Boosting Vertex-Cut Partitioning For Streaming Graphs

Hooman Peiro Sajjad†, Amir H. Payberah‡, Fatemeh Rahimian‡, Vladimir Vlassov†, Seif Haridi‡
†KTH Royal Institute of Technology, Sweden
‡SICS Swedish ICT, Sweden
†{shps, vladv}@kth.se ‡{amir, fatemeh, seif}@sics.se

***Abstract*—While the algorithms for streaming graph partitioning are proved promising, they fall short of creating timely partitions when applied on large graphs. For example, it takes 415 seconds for a state-of-the-art partitioner to work on a social network graph with 117 millions edges. We introduce an efficient platform for boosting streaming graph partitioning algorithms. Our solution, called HoVerCut, is Horizontally and Vertically scalable. That is, it can run as a multi-threaded process on a single machine, or as a distributed partitioner across multiple machines. Our evaluations, on both real-world and synthetic graphs, show that HoVerCut speeds up the process significantly without degrading the quality of partitioning. For example, HoVerCut partitions the aforementioned social network graph with 117 millions edges in 11 seconds that is about 37 times faster.**

***Index Terms*—streaming graph; vertex-cut partitioning; graph partitioning; parallel scalability**

I. Introduction

Graph partitioning is an NP-Complete problem with a long history in graph theory [1], [2]. The traditional form of graph partitioning is *edge-cut* partitioning, which divides vertices of a graph into disjoint partitions of nearly equal size, while the number of edges that span partitions is minimum. There is another form of partitioning, based on *vertex-cut*, which divides the edges of a graph into nearly equal size partitions, such that the number of replicated (cut) vertices is minimum.

Most of the real-world graphs exhibit power-law degree distribution, i.e., the majority of vertices have relatively few neighbors, while a small fraction of them have many neighbors [3]. Several studies [4], [5], [6] have shown that the graph processing systems cannot achieve a good performance, when they apply edge-cut partitioning on power-law graphs. This is due to unbalanced number of edges in each partition. On the other hand, both theory [7] and practice [8], [9] prove that power-law graphs can be efficiently processed, if vertex-cuts are used. We, therefor, focus on vertex-cut partitioning algorithms.

Streaming graph partitioning is a new approach to graph partitioning, where graph elements (vertices or edges) are received continuously over time, and assigned to partitions as they are being streamed [10]. This is usually done in *one-pass*, i.e, while reading through the graph, edges and vertices are assigned to partitions once and for all. A few algorithms have recently been developed for streaming vertex-cut partitioning [11], [12], [13]. To the best of our knowledge, HDRF [12] and the PowerGraph Greedy algorithm [11] have shown the best partitioning results for power-law graphs.

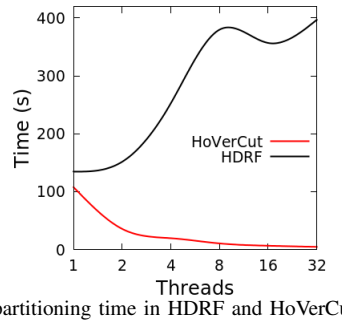


Fig. 1: The partitioning time in HDRF and HoVerCut with different number of threads for the graph of LiveJournal social network with 48M edges.

These algorithms create nearly equal size partitions with small number of replicated vertices. However, these systems are centralized and do not scale neither *horizontally* nor *vertically*. They are not horizontally scalable, because they are not designed to be deployed on a network or cluster of machines. They are also not vertically scalable, because they can not leverage the available computational resources of a machine to improve their performance. For example, as Figure 1 depicts, the partitioning time of HDRF drops dramatically, when it is implemented as a multi-threaded application.

In this paper, we present *HoVerCut*, a parallel and distributed vertex-cut partitioner for streaming graphs. HoVerCut uses multi-threading together with a windowing technique that brings about a light-weight state sharing between the threads. Each thread, called a *subpartitioner*, will run an instance of a partitioning algorithm (e.g., any of the existing ones [11], [12]), while receiving an exclusive subset of input edges. Subpartitioners do not have to be on a single machine only. In fact, HoVerCut can employ multiple machines in a distributed environment to better utilize the available resources.

Every subpartitioner has a local *state*, based on which it partitions the incoming edges. The local states include information about the processed edges and vertices, and can be shared on-demand, through a single shared state. To reduce the contention over the shared state, subpartitioners do not communicate with it every time they receive an edge. Instead, they collect a number of incoming edges in a window of a certain size, and then pull from the shared state the data that concerns the current window, in one batch. The result of the partitioning will also be pushed to the shared state in one batch. Note that it is likely to encounter the same vertices several time in a window, in particular the high degree vertices, which constitute the most demanding part of the shared state. Therefore, HoVerCut not only reduces the number

of communications between subpartitioner and the shared state, but also the size of information that is pulled or pushed. This results in a significant improvement in partitioning time, which is the main contribution of our work.

Moreover, by decoupling the state from the actual partitioning algorithm, HoVerCut facilitates the parallel as well as distributed implementation of any of the existing partitioning algorithms, without degrading the partitioning quality. This means even in presence of network latency, these algorithms can gain a remarkable speedup if used together with HoVerCut.

