Lightweight Streaming Graph Partitioning by Fully Utilizing Knowledge from Local View

Zhigang Wang[§†], Zichao Yang[§†], Ning Wang[§*], Yujie Du[¶], Jie Nie[§], Zhiqiang Wei[§], Yu Gu[‡], Ge Yu[‡]
[§]Ocean University of China {wangzhigang, yangzichao, wangning8687, niejie, weizhiqiang}@ouc.edu.cn
[¶]Yantai Eng.&Tech. College {duyujie}@ytetc.edu.cn
[‡]Northeastern University {guyu, yuge}@mail.neu.edu.cn

2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS) | 979-8-3503-3986-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDCS57875.2023.00079

***Abstract*—Data partitioning is the most fundamental procedure before parallelizing complex analysis on very big graphs. As a classical NP-complete problem, graph partitioning usually employs offline or online/streaming heuristics to find approximately optimal solutions. However, they are either heavyweight in space and time overheads or suboptimal in quality measured by workload balance and the number of cutting edges across partitions, both of which cannot scale well with the ever-growing demands of quickly analyzing big graphs. This paper thereby proposes a new vertex partitioner for better scalability. It preserves the lightweight advantage of existing streaming heuristics, and more importantly, fully utilizes the knowledge embedded in the local view when streaming a vertex, which significantly improves the quality. We present a sliding window technique to compensate for the additional memory costs caused by knowledge utilization. Also, a parallel technique with dependency detection optimization is designed to further enhance efficiency. Experiments on a spread of real-world datasets validate that our proposals can achieve overall success in terms of partitioning quality, memory consumption, and runtime efficiency.**

***Index Terms*—Streaming Graph Partitioning, Local Streaming View, Memory Consumption, Parallel Partitioning**

I. Introduction

As one of the most commonly used data structures, graph can reasonably abstract entities and their relationships in the real world. In the big data era, the scale of graph data has grown dramatically and the associated analysis is becoming more and more complex. The traditional centralized solution clearly cannot handle such data- and compute-intensive applications. Now the preferred underlying replacement is to parallelize graph processing [1]–[3]. However, before that, a key procedure is to partition the large-scale input graph into multiple subgraphs/partitions.

Nowadays, a prominent graph partitioning solution must satisfy multi-constraints for the efficient subsequent processing. Typically, graph analysis needs to traverse the topology along edges to exchange intermediate data and then update vertex values. In parallel or distributed environments, edges might be cut across partitions, and then such data as messages are delivered, yielding additional and expensive communication costs. Thus, the first focus is decreasing the number of cutting edges. Another constraint is to balance the distribution of workload measured by the number of vertices and/or edges, so as to avoid possible waiting costs. Besides, now for most distributed graph processing systems like *Pregel* [1], the partitioner as a built-in component is run together with the subsequent processing in each analysis job. The partitioning efficiency significantly affects the overall performance, especially when real graphs are frequently updated and/or shared by multi-tenants with different analysis goals. However, graph partitioning is NP-Complete due to the goals of cutting-edge reduction and workload balance (*Quality*) [4]. The new *Efficiency* constraint further exacerbates the challenge of seeking a good solution.

There have been many studies on graph partitioning in recent years [5], among which there exist two main research lines: offline partitioning and streaming/online partitioning. The former utilizes full knowledge of the input graph by multilevel coarsening and refinement like *METIS* [6] or iterative label propagation like *MLP* [7]. They refine partitioning results again and again to achieve prominent quality, but also have significantly expensive time latency and/or memory consumption. The latter, represented by *LDG* and *FENNEL* [8], [9], are naturally lightweight in efficiency and memory footprints, since data are scanned only once and just a local view (including the currently streamed graph record and the distribution of already streamed and placed vertices) is stored. However, now the knowledge extracted from the local view is very limited, which heavily impairs quality.

Hence, a naturally desirable goal for graph partitioning is to pursue a solution with prominent quality and efficiency. This paper pursues such a target based on streaming heuristics by fully utilizing knowledge embedded in the local view, including neighbor distribution and the topology locality of the input graph. We are aware that recently some pioneers have attempted to make a compromise on top of offline and streaming solutions [10], [11]. However, this paper will show that there still exist huge improvement spaces for the pure streaming methods, without any compromise in its lightweight feature. And our proposal actually can also work as the replacement for the streaming component in their hybrid frameworks. While, others make efforts to reduce the runtime cost of offline methods [7], [12], but the resulting variants still cannot work as efficiently as streaming competitors, due to the built-in multilevel or iterative operations. In particular, this

This work is supported in part by the National Natural Science Foundation of China under Grants U22A2068 and 62072083, in part by the Fundamental Research Funds for the Central Universities under Grant 202042008, and in part by the National Key Research and Development Program of China under Grant 2021YFF0704000.

[†]Co-first authors

[*]Corresponding author

2575-8411/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDCS57875.2023.00079

paper focuses on vertex partitioning, since many distributed graph processing systems provide vertex-centric *APIs* [13].

We first challenge the conventional wisdom that streaming methods like *LDG*, can only extract out-neighbor-based knowledge from the local view. Intuitively, given a newly arrived vertex v in the streaming, we should assign it to the partition which closely connects to v. In *LDG*, the closeness is scored by analyzing the distribution of only v's out-neighbors among partitions. The idea behind it is to localize the delivery of messages sent along outgoing edges as many as possible. However, v as a target also receives messages from its source vertices, which evenly contributes to communication costs. Such important knowledge is embedded in in-neighbors. Inspired by this, we compute scores by in- and out-neighbors together to obtain more knowledge. Note that the most commonly used adjacency list representation only contains out-neighbors. We thereby transfer the in-neighbor analysis into the expectation estimation for already placed vertices (as sources) in the current partitions, which avoids expensive preprocessing about adding in-neighbors. Our another observation is that in the initial phase of the streaming partitioning, the number of already placed vertices is small. The extracted knowledge is still very limited, even though in- and out-neighbors are both involved. We thereby utilize the topology locality of the input graph to quickly make an assumption about partitioning results. The logically assigned vertices help to enhance knowledge.

In-neighbor-based expectation estimation can enhance knowledge and then improve partitioning quality, but it essentially requires each partition P_i to individually count how much P_i expects every vertex v to be assigned to P_i. That yields additional $O(K|V|)$ memory consumption when partitioning a graph with $|V|$ vertices into K parts. The heavy memory footprints make it difficult to scale to large graphs, especially for a big K setting. However, the streaming nature provides an opportunity for optimization. Since already arrived and placed vertices before the current v cannot be moved again, it is not necessary to count the expectation associated with them. Motivated by this, we separate vertices into X shards and smoothly slide the counting window over them in a fine-grained manner, along with the arrival of streaming data. For each partition, only $\frac{|V|}{X}$ spaces are essentially required and can be reused when the focused window slides. The additional space complexity is thereby reduced to $O(\frac{K|V|}{X})$.

Although the streaming solution has a runtime advantage compared with offline, we parallelize it to further enhance efficiency. Our optimization is in shared memory, so that streaming heuristics can be accurately and timely maintained in a centralized manner, to provide a reasonable guide for vertex placement. We particularly parallelize the placement decisions of multiple graph records/adjacency lists, to resolve the compute bottleneck. We especially study how to quickly detect the dependency among parallelized streamed vertices to reduce the possible conflicts when placing vertices.

The major contributions are summarized below.

