Fennel and LDG. Here, we measure the memory consumption of HeiStream for various buffer sizes and compare it to Fennel and LDG. To do that, we measured the memory consumption of HeiStream(1,024 k), HeiStream(32 k), Fennel, and LDG on the three largest graphs (RHG1, RHG2, and uk-2007-05). On average, HeiStream(1,024 k) consumes, respectively, 2.5*GB*, 4.1*GB*, and 9.9*GB* of memory to partition these graphs. while HeiStream(32 k) consumes 472*MB*, 521*MB*, and 3.7*GB*, and Fennel and LDG use 399*MB*, 401*MB*, and 445*MB*. Note that the increased amount of used memory of HeiStream(1,024 k) is expected, since we use a fairly large buffer. Given the size of the graphs, we believe those required amounts of memory are more than reasonable.

5 CONCLUSION

We proposed HeiStream, a buffered streaming graph partitioning algorithm. We combined the buffered streaming model with multilevel graph partitioning techniques and extended Fennel to a multilevel algorithm. Compared to the previous state of-the-art of streaming graph partitioning, HeiStream computes significantly better solutions than known streaming algorithms while at the same time being faster in many cases. Note that improved edge cuts directly improve the communication cost in applications such as online queries on distributed graph databases [25]. Hence, our results directly yield improvements in applications. An important property of HeiStream is that its running time does not depend on the number of blocks, while the previous state-of-the-art streaming partitioning algorithms have running time almost proportional to this number of blocks.

SUPPLEMENTARY MATERIAL

The source code and experimental data are available online at https://heibox.uni-heidelberg.de/d/581b48a61a894a83a687/.

REFERENCES

[1] Zhao An, Qilong Feng, Iyad Kanj, and Ge Xia. 2020. The complexity of tree partitioning. *Algorithmica* 82, 9 (2020), 2606–2643.

[2] K. Andreev and H. Räcke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939. DOI: https://doi.org/10.1007/s00224-006-1350-7

[3] Amel Awadelkarim and Johan Ugander. 2020. Prioritized restreaming algorithms for balanced graph partitioning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1877–1887. DOI: https://doi.org/10.1145/3394486.3403239

[4] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. 2014. Benchmarking for graph clustering and partitioning. In *Proceedings of the Encyclopedia of Social Network Analysis and Mining*. Springer, 73–82. DOI: https://doi.org/10.1007/978-1-4939-7131-2_23

[5] C. Bichot and P. Siarry (Eds.). 2011. *Graph Partitioning*. Wiley. DOI: https://doi.org/10.1002/9781118601181

[6] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. 2007. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering* 20, 2 (2007), 172–188. DOI: https://doi.org/10.1109/TKDE.2007.190689

[7] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. *Recent Advances in Graph Partitioning*. Springer International Publishing, Cham, 117–158. DOI: https://doi.org/10.1007/978-3-319-49487-6_4

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[9] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 85–98. DOI: https://doi.org/10.1145/2168836.2168846

[10] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815. DOI: https://doi.org/10.14778/2824032.2824077

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113. DOI: https://doi.org/10.1145/1327452.1327492

[12] Gunduz Vehbi Demirci, Hakan Ferhatosmanoglu, and Cevdet Aykanat. 2019. Cascade-aware partitioning of large graph databases. *The VLDB Journal* 28, 3 (2019), 329–350. DOI: https://doi.org/10.1007/s00778-018-0531-8

[13] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. 2019. Communication-free massively distributed graph generation. *Journal of Parallel and Distributed Computing* 131 (2019), 200–217.

[14] M. R. Garey, D. S. Johnson, and L. Stockmeyer. 1974. Some simplified NP-complete problems. In *Proceedings of the 6th ACM Symposium on Theory of Computing*. ACM, 47–63. DOI: https://doi.org/10.1145/800119.803884

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 17–30. DOI: https://doi.org/10.5555/2387880.2387883

[16] B. Hendrickson and R. Leland. 1995. A multilevel algorithm for partitioning graphs. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM. DOI: https://doi.org/10.1145/224170.224228

[17] M. Holtgrewe, P. Sanders, and C. Schulz. 2010. Engineering a scalable high quality graph partitioner. *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*. 1–12.

[18] Nazanin Jafari, Oguz Selvitopi, and Cevdet Aykanat. 2021. Fast shared-memory streaming multilevel graph partitioning. *Journal of Parallel and Distributed Computing* 147 (2021), 140–151. DOI: https://doi.org/10.1016/j.jpdc.2020.09.004

[19] G. Karypis and V. Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

[20] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved June 2021 from http://snap.stanford.edu/data.

[21] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. 135–146. DOI: https://doi.org/10.1145/1807167.1807184

[22] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. 2018. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems*. IEEE, 685–695. DOI: https://doi.org/10.1109/ICDCS.2018.00072

[23] H. Meyerhenke, P. Sanders, and C. Schulz. 2014. Partitioning complex networks via size-constrained clustering, In *Proceedings of the International Symposium on Experimental Algorithms*. DOI:https://doi.org/10.1007/978-3-319-07959-2_30

[24] Joel Nishimura and Johan Ugander. 2013. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1106–1114. DOI: https://doi.org/10.1145/2487575.2487696

[25] Anil Pacaci and M. Tamer Özsu. 2019. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*. 1375–1392. DOI: https://doi.org/10.1145/3299869.3300076

[26] Md Anwarul Kaium Patwary, Saurabh Garg, and Byeong Kang. 2019. Window-based streaming graph partitioning algorithm. In *Proceedings of the Australasian Computer Science Week Multiconference*. 1–10. DOI: https://doi.org/10.1145/3290688.3290711

[27] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. 2022. *Recent Advances in Scalable Network Generation*. Book chapter for Massive Graph Analytics, Hall/CRC.

[28] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. 243–252. DOI: https://doi.org/10.1145/2806416.2806424

[29] U. N. Raghavan, R. Albert, and S. Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E* 76, 3 (2007), 036106. DOI: https://doi.org/10.1103/PhysRevE.76.036106

[30] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. Retrieved June 2021 from http://networkrepository.com.

[31] Hooman Peiro Sajjad, Amir H. Payberah, Fatemeh Rahimian, Vladimir Vlassov, and Seif Haridi. 2016. Boosting vertex-cut partitioning for streaming graphs. In *Proceedings of the 2016 IEEE International Congress on Big Data*. IEEE, 1–8. DOI: https://doi.org/10.1109/BigDataCongress.2016.10

[32] P. Sanders and C. Schulz. 2011. Engineering multilevel graph partitioning algorithms. In *Proceedings of the 19th European Symposium on Algorithms*. Vol. 6942. Springer, 469–480. DOI: https://doi.org/10.1007/978-3-642-23719-5_40

[33] C. Schulz and D. Strash. 2019. Graph partitioning: Formulations and applications to big data. In *Proceedings of the Encyclopedia of Big Data Technologies*. DOI: https://doi.org/10.1007/978-3-319-63962-8_312-2

[34] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1222–1230. DOI: https://doi.org/10.1145/2339530.2339722

[35] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. 333–342. DOI: https://doi.org/10.1145/2556195.2556213

Received 21 December 2021; revised 1 June 2022; accepted 28 June 2022