We perform a comprehensive evaluation of HoVerCut with two of the exisiting heuristics for vertex-cut partitioning [11], [12]. We use both synthetic and real world datasets, and show that HoVerCut produces partitions as good as the original heuristics, but tens of times faster in both single machine and distributed configurations. For example, the implementation of HDRF in HoVerCut with 32 threads, partitions the Orkut social network graph (with 3.1 million vertices and 117 millions edges) 37 times faster than the original HDRF.

The paper is structured as follows. In section II, we define the problem of streaming graph partitioning and its challenges. In section III, we present HoVerCut and elaborate on the state sharing and the partition selection policies. We present the evaluation results in section IV. Our study on the related work is presented in section V. We conclude the paper in section VI.

II. Problem Definition

We address the problem of *k-way vertex-cut partitioning* for *streaming graphs*, in a parallel and/or distributed environment.

A. Vertex-cut Partitioning

To formally define the k-way vertex-cut partitioning, let us define a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. We consider a partition function $\pi : E \rightarrow \{1, \ldots, k\}$ that assigns a *partition* to each edge, where $\pi(e)$, or π_e for short, refers to the partition of edge e. When an edge is assigned to a partition, the two vertices at its endpoints are also assigned to that partition. We denote the set of edges that are connected to vertex v by E_v. Since the edges in E_v may be scattered across partitions, vertex v might have a copy in multiple partitions. $E_v(p)$ indicates the subset of edges incident with v that belong to partition p:

$$E_v(p) = \{e \in E_v : \pi_e = p\} \tag{1}$$

The number of copies of vertex v (replicas) across all partitions, is denoted by $RF(v, \pi)$ and calculated by:

$$RF(v, \pi) = \sum_{|E_v(p)|>0} 1, \forall\ p \in \{1, \ldots, k\} \tag{2}$$

The *Replication Factor (RF)* of the graph, $RF(G, \pi)$, is then defined as the average of the replicas of all the vertices:

$$RF(G, \pi) = \frac{\sum_{v \in V} RF(v, \pi)}{|V|} \tag{3}$$

We want to minimize the RF, subject to a constraint on the balanced size of the partitions. If we denote size of partition p as $|E(p)|$ (the number of edges in p), we can formulate an optimization problem as follows:

$$\begin{aligned} \pi^* &= \arg\min_\pi RF(G, \pi) \\ s.t.\ \ &|E(p_1)| = |E(p2)|, \forall\ p_1, p_2 \in \{1, \ldots, k\} \end{aligned} \tag{4}$$

B. Data Processing Model and Assumptions

We assume that edges of the graph are read in a streaming fashion. They can be generated at some source on the fly, or can be read from a disk, or a cluster. In the former case, part of the graph data is not generated yet, thus, we can only process partial data. In the latter cases, the whole graph already exists, but will not be loaded into memory all at once (e.g., the size of the graph is big to fit in the memory).

We assume that we do not have access to the entire graph, and therefore, we can not perform global operations. At every point in time we only have access to a subset of edges that are read latest, and if needed, some aggregate values or states about what we have read so far. The stream could have either a single source or multiple sources. In case of multiple sources, we process the received data in parallel, and strict synchronization between the parallel processing units is not required.

III. The HoVerCut Framework

HoVerCut is a parallel and distributed vertex-cut partitioning platform for streaming graphs. It runs as a multi-threaded process on a single machine, or as a distributed partitioner across a cluster of machines. All the threads apply the same partitioning algorithm on a subset of the graph edges. This means we can have multiple streaming sources, or we can load a graph from disk using parallel loaders. We call each instance (thread) a *subpartitioner*. Each subpartitioner receives a subset of edges over time, and assigns them to partitions based on a given heuristic, referred to as the *partitioning policy*. We can use different partitioning policies in HoVerCut, e.g., Greedy [11] or HDRF [12].

HoVerCut decouples the partitioning policy from the partitioning state, to enable efficient parallelism. In a distributed configuration, subpartitioners can access the partitioning state remotely, but their algorithm would be the same. Each subpartitioner holds a local state, which contains the information required by the partitioning policy. For example in case of HDRF policy, the state includes (i) the partial degree of vertices that have been processed so far, (ii) the partitions they are assigned to, and (iii) the current size of partitions. Further, HoVerCut maintains the global system *state* in a *shared storage* accessible by all subpartitioners. Subpartitioners read and update the shared state periodically and asynchronously. They use this information to decide about the partitions of receiving edges. The more subpartitioners refer to the shared state, the more updated information they receive, and consequently better decisions they make, but the slower the system will be. Therefore, instead of contacting the shared state for each incoming edge, subpartitioners access the shared state for a *window* of edges. As a result, subpartitioners access the shared state less frequently and pull/push information in batches. This