- Proposing a new *S*treaming *P*artitioner based on in&out-*N*eighbors and topology *L*ocality (*SPNL*), which enhances knowledge extracted from local view and then improves partitioning quality.
- Proposing lightweight optimizations through the fine-grained sliding window and dependency-reduced parallel techniques, which reduces memory consumption and improves runtime efficiency.
- Performing extensive experimental studies where *SPNL* yields 92% quality improvement compared with streaming partitioners *LDG*/*FENNEL*, and 40% even for the underlying offline *METIS*; and runs 2X and 15X faster respectively than *LDG* and the up-to-date parallel offline partitioner *XtraPuLP*.

The remainder of this paper is organized as follows. Sec. II gives the definition of the graph partitioning problem. Sec. III overviews related works. Sec. IV presents the detailed design of our SPNL. Sec. V introduces optimizations of memory consumption and runtime efficiency. Sec. VI shows the evaluation results. Finally, Sec. VII concludes this paper.

II. Preliminaries

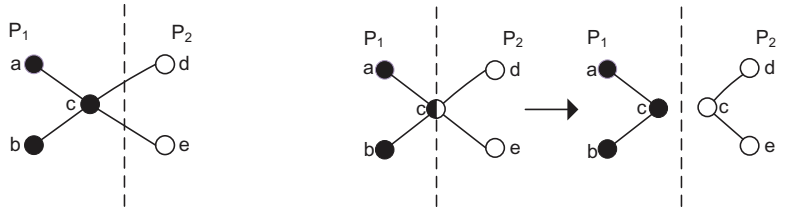
Given a directed graph G=(V, E), V is the set of vertices and E is the set of directed edges. We assume that vertices are consecutively numbered to simplify some design in our proposals. This is reasonable because all of the publicly available graphs we encountered have done the numbering work. Let $|V|$ and $|E|$ denote the numbers of vertices and edges, respectively. Given a directed edge $(v, u) \in E$ with two end-points v and u, v as the source links to u as the target. The in-neighbors of v are a set of vertices linking to it by edges, and the out-neighbors are a set of vertices that v has an edge to link, denoted by $N^{in}(v)$ and $N^{out}(v)$, respectively.

The objective of graph partitioning is to evenly divide G into K disjoint subgraphs/partitions $P_i = (V_i, E_i)$ with reduced connections among them as much as possible, where $i \in \{1, 2, \ldots, K\}$, and $|V_i|$ and $|E_i|$ respectively denote the numbers of vertices and edges in P_i. For any $i \neq j$, we have $P_i \cap P_j = \emptyset$ and $P_1 \cup P_2 \cup \ldots \cup P_K = G$. The evenness is measured by workload balance, which can be further categorized into vertex- and edge-based branches. Eqs. (1) and (2) mathematically show the two respective metrics. Clearly, $\delta_v = 1.0$ ($\delta_e = 1.0$) means that vertices (edges) are evenly distributed across partitions.

$$\max_{i \in [1,K]} \{|V_i|\} \leq \delta_v \frac{|V|}{K} \tag{1}$$

$$\max_{i \in [1,K]} \{|E_i|\} \leq \delta_e \frac{|E|}{K} \tag{2}$$

The specific partitioning logic can also be divided into vertex- and edge-based variants. The former assigns vertices together with their adjacent edges across partitions where an edge (v, u) might be cut if v and u are placed in two different partitions. Now the connection between any two partitions is measured by the number of cutting edges, which indicates the communication costs in subsequent graph processing jobs. The



(a) Vertex partitioning (b) Edge partitioning

Fig. 1. Two types of graph partitioning

latter focuses on assigning edges but an end-point vertex will be replicated in multi partitions if its incoming or outgoing edges are distributed across different partitions. Here the connection metric is the total number of replicators, which dominates the communication cost. Figure 1 gives the two types of graph partitioning.

Generally, a myriad of graph algorithms are programmed in a vertex-centric manner, with easily used *APIs* provided by existing distributed systems [13] like the early pioneer *Pregel* [1]. We then focus on the vertex partitioning. Also, these systems typically plug the partitioner as a necessary preprocessing step for each job. That means a graph will be partitioned multiple times, especially when end-users have very different analysis goals (like running *PageRank* and *Shortest Path* computations in two jobs but on the same graph). The partitioning time is a portion of the total runtime and should be reduced if possible.

Overall, this paper focuses on scalable vertex-based graph partitioning with high quality (reducing the number of cutting edges and making δ_v, or δ_e close to 1.0) and efficiency. Secs IV and V give our new streaming solutions. In particular, Table I summarizes important symbols used in this paper.

TABLE I
DEFINITIONS OF IMPORTANT SYMBOLS

Svmbol	Definition
P_i	The i-th partition of the total K partitions
$\|V\|, \|E\|$	The numbers of vertices and edges
$\|D\|$	The numbers of cutting edges across partitions
$N^{in}(v)$	A set of vertices linking to v by edges
$N^{out}(v)$	A set of vertices that v has an edge to link
δ_v, δ_e	The load balancing factors in terms of vertices and edges
$w^t(i, v)$	The real-time remaining workload capacity of P_i when partitioning vertex v at time t
λ	The weight used to balance the importance of in-neighbors and out-neighbors
$\Gamma_i^t(v)$	A metric measuring how much P_i expects v to be assigned into its vertex set V_i at time t
V_i^{pt}/V_i^{lt}	The set of vertices physically/logically allocated into P_i at time *pt/lt*
X	The number of shards for each partition used in the sliding window technique
RCT	A hash-based Reversed-Counting-Table for vertex v to detect the dependency associated with $u \in N^{out}(v)$
ECR	The Edge Cut Ratio evaluated by $\frac{\|D\|}{\|E\|}$
PT	Time spent on partitioning
MC	Memory consumption during partitioning

III. Related Works

This section begins with an introduction to two important research branches: offline and streaming, and then gives a general overview of parallel optimization techniques.

A. Offline graph partitioning

There is a rich literature on offline graph partitioning where the input graph is scanned multiple times to gradually refine the quality. The most well-known representative is to first progressively coarsen the input graph using maximum matching clustering schemes, and then partition the coarsest graph [14] and project back the division to the original graph. *METIS* [6] and *Scotch* [15] belong to this category. *MLP* [7] replaces the costly maximum matching with label propagation-based connected component computation. Some methods further rely entirely on iterative label propagation for parallel partitioning [16], [17]. Slota et al. extend this idea with multiple constraints and multiple objectives [18]. Besides, Deng et al. explore its advantage when partitioning heterogeneous graphs [19]. These offline methods can scale to moderate graphs with relatively good quality. Among them, *METIS* is usually regarded as the quality benchmark. However, they are runtime inefficient and memory consuming, due to multiple data scans and the storage of immediate results.

B. Streaming/online graph partitioning

Some researchers consider the arrival of graph data to be streaming. As vertices arrive, we can determine the location of new arrivals by computing the distribution of the already placed data. This one-pass data-scanning design enables it to scale to very large-scale graphs, although the quality is reduced because of the lack of a global view. *LDG* [8] and *FENNEL* [9] are two representatives. The quality can be further improved by providing the entire or partial output of a previous execution to the current one, termed as fully or partially re-streaming [20], [21]. Recent works extends the streaming design to edge partitioning, especially for power-law graphs [22]. Compared with vertex partitioning where a whole adjacency list is available, now the knowledge learned from an edge is very limited. Researchers thereby focus on quality improvement by multi-streaming [23], [24], buffering edges [25], and establishing a hybrid solution based on offline and online schemes [26]–[29]. Other researchers also explore the impact of a hybrid design for vertex partitioning [10],

[11]. Clearly, our proposal can work as a plugged underlying streaming component in these works.

C. Parallel graph partitioning

With the increasing size of graph data, it is inevitable to implement partitioning in parallel or distributed environments.

Offline: The most common approach is to parallelize existing partitioners, for example, *ParMETIS* [30] of *METIS* and *PT-Scotch* [31] of *Scotch*. Holtgrewe et al. and Akhremtsev et al. also devote efforts into parallelizing such multi-level solutions [32], [33]. Nevertheless, the efficiency improvement is still limited. *Spinner* [34] and *JA-BE-JA* [17] thereby solely use iterative label propagation for efficiency. *XtraPuLP* [12] also follows this idea and parallelizes it in shared and distributed memory settings. It can easily scale to over 8K machines to partition a trillion-edges graph in minutes. However, the excessive hardware resource requirement still makes it cost-inefficient, due to iterative label propagation.

Online/streaming: Previously, Shi et al. parallelize the placement decisions on a distributed memory platform and devote efforts into reducing network latency [35]. While, our parallel optimization works in shared memory without this shortcoming, since currently a regular multi-cores service has strong enough computing power and storage capacity. Recently, Hua et al. split the streaming into massive parts for independently parallel edge partitioning [36] where each requires inefficient multiple rounds of computations to break the dependency. Faraj et al. and Wang et al. also focus on efficiency in terms of parallelism [37] and topology-locality utilization [38]. However, as reported, their partitioning quality generally heavily degrades, compared with centralized *LDG*/*FENNEL*.

IV. SPNL: Streaming graph partitioning with enhanced heuristics

This section first introduces the existing linear deterministic greedy heuristic used in *LDG* as a basis, and then proposes our enhanced heuristics by considering in-out neighbors and topology locality.

A. The basic linear deterministic greedy heuristic

Although offline partitioners can output high-quality partitions, they have a significant increase in runtime costs and memory consumption for large-scale graphs. For better scalability, streaming partitioners have received extensive attention. The classical representative *LDG* uses a Linear Deterministic Greedy heuristic, which tries to place adjacent vertices in the same partition to reduce the number of cutting edges, while satisfying the capacity constraint of each partition for workload balance. The heuristic function of *LDG* is given in Eq. 3 to select a reasonable partition id *pid* for the currently arrived vertex v. Here $\left|V_i^{pt} \cap N^{\text{out}}(v)\right|$ indicates till now, at the time instance t, how many out-neighbors of v have already arrived and placed into the i-th partition P_i. $w^t(i,v)$ is the real-time remaining workload capacity of P_i, which works as a penalty function to balance the workload. We can use $|V_i|$ or $|E_i|$ to support vertex-based or edge-based balance.

$$pid = \arg\max_{i\in[1,K]} \left\{\left|V_i^{pt} \cap N^{\text{out}}(v)\right| \cdot w^t(i,v)\right\} \qquad (3)$$

Fig. 2 demonstrates how Eq. 3 works in *LDG* with $K = 3$. We assumed that vertices numbered from 1 to 6 have been serially streamed and placed by $V_1 = \{3, 5\}$, $V_2 = \{1, 2\}$, and $V_3 = \{4, 6\}$. For the currently arrived vertex 7 with $N(7) = \{6, 9, 10\}$, based on Eq. 3, the distribution score associated with P_1, P_2 and P_3 is $(0, 0, 1)$. Since now all partitions have the same remaining capacity about vertices, vertex 7 is finally assigned to P_3 and cannot be moved again.

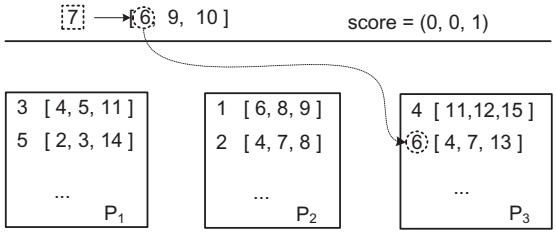


Fig. 2. Illustration of LDG

B. A heuristic with runtime in-out neighbors

The streaming *LDG* has a significant improvement in efficiency since data is scanned only once. However, it cannot reduce the number of cutting edges very well, and hence affects the quality. The reasons are twofolds. The first is that if the out-neighbors of the currently arrived vertex v have been roughly evenly placed across partitions, the score is then mainly dominated by the capacity. In another word, we assign v without any optimization about cutting edge reduction. The second is that at the initial streaming stage, all partitions are roughly empty. The nearly equal score values enforce *LDG* to blindly randomly assign v.

In fact, the two reasons are caused by the limited knowledge extracted from the *Local View* consisting of the currently arrived vertex and the distribution of already placed vertices. Utilizing only out-neighbors increases the probability of ignoring cutting edge optimization. That motivates us to enrich knowledge extracted from the local view. Here, we propose the new Streaming Partitioner by additionally considering in-Neighbors (*SPN*). The idea behind it is that a vertex v not only sends messages along outgoing edges to its targets, but also receives messages along incoming edges from in-neighbors. Assigning v to the partition, to which a maximal number of its out-neighbors have been placed, of course can reduce the number of cutting edges (outgoing edges) linking from v. However, assigning it to the partition with the maximal number of its in-neighbors, can also achieve the goal of cutting edge reduction but now they are incoming edges from v (or the outgoing edges from these in-neighbors). Clearly, the latter can equivalently optimize communication costs, as demonstrated in the top in Figure 3. Thus, taking both in- and out-neighbors into account can decrease the probability of encountering the two scenarios/reasons explained above.

In this way, the partitioner will make the placement decision more smartly. Eq. 4 mathematically shows this new design, where the parameter λ is used to balance the importance of in-neighbors and out-neighbors. Users can specify it manually.

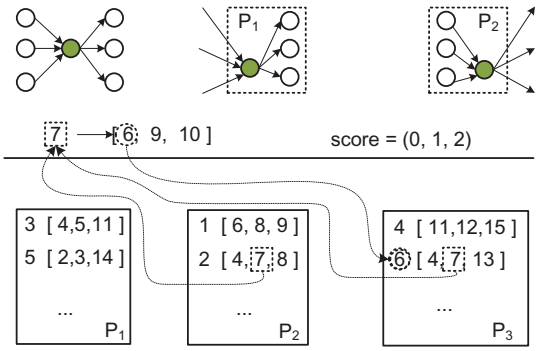


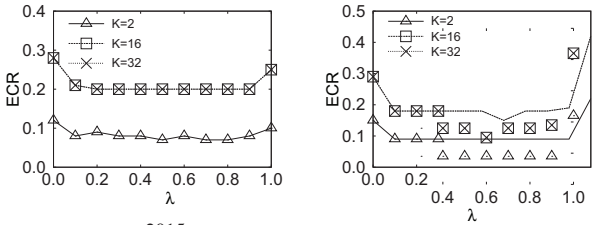
Fig. 3. Illustration of *SPN*

$$pid = \arg \max_{\mathrm{i}\in[1,\ \mathrm{K}]} \Big\{ \big(\lambda |V_i^{pt} \cap N^{\mathrm{out}}(v)| + (1-\lambda)|V_i^{pt} \cap N^{\mathrm{in}}(v)|\big) \cdot w^t(i,v) \Big\} \tag{4}$$