Algorithm 1 The HoVerCut core algorithm.

```
1: procedure MAIN
2:     while loader.hasnext() do
3:         e ← loader.nextEdge()                    ▷ read the next edge
4:         window.add(e)
5:         e.src.incDegree()
6:         e.dest.incDegree()
7:         vertices.update(e.src)
8:         vertices.update(e.dest)
9:         if trigger(window) then
10:            partitionWindow(window, vertices)
11:            window.clear()
12:            vertices.clear()
13:        end if
14:    end while
15: end procedure
```

Algorithm 2 The algorithm for partitioning a window of edges.

```
1: procedure PARTITIONWINDOW(Set edges, Set vids)
2:     state ← SharedState.getState(vids)
3:     for e ∈ edges do
4:         u ← state.vertices.get(e.src)   ▷ the degree and partitions of e.src
5:         v ← state.vertices.get(e.dest)
6:         u.updateDegree(u.degree, vids.vertice(u).degree)
7:         v.updateDegree(v.degree, vids.vertice(v).degree)
8:         p ← selectPartition(u, v, state.partitions)
9:         u.addPartition(p)
10:        v.addPartition(p)
11:        p.incrementSize()
12:        state.update(u, v, p)
13:    end for
14:    SharedState.putState(state)
15: end procedure
```

reduces the latency entailed by the network communication and the overall size of transferred information in a distributed HoVerCut.

The generic HoVerCut framework is illustrated in Algorithms 1 and 2. Each subpartitioner executes Algorithm 1 locally to make a window (buffer) of incoming edges. Upon receipt of a new edge e at a subpartitioner, the local state is partially updated, e.g., the local degree of visited vertices is incremented.

When a window reaches a certain condition (based on time or the number of buffered edges), the subpartitioner reads the information of the buffered edges and their corresponding vertices from the shared state (Algorithm 2). The subpartitioner, then, iterates through the edges in its buffer, and for each edge e, calls the `selectPartition` method. In case of using HDRF policy (Algorithm 3), the `selectPartition` method takes as input the latest partition sizes (received from the shared storage) and the information about the two endpoint of an edge e (their partial degree, and the current partitions they are assigned to). We can use any other policy to define the `selectPartition` method. When all the edges in a buffer are assigned to partitions, the subpartitioner writes back the modified part of its local state to the shared storage.

A. Partitioning Policy

We can use any vertex-cut partitioning heuristics in HoVerCut, e.g., DBH [13], PowerGraph Greedy [11], or HDRF [12]. DBH [13] indicates that replication factor (RF) of a power-law graph can be reduced if vertices with relatively higher degrees

Algorithm 3 HDRF as partition selection policy.

```
1: procedure SELECTPARTITION(Vertex u, Vertex v, Set partitions)
2:     maxP ← findMaxPartitionSize(partitions)
3:     minP ← findMinPartitionSize(partitions)
4:     for p ∈ partitions do
5:         rScore ← replicationScore(u, v, p)          ▷ Eq. 6 and 8
6:         bScore ← balanceScore(p, maxP.size, minP.size)   ▷ Eq. 10
7:         scores.set(p, rScore + bScore)                   ▷ Eq. 5
8:     end for
9:     p ← findMaxScore(scores)
10: return p;
11: end procedure
```

are cut. Therefore, for a given edge with two end vertices, DBH chooses the identifier of a vertex with lower degree to create a hash value and assigns the edge to a partition with the correspondent identifier. To implement this heuristic, we need to know the degree of vertices, which is not available in a one-pass streaming graph partitioning. Inspired by Petroni et al. [12], we can use *partial degree* of vertices (number of times that a vertex has been encountered in incoming edges).

The idea of DBH [13] is enhanced in Greedy [11] and HDRF [12] heuristics by adding the impact of the partitions sizes. For an edge with two end vertices u and v, and for each partition p, Greedy and HDRF compute a score $S(u, v, p)$, and select a partition that maximizes the score. $S(u, v, p)$ is computed as bellow:

$$S(u, v, p) = S_R(u, v, p) + S_B(p) \tag{5}$$

where S_R is formulated to reduce the RF, and S_B is computed based on the size of partitions. The difference between the Greedy and HDRF is in the way they compute S_R and S_B. To compute S_R, the Greedy heuristic gives a higher score to partitions that have replicas of the end vertices u and v, while HDRF gives a higher score to partitions that have a replica of lower degree vertices. For both the Greedy and HDRF we can define S_R as follows:

$$S_R(u, v, p) = g(v, u, p) + g(u, v, p) \tag{6}$$

where $g(v, u, p)$ in Greedy heuristic is:

$$g^{Greedy}(v, u, p) = \begin{cases} 1, & \text{if } p \in P(v). \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

and in HDRF is:

$$g^{HDRF}(v, u, p) = \begin{cases} 1 + \frac{d(v)}{d(v)+d(u)}, & \text{if } p \in P(v). \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

where $P(v)$ is set of partitions that vertex v has been replicated in and $d(v)$ is the partial degree of vertex v. S_B, in both the Greedy and HDRF, is formulated according to number of edges in partition p compared to the partitions with maximum (p_{max}) and minimum (p_{min}) number of edges. The difference is an added weight (λ) in HDRF that controls the importance of load balancing. S_B for a partition p in the Greedy heuristic is as follows:

$$S_B^{Greedy}(p) = \frac{|p_{max}| - |p|}{1 + |p_{max}| - |p_{min}|} \tag{9}$$

Dataset	$\|V\|$	$\|E\|$
Autonomous systems (AS) [14]	1.7M	11M
Pokec social network (PSN) [15]	1.6M	22M
LiveJournal social network (LSN) [6]	4.8M	48M
Orkut social network (OSN) [16]	3.1M	117M