Note that for a directed graph, in-neighbors are not available in the adjacency list. However, we can partially infer this information from out-neighbors of already placed vertices in the local view. At the time instance t, we now use a lookup table Γ_i^t with the size of $|V|$, to count how many times an arbitrary vertex $x \in V$ exists in the out-neighbors of vertices already maintained by P_i. Assume that now v is arriving and u has already been placed into P_i. The counting operation can be performed by traversing out-neighbors of u when placing it into P_i. If u is one of the in-neighbors of v, then $(u,v) \in E$ and the counter associated with v increases by 1. $\Gamma_i^t(v)$ implies that how much P_i expects v to be assigned into its V_i, and if yes, then the message from u to v can be quickly delivered in local. In this way, we can estimate $|V_i^{pt} \cap N^{\mathrm{in}}(v)|$ in Eq. 4 by traversing the out-neighbors of v's out-neighbors and then summing up the corresponding counts. We call this as runtime in-out neighbor estimation since we do not directly know v's in-neighbors. Eq. 4 is thereby re-written as Eq. 5.

$$pid = \arg \max_{\mathrm{i}\in[1,\ \mathrm{K}]} \Big\{ \big(\lambda |V_i^{pt} \cap N^{\mathrm{out}}(v)| + (1-\lambda) \sum_{u \in N^{\mathrm{out}}(v)} \Gamma_i^t(u)\big) \cdot w^t(i,v) \Big\} \tag{5}$$

Note that *SPN* degrades to *LDG* when $\lambda = 1$, because the impact of in-neighbors is completely ignored. On the other hand, in-neighbors dominate the cutting edge reduction if $\lambda = 0$. To seek an optimal setting, we run *SPN* by manually varying λ on many real graphs like *eu2015* and *indo2004* (described in Table II in Sec. VI-A). Let *ECR* be the ratio of the number of cutting edges to $|E|$. Our goal is to reduce this metric.



(a) eu2015 (b) indo2004

Fig. 4. Evaluating the impact of λ

As shown in Figure 4, the two extreme settings both yield a suboptimal result. That makes sense because solely utilizing in-neighbors or out-neighbors will definitely encounter the two scenarios explained at the beginning of this subsection. We thereby use $\lambda = 0.5$ by default to equally weight contributions from the two kinds of neighbors.

Figure 3 demonstrates how *SPN* works using the same example in Figure 2. By Eq. 5, it is known that the out-neighbors based score is (0,0,1) and the in-neighbors based expectation estimation score is (0,1,1). Vertex 7 is finally assigned to P_3 since the combined score is (0,1,2)[1].

Algorithm 1 gives the procedures for processing a newly arrived vertex v with *SPN*. After deciding the placement, we need to update V_{pid}^{pt}, E_{pid}^{pt}, w_{pid}^t, and Γ_{pid}^t associated with the selected partition. In Line 4, C is the capacity constraint of each partition, which is computed by $\delta \cdot \frac{|G|}{K}$ with the user-given balance threshold. $|G|$ and $|P_i^t|$ are respectively equal to $|V|$ and $|V_i^{pt}|$ for vertex-based balance, and $|E|$ and $|E_i^{pt}|$ for edge-based balance. w^t clearly indicates the remaining capacity.

Algorithm 1 Assigning a vertex v using *SPN*

Input: v, $N^{out}(v)$, K, $\mathbb{V}^{pt} = \{V_1^{pt}, V_2^{pt}, ..., V_K^{pt}\}$, $\Gamma^t = \{\Gamma_1^t, \Gamma_2^t, ..., \Gamma_K^t\}$
Output: the partition id *pid*

1: Compute *pid* by Eq. 5
2: $V_{\mathrm{pid}}^{pt} = V_{\mathrm{pid}}^{pt} \cup \{v\}$
3: $E_{\mathrm{pid}}^{pt} = E_{\mathrm{pid}}^{pt} \cup N^{\mathrm{out}}(v)$
4: Update $w_{\mathrm{pid}}^t = 1 - \frac{|P_i^t|}{C}$
5: **for** $u \in N^{\mathrm{out}}(v)$ **do**
6: Increase $\Gamma_{\mathrm{pid}}(u)^t$ by 1
7: **end for**
8: **return** *pid*

C. A heuristic utilizing topology locality

Adding the runtime in-neighbor expectation indeed can enrich knowledge, however, at the initial streaming stage, the effectiveness is still marginal because now few vertices are placed. This is because *SPN* (also *LDG*) only cares about the physically assigned neighboring vertices and ignores the temporarily unassigned ones. As shown in Figure 3, for vertex

[1]For better understanding, here we remove the weight parameter λ.

7, only the neighboring vertex 6 is assigned and counted, and others like 9 and 10 cannot provide any useful heuristic information. Otherwise, their assignment as prior-known knowledge can further boost the accuracy of assigning 7.

In this section, we design a two-step variant of *SPN* with an additional but very lightweight logical pre-assignment for all vertices. Then at the subsequently physically assignment step, such pre-assignments V_i^{lt} as prior-knowledge can be used, as shown in Eq. 6. Here we do not directly use $V_i^{pt} \cup V_i^{lt}$ to compute the size of the intersection set with $N^{\text{out}}(v)$, because the logical assignment might not be accurate. Instead, we separately compute the two intersection sets and weight them with another parameter η, resembling λ. Different from λ with a fixed value, η_i^t as a decay factor gradually decreases with the elapsed time when more neighboring vertices have already been physically placed, since we believe the latter a little bit more. This paper typically sets $\eta_i^t = \max\{0, \frac{|V_i^{lt}|-|V_i^{pt}|}{|V_i^{lt}|}\}$, and more interesting yet effective settings will be explored as future work. Note that V_i^{lt} also dynamically changes. $v \in V_i^{lt}$ will be immediately removed once it is physically assigned.

$$pid = \arg\max_{\mathrm{i}\in[1,\ \mathrm{K}]} \left\{ w^t(i,v) \cdot \Big(\big(1-\lambda\big) \sum_{u \in N^{\text{out}}(v)} \Gamma_i^t(u) + \lambda\big((1-\eta_i^t)|(V_i^{pt}) \cap N^{\text{out}}(v)| + \eta_i^t |(V_i^{lt}) \cap N^{\text{out}}(v)|\big) \Big) \right\} \tag{6}$$

Last but not least, we need to carefully select the logical pre-assignment policy. It should (1) be lightweight with negligible memory consumption and runtime latency for better scalability; and (2) capture the topology locality for better accuracy, i.e., $v \in V_i^{lt}$ will be indeed physically added into V_i in high probability. This paper employs a *Range* method where vertices are consecutively distributed among partitions. This can be efficiently done by constructing a lookup table, where each partition only records the minimal and the maximal vertex ids since all vertices are numbered. The specific range size is decided by the balance constraint. We evenly range-partition vertices if δ_v is primarily concerned; or split the input graph file to infer the boundary ids if δ_e is used, since edges dominate the storage size. Such a table can be quickly formed with $O(2K)$ space complexity, satisfying the first constraint. For the second, in fact our previous work has already validated that a large amount of graphs are crawled by *BFS*, which naturally embeds the topology locality into the vertex storage order in the disk file[2] and can be preserved by the Range policy. We thereby call the variant of *SPN* with enhanced topology Locality as *SPNL*.