TABLE I: Real world graph datasets.

and S_B in HDRF is:

$$S_B^{HDRF}(p) = \lambda . S_B^{Greedy}(p) \tag{10}$$

In Algorithm 3, we demonstrate HDRF heuristic in HoVerCut. HDRF for end vertices u and v, requires to have information about $d(u)$, $d(v)$, $P(u)$, $P(v)$ and the edge size of all partitions. The implementation of Greedy heuristic is slightly different. It does not need the partial degree of vertices.

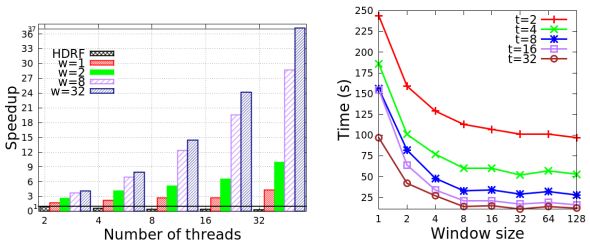
B. Partitioning State

The partitioning heuristics explained in the previous section require the *state* of the processed edges. For example, DBH needs the partial degree of vertices and the Greedy requires the current size of partitions. To provide this, HoVerCut maintains a *shared state* that includes two tables: the *vertex table*, and the *partition table*. The vertex table maps each vertex to its partial degree, as well as the partitions that the vertex is replicated in, and the partition table holds the edge size of each partition.

The shared state provides two interfaces *getState* and *putState*. The getState operation receives a list of vertex identifiers as input and returns the state of all partitions and corresponding vertices. The getState operation does not entail any lock either on the tables or the data entries, which increases the number of parallel access to the shared state. The putState operation receives the local state of a subpartitioner as input and updates the shared state accordingly. We update the entries of the vertex and partition tables with aggregation operations, in which the state of the vertices and partitions are accumulated with deltas. This guarantees that the shared state will be consistent regardless of the order in which it applies the update operations from parallel subpartitioners.

Each subpartitioner has a local copy of the shared state. The more subpartitioners contact the shared storage, the more up-to-date copy they will have, thus, the better partitioning decisions they can make. However, contacting the shared storage is costly and has a negative impact on the processing time. Therefor, subpartitioners should contact the shared storage as less as possible. In other words, there is a trade of between partitioning quality and the partitioning time.

To control the state update interval, we use a configurable *window*. Subpartitioners buffer the incoming edges in a window of certain size (either counter-based or time-based size). When the window is full or timed-out, the subpartitioner pulls the required updates from the shared storage. We assume a *tumbling* window model, in which non-overlapping subset of edges are processed, one at a time.



(a) Speedup. (b) Partitioning time.

Fig. 3: The impact of window size on speedup and partitioning time of HoVerCut with different number of subpartitioners.

IV. Experiments

In this section we demonstrate the performance results of HoVerCut[1] compared to the of state-of-the-art algorithms[2] HDRF [12] and Greedy [11].

A. Experimental Settings

We use two different types of graphs in the experiments: (i) real-world datasets with power-law degree distribution, listed in Table I, and (ii) synthetic graphs, which are generated by Gengraph [17]. In the synthetic graphs, the probability that a vertex has degree d under power-law distribution is $d^{-\alpha}$, where α is a constant. We increase α from 1.4 to 1.7, which ends up with graphs from $195M$ to $33M$ edges, respectively. We set the minimum and maximum degrees of the synthetic graphs to 4 and 30000, and the number of vertices to $1M$.

We measure the following metrics to compare the systems:

1) *Replication Factor (RF)*: the average number of replicated vertices (Equation 3).
2) *Load Relative Standard Deviation (LRSD)*: the relative standard deviation of edge size in each partition. The value zero for LSRD indicates equal size partitions.
3) *Partitioning time*: the required time to partition a graph.
4) *Speedup*: the time it takes for partitioning with multi-thread compared to a single thread.

We denote the number of parallel subpartitioners by t, and the size of the window by w, and we use $H(t = x, w = y)$ to represent a configuration with x subpartitioners and the window size y. We also show the implementation of HDRF and Greedy in HoVerCut by HoVerCut(H) and HoVerCut(G), respectively. All the reported results are the average of three runs. In all the experiments, we set the number of partitions to 16, and the input graphs are streamed by their edges.

B. One Host Configuration

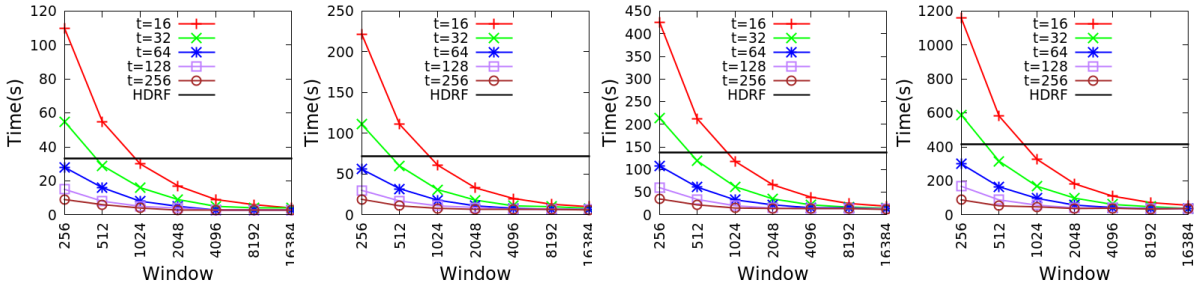
In these experiments, we deployed HoVerCut as a single process in one machine, and each subpartitioner runs as a separate thread in this process. Here, we apply $H(w = 32, t = 32)$ configuration in HoVerCut, and use the real-world datasets, shown in Table I.

Table II represents the result of comparison of the RF and partitioning time of HoVerCut(H) and HoVerCut(G) with the original HDRF and Greedy algorithms. As it shows, HoVerCut

[1]https://github.com/shps/HoVerCut
[2]https://github.com/fabiopetroni/VGP

	Partitioning time [s]				RF			
Dataset	HDRF	HoVerCut (H)	Greedy	HoVerCut (G)	HDRF	HoVerCut (H)	Greedy	HoVerCut (G)
AS	33	1	28	1	1.99	2.00	2.24	2.24
PSN	72	2	66	3	3.90	3.89	3.89	3.89
LSN	138	5	123	5	2.76	2.76	2.83	2.83
OSN	415	11	371	11	5.57	5.54	5.29	5.29

TABLE II: Summary of the partitioning results for HDRF, Greedy, HoVerCut(H) and HoVerCut(G). The configuration of HoVerCut is $H(t = 32, w = 32)$. The LSRD is 0.00% in all the experiments.



(a) AS (b) PSN (c) LSN (d) OSN

Fig. 2: The speedup in HoVerCut with different number of threads for different window sizes.

partitions the AS graph 33 times faster than HDRF and 28 times faster than the Greedy. For a bigger graph like OSN, HoVerCut is 37 times faster than HDRF. Note that this level of speedup is reached without degrading the quality of partitions, i.e., RFs are almost equal and $LRSD = 0.00$ in the final partitions. Due to the lack of space, in the rest of this section we only compare HoVerCut with HDRF partition that shows a better performance compare to the other heuristics.

Figure 3(a) shows the gained speedup of HoVerCut with different number of subpartitioners and window sizes. As we see, for a fixed number of threads (subpartitioner), increasing the window size can effectively improve the speedup, due to reducing the congestion over the shared state. For example, for $t = 32$, increasing w from 1 to 32 makes HoVerCut to run 5 to 37 times faster than HDRF. In contrast to HoVerCut, increasing the number of threads has a negative impact on the performance of HDRF, because HDRF requires to entail locks over the vertex and partition entries in every partitioning step. We remark that, in all of the configurations, RF and LRSD are the same as in HDRF. Figure 3(b) represents how increasing w improves the partitioning time in different number of subpartitioners. However, the partitioning time becomes constant for w greater than 16.

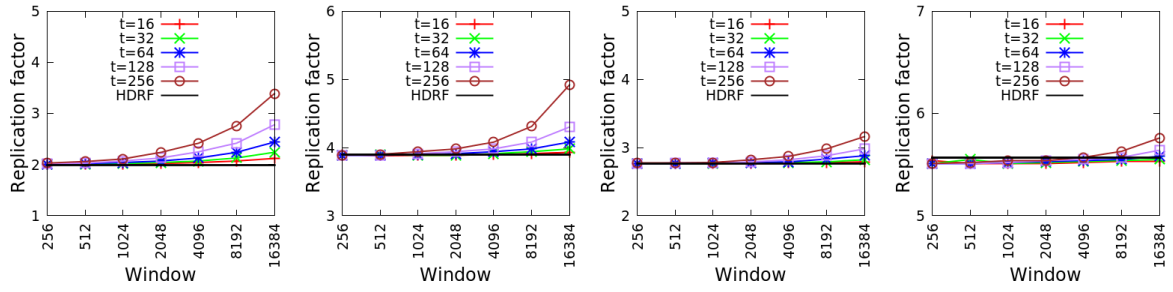
C. Distributed Configuration

To further utilize the available resources in a distributed environment, we implemented each subpartitioner as a separate process on a machine, and we deployed the shared state on a remote storage. However, in this configuration, the partitioning time can dramatically increase due to the network latency. For example, the distributed partitioning of the graph LSN with $H(t = 256, w = 1)$ takes around six hours, while in a single process with $H(t = 2, w = 1)$, the time is around one minute. In this section, we show that HoVerCut can significantly reduce the partitioning time if we use bigger window sizes.