Besides locality originally provided by input, *SPNL* can also enhance the accuracy of logical pre-assignment by itself. By Eq. 6, we know a vertex v will tend to be physically assigned into P_i, if most of its out-neighboring vertices like u are logically assumed to be in V_i^{lt}. That in turn increases the expectation counter $\Gamma_i(u)$ if v indeed belongs to V_i. Thus, when many vertices are attracted by the assumed u, the probability of placing u into V_i will inevitably increase.

Figure 5 finally demonstrates *SPNL*. Using the vertex-based balance constraint, the total 15 vertices are evenly logically pre-assigned across 3 partitions. When vertex 7 arrives, we compute the score contributed by not only already physically placed in-neighbors 2 and 6, and out-neighbor 6, but also logically assigned out-neighbors 9 and 10. The final score is then (0,3,2)[3] and hence 7 is placed in V_2. After that, 7 is removed from V_2^{lt} since it has been physically assigned.

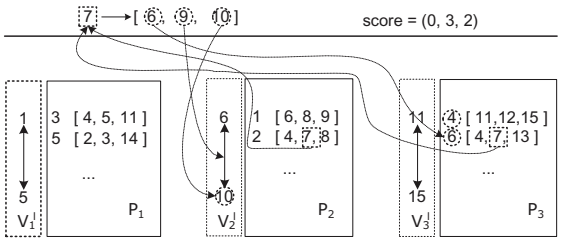


Fig. 5. Illustration of *SPNL*

Compared with *SPN* in Algorithm 1, the only difference is that *SPNL* requires a logical lookup table and dynamically updates it by removing already physically placed vertices.

V. Lightweight Streaming Optimizations

This section introduces our optimizations on memory consumption and parallel efficiency. They make *SPN* and *SPNL* work in a lightweight and scalable manner.

A. Optimizing memory footprints with sliding window

Compared with *LDG*, our *SPN* and *SPNL* can output partitions with a lower number of cutting edges, with enhanced knowledge from in-out neighbors and topology locality, but at expense of consuming more memory resources. In *LDG*, we only need to allocate memory resources for the local view. Let $\max d$ stand for the maximal out-degree in G. The local view includes an adjacency list with the length of $\max d$ at most, a K-dimensional score vector, and the partitioning route table that records the placements of all $|V|$ vertices. Its space complexity is then $O(|V| + K + \max d)$. For *SPN*, it requires to count the expectation on all vertices for each partition. The total memory consumption then significantly increases to $O((K+1)|V| + K + \max d)$. For *SPNL*, the additional cost is caused by the logically pre-assigned route table, yielding the total $O((K+1)|V| + 3K + \max d)$ complexity. Clearly, expectation estimation incurs the most significant memory consumption for our proposals, which poses great scalability challenges for very large-scale graphs. This subsection thereby focuses on optimizing the additional $O(K|V|)$ memory costs.

Recall that the already placed vertices will not be moved since streaming partitioning scans data only once. Then at the time instance t, we actually do not need to count the

[2]Please refer to Table 1 in Ref. [38].

[3]For better understanding, here we remove weight parameters λ and η.

expectation for vertices arrived before t, since these counts will not be used forever. That clearly saves the compute and storage resources. By this straightforward optimization, for each partition P_i, we evenly divide all vertices in Γ_i into X shards, denoted by S_{i1}, S_{i2}, ..., S_{iX}. Since vertices are consecutively numbered and serially streamed, we can safely abandon a shard S_{ij} if and only if all vertices within it have been physically assigned. The total memory consumption thereby gradually decreases. However, at the very beginning, all X shards should be preserved and the peak consumption is still $O(K|V|)$.

Another observation is that benefitting from the topology locality, the currently arrived vertex usually has edges linking to neighbors numbered roughly around it. This motivates us to further slide a focus window from one shard to another along with the streaming operation, and only vertices in the window are counted as the associated expectation. Then different shards can share the memory for counters, to reduce the peak consumption to $O(\frac{K|V|}{X})$.

X is a key parameter in the sliding window technique. A large X of course can significantly reduce the memory cost, but some useful counts might be missed. Suppose the currently arrived vertex v falls into S_{xj} for the j-th shard of all $x \in [1, K]$ partitions. The distribution of its out-neighbors $u \in N^{\text{out}}(v)$ can be categorized into three cases: (1) u rightly falls into the same shard and then can be accurately counted about its expectation; (2) the belonging shard has been slid where the counting loss can be ignored since vertices in that shard have already been placed; (3) u belongs to a shard which will be focused on in future, then the loss brings a negative impact on placing u. In Sec.VI-B, we empirically give a recommendation about setting the value of X and validate that the recommended value can strike a good balance between memory reduction and partitioning quality.

The third case also tells us that a coarse-grained shard-by-shard implementation leads to a sharp sliding, which incurs huge accuracy loss on expectation, especially for boundary vertices. We thereby implement the sliding window technique in a fine-grained manner. The sliding unit is a vertex, rather than a shard. When a new vertex arrives, we remove the oldest vertex in the window and extend the boundary to the next vertex. Since vertices are consecutively numbered and serially streamed, this can be logically implemented by rotating over a fixed-size array by carefully computing the location index. The fine-grained sliding operation can smoothly embrace the expectation estimation of boundary vertices and still have $O(\frac{K|V|}{X})$ memory requirements.

Figure 6 demonstrates our fine-grained sliding window technique with the settings of $X = K = 2$ and $|V| = 4$. Assume that vertices 1, 2, and 3 are serially respectively assigned into P_1 and P_2. Since the shard size is $\lceil \frac{|V|}{X} \rceil = 2$, each partition P_i only maintains the expectation for two vertices in Γ_i. At the very beginning, vertices 1 and 2 are covered by the window. Then $\Gamma_1(2)$ and $\Gamma_2(1)$ are both increased by 1, because of the edges $(1, 2)$ and $(2, 1)$. However, neither $\Gamma_1(4)$ nor $\Gamma_2(3)$ is changed since 3 and 4 are excluded from the window, even

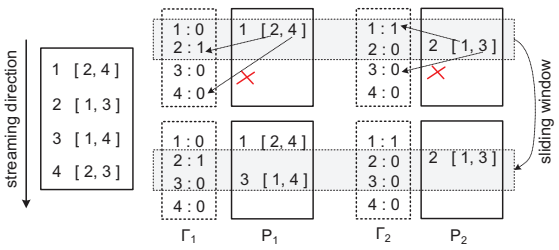


Fig. 6. Memory optimization by sliding window

though there exist edges $(1, 4)$ and $(2, 3)$. When the new vertex 3 arrives, we know the window must be slid to the next one as its upper bound vertex id is less than 3. Similarly, neither $\Gamma_1(1)$ nor $\Gamma_1(4)$ is updated. This is because 1 have been removed and 4 has not been involved. These missing updates on Γ except $\Gamma_1(1)$ clearly generate a negative impact on the quality.

B. Optimizing efficiency with parallel partitioning

Although the streaming partitioner has native efficiency advantages due to the single-pass data scan, the scalability is still poor when processing large graphs. This subsection thereby introduces our parallel efforts.

The workload of a streaming partitioner mainly includes two parts: loading and streaming data from the input disk file, and computing the distribution score for each vertex. Our solution only parallelizes the latter. The reasons for this choice are twofolds. Firstly, our tests reveal that the hot part is score computation, instead of the sequential reads when loading data. Secondly, our sliding window technique essentially requires vertices to be streamed in their numbered order, to effectively count expectations as much as possible, which can be easily guaranteed in the centralized scenario.