Figure 2 represents the partitioning time of the real-world graphs (Table I). The straight line in these plots is the time of a single thread HDRF. The results show when the number of parallel subpartitioners is more than 64, HoVerCut achieves a better partitioning time with widows bigger than 256. These plots also show that when there are fewer subpartitioners in the system, HoVerCut still can beat HDRF by increasing the window size. This improvement is mainly due to reducing the rate of access to the remote shared state, which entails costly communication over the network.

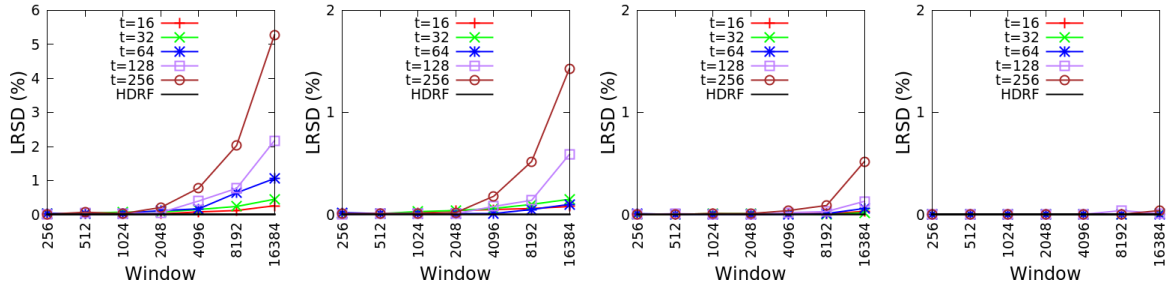
Although increasing w improves the partitioning time, it may increase RF and LRSD. In Figures 4 and 5, we see the RF and LRSD of the same experiments as in Figure 2. These figures demonstrate that in bigger graphs (with respect to their number of edges, $|E_{PSN}| < |E_{LSN}| < |E_{OSN}|$), we can employ bigger windows and more parallel subpartitioners to achieve better partitioning time without degrading the partitioning quality. For example, $H(t = 256, w = 16384)$ for the graph AS, with $11M$ edges, makes partitions with $LRSD = 5\%$, while for the graph OSN with $117M$ edges, it makes partitions with near zero LRSD. These results confirm that we can horizontally scale HoVerCut for larger graphs, without degrading the quality of output partitions.

We also evaluate HoVerCut with a set of synthetic graphs to study the effect of vertices degree distribution on the its performance. Figure 6 shows the results of HoVerCut with 64 subpartitioners. As we see, HoVerCut generates competitive partitions to HDRF for graphs with different values of α. Furthermore, for small values of α, HoVerCut can employ



(a) AS (b) PSN (c) LSN (d) OSN

Fig. 4: RF for different datasets.



(a) AS (b) PSN (c) LSN (d) OSN

Fig. 5: LRSD for different datasets.

bigger windows without significantly degrading the quality of partitions. For example, for $\alpha < 1.55$, the RF and LRSD of partitions in HoVerCut with $w = 16384$ are close to HDRF.

V. Related Work

In this section we study some of the existing work on graph partitioning. Considering edge-cut vs. vertex-cut, as well as offline vs. online (streaming) algorithms, we review four groups of graph partitioning algorithms: offline edge-cut, online edge-cut, offline vertex-cut, and online vertex-cut partitioning. In addition to these, we also study the state-of-the-arts on multi-loader graph partitioning systems.

A. Offline Edge-Cut Partitioning

Kernighan-Lin (KL) [18] is a classic graph partitioning algorithm that minimizes the edge-cut while keeping the cluster sizes balanced. Hierarchical algorithms (agglomerative or divisive) are another approach in edge-cut partitioning, in which in agglomerative algorithms [19], initially all vertices are placed in different partitions of size one, and over time the pair of partitions with the shortest distance are merged into a single partition. The divisive algorithm [20] operates in reverse, such that in the beginning, all the vertices are put in a single partition, and then at each step it chooses a certain partition and split it into two parts.

A common solution in edge-cut partitioning is the multi-level partitioning [2] that consists coarsening, partitioning, and un-coarsening phases. METIS [21], KAFFPA [22], and [23], [24], [25] are examples of multi-level partitioning algorithms.

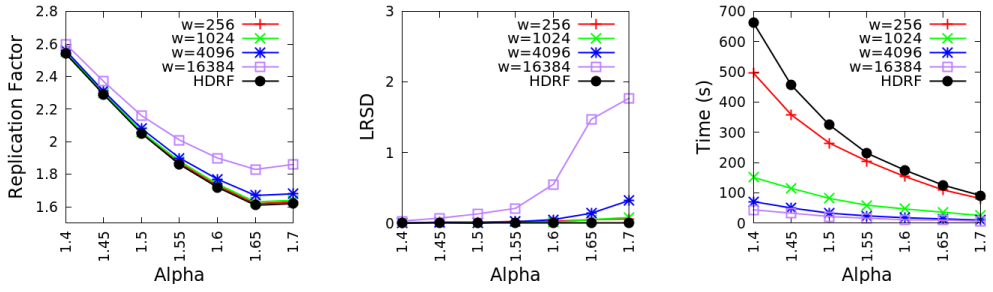
We can also refer to spectral [26] and Markov clustering [27] algorithms as two other partitioning solutions. The former algorithm assign vertices to partitions based on the eigenvectors of matrices, and the latter clusters graphs via manipulation of the transition probability matrix corresponding to the graph.

In addition to the above solutions, which all require access to the entire graph, there exist distributed algorithms for large graphs. Ja-be-Ja [28], [29] is a fully distributed algorithm that uses local search and simulated annealing techniques [30] for graph partitioning. Sheep [31] is a distributed graph partitioner that reduces the graph to an elimination tree, partitions the tree, and then translates the tree partitions into graph partitions. Spinner [32] and [33] are two other large scale graph partitioners that take advantage of the label propagation algorithm for graph partitioning.

B. Online Edge-Cut Partitioning

In online (streaming) edge-cut partitioning algorithms, vertices/edges arrive in sequence and the algorithms assign them to different partitions. Most of the streaming algorithm are one-pass algorithm and they forbid partition refinement after assignments. Stanton et al. [34], [35] studied different heuristics, and observed the best performance with the linear deterministic greedy (LDG) heuristic. The algorithm assigns each vertex to a partition where the vertex has the most edges, and the algorithm weights the assignment by a penalty function based on the size of the partitions. FENNEL [10] is another online edge-cut algorithm with the aim of maximizing the modularity.

A new approach in this group of partitioners is restreaming the data, meaning that the same (or approximately the same) graph repeatedly streamed on a regular basis. Nishimura et al. [36] showed that they achieved a better performance by



(a) RF (b) LRSD (c) Partitioning time

Fig. 6: RF, LRSD and partitioning time with respect to different alphas.

restreaming LDG and FENNEL.

C. Offline Vertex-Cut Partitioning

While there exist numerous solutions for edge-cut partitioning, very little effort has been made for vertex-cut partitioning. Florian et al. in [37] analyzed the balanced vertex-cut as an alternative to balanced edge-cut partitioning by providing explicit characterization of the expected communication cost for different variants of the graph partition problems. SBV-Cut [38] is one of the few algorithms for vertex-cut partitioning. In this algorithm, initially a set of balanced vertices are identified for bisecting a directed graph, and then he graph is further partitioned by a recursive application of structurally-balanced cuts to obtain a hierarchical partitioning of the graph.

Ja-be-Ja-vc [39] is a recent distributed vertex-cut partitioning algorithm, inspired by Ja-be-Ja [28], [29] for edge-cut partitioning. Similar to Ja-be-Ja, this algorithm uses local search and simulated annealing to iteratively improve initial random assignment of edges to partitions. DFEP [40] is another distributed vertex-cut partitioning algorithm that works based on a market model, where the partitions are buyers of vertices with their funding. Firstly, all partitions are given the same amount of funding. Then, in each round, a partition p tries to buy edges that are neighbors of the already taken edges by p, and an edge will be sold to the highest offer. Another algorithm in this area is VSEP [41] that presented a parallel vertex-cut partitioning based on two heuristic methods to compute edge partitioning iteratively.

D. Online Vertex-Cut Partitioning

Existing online vertex-cut partitioning algorithms are grouped into two main categories: hashing algorithms and greedy algorithms. The former group of the algorithms ignore the history of the edge assignments and rely on the presence of a predefined hash function, while the latter group uses the entire history of the edge assignments to make the next decision.

The hashing algorithms can end up with a good load balancing with a uniform hash function. A simple solution based on the hashing technique was presented at [11], where the algorithm assigns each edge randomly to a partition based on given hash function. This solution results in a large number of vertex-cuts. Degree-Based Hashing (DBH) [13] is another hash-based algorithm that considers the degree of the vertices for the placement decision. Grid-based hashing solution is an approach that presented at GraphBuilder [42]. This model arranges partitions in a matrix and maps each vertex to a matrix cell using a hash function. The algorithm allows each vertex to be replicated only in a small subset of partitions.

In addition to the hashing solutions, there are a number of algorithms presented based on the greedy model. One of the early solutions introduced at PowerGraph [11], where edges are evenly assigned to multiple machines. The aim of the algorithm is to make the number of machines spanned by each vertex small, and to reduce the communication overhead and impose a balanced computation load on the machines. Rong et al. proposed an hybrid solution in Ginger [43] that combines both edge-cut and vertex-cut approaches together. Ginger, however, needs extra reassignment phases after the original streaming graph partitioning. A recent work in this area is HDRF [12], where our work is inspired by it. HDFR is a streaming vertex-cut graph partitioning algorithm that exploits skewed degree distributions by explicitly taking into account vertex degree in the placement decision.