Our data-parallel implementation constructs a producer-consumer queue to buffer streamed data. Many threads concurrently take adjacency lists from the buffer and then compute the scores. However, it is most likely that the concurrently processed vertices are also adjacent to each other. In centralized scenarios, they are streamed one by one and then the former can guide the placement of the latter. But in parallel scenarios, such heuristic information will be ignored if we assign them at the same time. That significantly decreases the efforts of reducing the cutting edge number and hence impairs the quality, especially when the parallelism is large.

To alleviate the quality degradation, we enforce these concurrent vertices to actively detect the dependency conflicts. The assignment of a vertex with heavy conflict can be temporarily delayed so that it can fully utilize the heuristic knowledge from its in-out neighbors. However, conflict detection can be expensive, if, for example, we compute the intersection set of neighbors for any pair of vertices. That possibly can offset the parallel benefit.

Here we design a hash-based reversed-counting-table (*RCT*) to quickly detect dependency. For all concurrently processed vertices, *RCT* stores a series of corresponding pairs, each of

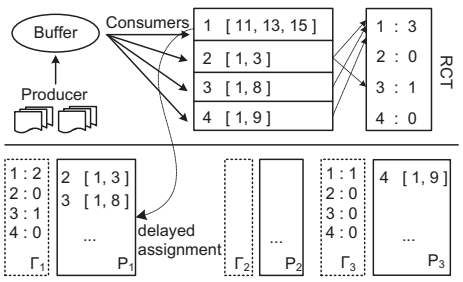


Fig. 7. Dependency detection optimization in parallel computations

TABLE II
DESCRIPTION OF GRAPH DATASETS

Graphs	|V|	|E|	Size
stanford	685,230	7,605,339	58.0MB
uk2005	100,000	3,050,615	17.0MB
eu2015	6,650,532	171,736,545	1.4GB
indo2004	7,414,866	195,418,438	1.5GB
uk2002	18,520,486	298,113,762	2.5GB
web2001	118,142,155	1,019,903,190	9.6GB
sk2005	50,636,154	1,949,412,601	16.0GB
uk2007	108,563,230	3,929,837,236	34.0GB

which consists of the vertex id and a dependency counter. Given a vertex v, we can count the dependency associated with $u \in N^{\text{out}}(v)$ by taking the counter in *RCT* with $O(1)$ time complexity. The counting operation is performed when its out-neighbors are traversed to compute the distribution score. No additional runtime cost is incurred. Once the u's score is computed, it can decide whether or not to delay the assignment by reading its dependency counter. If not, u will be removed from *RCT* so that the latter can store the new vertex taken from the buffer. The size of *RCT* is then fixed as ϵM, where M stands for the parallelism granularity. ϵ indicates at most how many vertices can be temporarily stored by a thread due to heavy dependency conflicts. We enlarge *RCT* by the factor of ϵ because the thread will continuously take new data from the buffer if the current vertex is delayed to be assigned.

Figure 7 shows the dependency detection with the settings of $M = 4$ and $\epsilon = 2$ as an example. Vertices 1-4 are concurrently processed. 1 heavily depends on others and hence, the latter are freely assigned but the former is delayed. After assigning vertices 2-4, the dependency counter associated with 1 decreases to 0 and then we can re-compute its distribution score. Now, it can fully utilize the heuristic information from its in-neighbors in Γ_1. By default, we use the average of non-zero counters as the dependency threshold.

VI. EXPERIMENTS

This section shows experiment studies by comparing our SPNL with well-known streaming and offline partitioners.

A. Benchmarks and Setups

Experimental Environments and Datasets. The experiments are run on a Dell Precision 7920 Server with two Intel XEON Silver 4216 Processors (32 cores and 64 threads, 2.1GHz), 64GB DDR4 ECC RAM, and 4TB SATA AG-Enterprise Hard Drive (7,200RPM). We conduct an overall performance analysis on eight real graph datasets ranging from social networks to web graphs, whose specific information is shown in Table II.

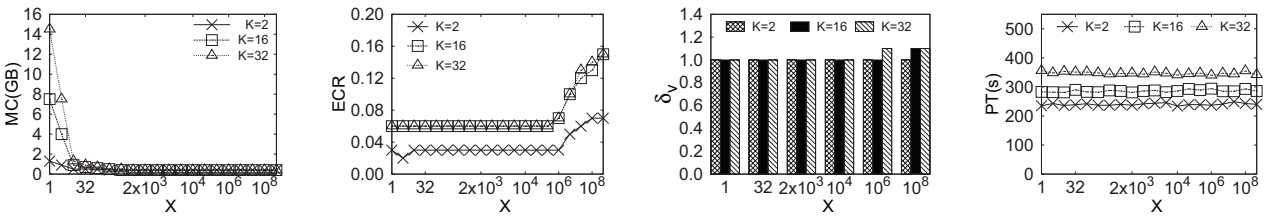
Benchmark Partitioners. We use *SPN* to stand for our basic partitioner based on only in-out neighbors, and *SPNL* for the advanced one when topology locality is additionally utilized. The competitors include classic streaming solutions *LDG* [8] and *FENNEL* [9], the offline *METIS* [6] with preferred quality, and the up-to-date offline *XtraPuLP* [12] with prominent scalability. *LDG*, *FENNEL*, and *METIS* are all centralized and used to validate the quality improvement, while *XtraPuLP* is used as a parallel competitor in shared memory. We implement *LDG*, *FENNEL*, and our proposals by own in Java, and directly use the open-source implementations of offline counterparts written in C++. We are aware that recently there exist many edge partitioning works. However, vertex partitioning and edge partitioning are not comparable because they have very different goals and hence heuristic optimizations. As reported in Ref. [12] and Ref. [39], we cannot perform an end-to-end comparison, no matter re-assigning edges (converting edge cuts to vertex cuts) or vertices (converting vertex cuts to edge cuts). We thereby do not analyze the performance of these vertex partitioning works.

Evaluation Metrics. We focus on both partitioning quality and efficiency. The quality is evaluated by two aspects: the Edge Cut Ratio (*ECR*) evaluated by $\frac{|D|}{|E|}$ where $|D|$ is the total number of cutting edges; and the load balancing factors in terms of vertices and edges, denoted by δ_v and δ_e respectively as shown in Eqs. (1) and (2). For *ECR*, the lower, the better; and for δ_v and δ_e, the closer δ_v and δ_e it is to 1.0, the better. The runtime efficiency is defined as Partitioning Time (*PT*) starting from the point loading the first adjacency list to the point outputting the vertex-assignment route table. Besides, the Memory Consumption (*MC*) is also tested to validate the effectiveness of our sliding window design. Note that we primarily care about the vertex-based balance constraint, but both δ_v and δ_e metrics are reported for a more complete comparative analysis. For multi-constraint *XtraPuLP*, we set $\delta_v = 1.0$ and $\delta_e = 50.0$ to enforce vertex balance.

Below, we first test the effectiveness of sliding window in Sec. VI-B to select a proper shard parameter; and then compare our proposals against existing streaming and offline partitioners respectively in Sec. VI-C and Sec. VI-D. In addition, the symbol 'F' indicates an unsuccessful due to the "out of memory" error.

B. Effectiveness of sliding window

As analyzed in Sec. V-A, the memory consumption of our proposed *SPN* and *SPNL* is sensitive to the number of shards X. The group of experiments thereby test its impact on *MC* and other metrics by manually enumerating X values, so that we can select a proper setting for the following



(a) *MC* (b) *ECR* (c) δ_v (d) *PT*

Fig. 8. Impact of different numbers of shards (X) on all metrics (*SPNL*, *web2001*)

TABLE III
THE OVERALL PERFORMANCE WHEN COMPARING *SPN* AND *SPNL* WITH STREAMING PARTITIONERS (K=32, *PT*: SECONDS).

Graphs	LDG				FENNEL				SPN				SPNL			
	ECR	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*
stanford	0.37	1.2	3.2	2.0	0.41	1.1	2.1	2.0	0.30	1.12	2.42	2.1	0.18	1.0	3.1	2.1
uk2005	0.49	1.2	2.1	0.7	0.54	1.1	2.1	0.7	0.39	1.2	2.4	0.8	0.32	1.1	2.7	0.8
eu2015	0.29	1.4	18.6	25.7	0.30	1.1	18.5	27.0	0.23	1.2	19.4	31.0	0.17	1.1	18.4	31.9
indo2004	0.31	1.0	9.0	28.8	0.32	1.1	8.6	30.7	0.19	1.0	8.5	33.2	0.04	1.0	8.6	33.3
uk2002	0.41	1.0	1.2	58.5	0.43	1.0	1.1	58.0	0.25	1.0	1.2	63.7	0.05	1.0	1.4	69.6
web2001	0.43	1.0	1.1	295.2	0.47	1.0	1.1	303.1	0.25	1.0	1.2	337.6	0.06	1.0	1.3	344.0
sk2005	0.43	1.1	1.4	326.8	0.47	1.1	1.3	331.6	0.26	1.0	1.4	348.8	0.10	1.0	1.6	369.7
uk2007	0.36	1.0	2.4	685.5	0.38	1.1	2.2	707.8	0.24	1.0	2.4	815.6	0.03	1.0	2.1	875.6

tests. Without loss of generality, we run *SPNL* over *web2001* as a test case, and the similar results can be observed on other combination cases. As shown in Figure 8(a), *MC* of course significantly decreases when increasing X, because the shard/window size is inversely proportional to X, and with sliding window, we need to allocate spaces for only one shard, instead of all vertices, for each partition. But the benefit dramatically drops when X becomes further large, as storing expectation estimation now does not dominate the memory consumption. On the other hand, Figure 8(b) tells us that an extremely large X yields a clear degradation about *ECR*. This is because a newly arrived vertex can only update and see expectation counts for vertices within the window; a small window clearly reduces the probability of looking up useful counts and hence increases *ECR*. Differently, Figure 8(c) and (d) reveal that δ_v (also δ_e) and *PT* always keep steady. The reason is that capacity estimation and expectation lookup (with $O(1)$ time complexity) are both independent of the length of the shard/window size. Figure 8 also shows that these metrics are not sensitive to the number of partitions, i.e., K.

Overall, a quite large range of X can improve the performance on *MC* without a negative impact on other metrics. Thus, for both *SPN* and *SPNL* in the following experiments, we empirically give $X = \min\{\alpha K, \frac{|V|}{\beta K}\}$, parameterized by $\alpha = 4$ and $\beta = 10^4$ by default. Taking *web2001* with K=32 as an example, the computed X is 128. Table IV then reports the memory consumption (*MC*) of streaming and offline partitioners. Because *METIS* and *XtraPuLP* need to load the whole input graph into memory for complex analysis, the two offline parritioners consume memory resources at least linearly related to the number of edges [24]. That yields a significant increase on *MC*, compared with streaming partitioners *LDG* and *FENNEL*. For our *SPNL*, its *MC* value is up to 14.53GB at the extreme case where X=1. However, by decreasing X down to 128, the *MC* value is comparable to existing streaming partitioners, and the negative impact on quality (like *ECR*) is negligible.

TABLE IV
SPACE COMPLEXITY EVALUATION

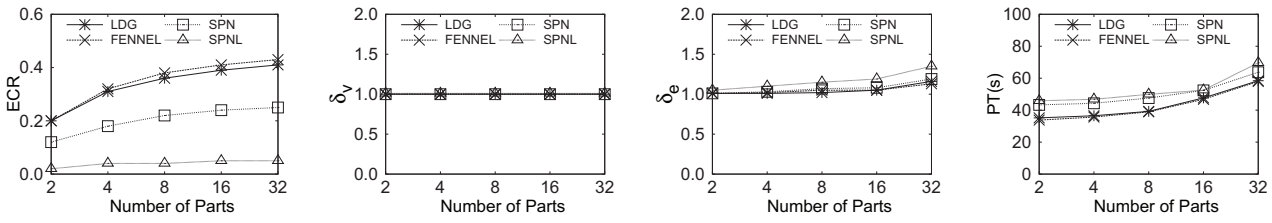
Methods	*MC*(GB)	*ECR*	Space Complexity
LDG	0.44	0.4318	$O(\|V\| + K + \max d)$
FENNEL	0.44	0.4733	$O(\|V\| + K + \max d)$
METIS	$\geq$ 3.80	0.1027	$\geq O(\|E\|)$ [24]
XtraPuLP	$\geq$ 3.80	0.2742	$\geq O(\|E\|)$ [24]
SPNL(X=1)	14.53	0.0620	$O(\|V\| + 3K + \frac{K\|V\|}{X} + \max d)$
SPNL(X=128)	0.55	0.0623	

C. Compared with streaming partitioners

Now we compare our *SPN* and *SPNL* with streaming partitioners *LDG* and *FENNEL*. Table III summarizes the overall performance with $K = 32$. *SPN* and *SPNL* generally outperform their counterparts in quality (with lower *ECR* and comparable δ_v and δ_e), but have slight runtime latency because of computing complex heuristics.

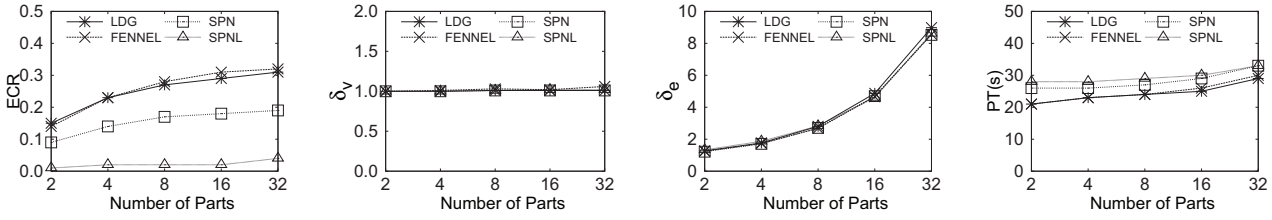
In particular, *SPN* reduces *ECR* by at most 47% compared to traditional *LDG* and *FENNEL* (from 19%), since the former considers the distribution of not only out-neighbors but also in-neighbors. Adding topology locality creates another gap between *SPN* and *SPNL*. The improvement thereby increases to 92% (from 35%).

Another observation is that although all partitioners perform well in δ_v, the distribution of edges is very skewed, since



(a) *ECR* (b) δ_v (c) δ_e (d) *PT*

Fig. 9. Impact of K on all metrics when compared with streaming partitioners (*uk2002*)



(a) *ECR* (b) δ_v (c) δ_e (d) *PT*

Fig. 10. Impact of K on all metrics when compared with streaming partitioners (*indo2004*)

TABLE V
THE OVERALL PERFORMANCE WHEN COMPARING *SPNL* WITH OFFLINE PARTITIONERS (K=32, *PT*: SECONDS).