E. Multi-Loader Online Graph Partitioning

There is not much work in multi-loader online graph partitioning. To the best of our knowledge, GraSP [44] is the only work on online edge-cut partitioning. Its implementation follows the restreaming partitioning [36] that shows the single-pass algorithms of FENNEL [10] and WDG [35] can be repeated over the same data in the same order, yielding a convergent improvement in quality.

VI. Conclusions

We targeted the scalability problem of vertex-cut partitioning algorithms for streaming power-law graphs. We introduced HoVerCut, a parallel and distributed vertex-cut partitioner that can employ different partitioning policies in a scalable fashion. HoVerCut decouples the partitioning policy from the state, and utilizes an efficient tumbling window model to share state between multiple instances of the partitioning algorithm. We demonstrated that HoVerCut scales both horizontally and vertically to partition large graphs tens of times faster than the state-of-the-art algorithms, without degrading the quality of partitions.

ACKNOWLEDGEMENT

This work was supported by the End-to-End Clouds project funded by the Swedish Foundations for Strategic Research under the contract RIT10-0043 and the BIDAF project funded by KK-stiffen (KKS) number 20140221.

REFERENCES

[1] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[2] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs." *SC*, vol. 95, p. 28, 1995.

[3] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM computer communication review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[4] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 10–pp.

[5] K. Lang, "Finding good nearly balanced cuts in power law graphs," *Preprint*, 2004.

[6] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[7] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," *nature*, vol. 406, no. 6794, pp. 378–382, 2000.

[8] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of OSDI*, 2014, pp. 599–613.

[9] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[10] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 2014, pp. 333–342.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI)*. USENIX, 2012, p. 2.

[12] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. ACM, 2015, pp. 243–252.

[13] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Advances in Neural Information Processing Systems*, 2014, pp. 1673–1681.

[14] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 177–187.

[15] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *International Scientific Conference and International Workshop Present Day Trends of Innovations*, 2012, pp. 1–6.

[16] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[17] F. Viger and M. Latapy, "Efficient and simple generation of random simple connected graphs with prescribed degree sequence," *Journal of Complex Networks*, p. cnv013, 2015.

[18] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[19] M. E. Newman, "Fast algorithm for detecting community structure in networks," *Physical review E*, vol. 69, no. 6, p. 066133, 2004.

[20] M. E. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical review E*, vol. 69, no. 2, p. 026113, 2004.

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[22] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *Algorithms–ESA 2011*. Springer, 2011, pp. 469–480.

[23] U. Benlic and J.-K. Hao, "An effective multilevel tabu search approach for balanced graph partitioning," *Computers & Operations Research*, vol. 38, no. 7, pp. 1066–1075, 2011.

[24] P. Chardaire, M. Barake, and G. P. McKeown, "A probe-based heuristic for graph partitioning," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1707–1720, 2007.

[25] A. J. Soper, C. Walshaw, and M. Cross, "A combined evolutionary search and multilevel optimisation approach to graph-partitioning," *Journal of Global Optimization*, vol. 29, no. 2, pp. 225–241, 2004.

[26] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[27] V. Satuluri and S. Parthasarathy, "Scalable graph clustering using stochastic flows: applications to community discovery," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 737–746.

[28] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "Ja-be-Ja: A distributed algorithm for balanced graph partitioning," in *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 51–60.

[29] ——, "A distributed algorithm for large-scale graph partitioning," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 2, pp. 1–24, June 2015.

[30] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.

[31] D. Margo and M. Seltzer, "A scalable distributed graph partitioner," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1478–1489, 2015.

[32] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," *arXiv preprint arXiv:1404.3861*, 2014.

[33] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *arXiv preprint arXiv:1404.4797*, 2014.

[34] I. Stanton, "Streaming balanced graph partitioning algorithms for random graphs," in *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2014, pp. 1287–1301.

[35] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 1222–1230.

[36] J. Nishimura and J. Ugander, "Restreaming graph partitioning: simple versatile algorithms for advanced balancing," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1106–1114.

[37] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 1456–1465.

[38] M. Kim and K. S. Candan, "Sbv-cut: Vertex-cut based graph partitioning using structural balance vertices," *Data & Knowledge Engineering*, vol. 72, pp. 285–303, 2012.

[39] F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi, "Distributed vertex-cut partitioning," in *Distributed Applications and Interoperable Systems*. Springer, 2014, pp. 186–200.

[40] A. Guerrieri and A. Montresor, "Distributed edge partitioning for graph processing," *arXiv preprint arXiv:1403.6270*, 2014.

[41] Y. Zhang, Y. Liu, J. Yu, P. Liu, and L. Guo, "Vsep: A distributed algorithm for graph edge partitioning," in *Algorithms and Architectures for Parallel Processing*. Springer, 2015, pp. 71–84.

[42] N. Jain, G. Liao, and T. L. Willke, "Graphbuilder: scalable graph etl framework," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 4.

[43] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 1.

[44] C. Battaglino, P. Pienta, and R. Vuduc, "Grasp: distributed streaming graph partitioning," in *1st High Performance Graph Mining workshop, Sydney, 10 August 2015*, 2015.