Graphs	METIS				XtraPuLP (centralized/parallel)				SPNL (centralized/parallel)			
	ECR	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*
stanford	0.21	1.0	2.6	2.6	0.41/0.37	1.0/1.0	3.5/3.3	7.3/6.0	0.18/0.19	1.0/1.0	3.1/3.1	2.1/2.1
uk2005	0.28	1.0	2.7	1.3	0.60/0.64	1.0/1.0	1.8/1.6	1.4/1.2	0.32/0.31	1.1/1.1	2.7/2.6	0.8/0.9
eu2015	0.16	1.0	18.7	649.7	0.36/0.68	1.0/1.0	18.3/11.5	143.1/108.8	0.17/0.18	1.1/1.2	18.4/18.6	31.9/17.8
indo2004	0.06	1.0	8.5	117.7	0.46/0.69	1.0/1.0	8.5/8.1	106.7/88.9	0.04/0.04	1.0/1.0	8.6/8.6	33.3/20.8
uk2002	0.06	1.0	1.7	318.8	0.22/0.30	1.0/1.0	1.6/1.2	224.5/187.5	0.05/0.05	1.0/1.0	1.4/1.4	69.6/34.4
web2001	0.10	1.0	1.4	2757.9	0.27/0.33	1.0/1.0	1.4/1.1	1721.5/1401.2	0.06/0.07	1.0/1.0	1.3/1.3	344.0/209.7
sk2005	F	F	F	F	0.35/0.33	1.0/1.0	1.4/1.3	880.5/652.8	0.10/0.10	1.0/1.0	1.6/1.6	369.7/161.6
uk2007	F	F	F	F	F/F	F/F	F/F	F/F	0.03/0.03	1.0/1.0	2.1/2.0	875.6/327.0

vertices usually have scale-free degrees. However, all of them can support for δ_e if necessary as reported in Sec. VI-E.

Figure 9 and Figure 10 further plots all metrics as a function of K, using *uk2002* and *indo2004* as an example graph. All partitioners work well in δ_v and δ_e. But *ECR* and *PT* inevitably increase when K becomes large, because finding a proper partition for a given vertex is more difficult and more time-consuming score computations are required. Generally, all these streaming partitioners have prominent scalability in terms of quality and efficiency.

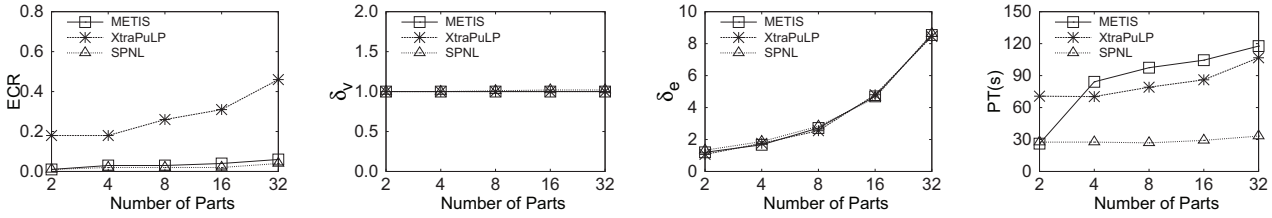
D. Compared with offline partitioners

Next, we focus on comparing our proposals with complex offline partitioners *METIS* and *XtraPuLP*. Since *SPNL* consistently beats *SPN* in all metrics, here we only report data associated with the former for brevity. Recall that both *SPNL* and *XtraPuLP* can be run in parallel in shared memory. We then explore their features in centralized and parallel (with manually tested optimal granularity) settings.

Table V reports our experiment results where all involved partitioners have different advantages. In the centralized setting, we generally observe that *METIS* has the best quality in a few cases but consumes large memory and compute resources; *XtraPuLP* runs faster than *METIS* at expense of high *ECR*; our *SPNL* is comparable to or even better than *METIS* in *ECR* in all cases, and always has the most prominent performance in *PT*, even though the other two are written in more efficient C++. In particular, *SPNL* reduces *ECR* up to 40%, compared with *METIS*, and runs 20X faster at most than it. For *XtraPuLP*, the two factors are 91% and 7X, respectively.

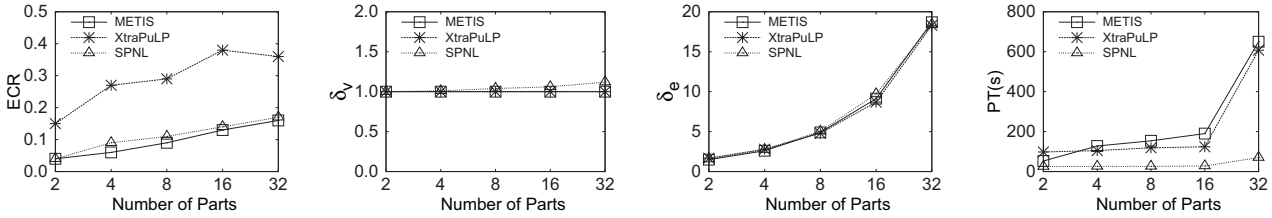
Note that we fail to run *METIS* on two large graphs *sk2005* and *uk2007* because of memory errors. The reason is that the multilevel coarsening operations generate a large amount of intermediate data and hence the memory is quickly exhausted. *XtraPuLP* alleviates this bottleneck by label propagation, but still fails on the largest graph *uk2007*, since loading the whole graph and storing labels pose great challenges for limited memory resources.

Besides, the parallel variants of *SPNL* and *XtraPuLP* can respectively decrease the runtime by 63% and 26% at most, compared with their centralized tests. Because of the parallel



(a) ECR (b) δ_v (c) δ_e (d) PT

Fig. 11. Impact of K on all metrics when compared with offline partitioners (*indo2004*)



(a) ECR (b) δ_v (c) δ_e (d) PT

Fig. 12. Impact of K on all metrics when compared with offline partitioners (*eu2015*)

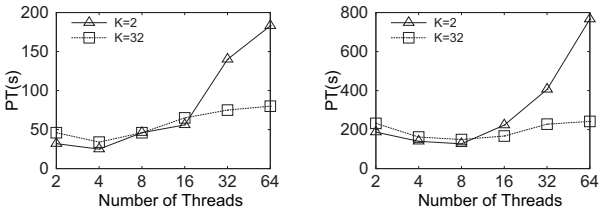
optimization in Sec. V-B, now the speedup of *SPNL* compared against *XtraPuLP* is up to roughly 15, which is larger than the factor 7 in centralized. Also, the parallel *SPNL* can run 2 times at most faster than the centralized *LDG* shown in Table III. However, the read/write conflicts caused by multiple threads make it difficult to accurately place a vertex into the right partition. *XtraPuLP* thereby generates up to 47% quality degradation in *ECR*. But for *SPNL*, the dependency-reduced optimization largely reduces the conflicting probability, which narrows the maximal of the corresponding degradation to 6%.

Resembling the comparison against streaming counterparts, now we explore features when varying K, using *indo2004* and *eu2015*. We can see the similar variation with that in Figure 11 and Figure 12 for all metrics except δ_e. δ_e roughly nearly increases with K. Compared with *uk2002* in Figure 9, *indo2004* and *eu2015* here has a very skewed degree distribution, which can also be validated by their different δ_e values in Tables III and V. When K increases, the skewness yields a different numbers of edges among partitions in high probability.

Figure 13 further studies how the runtime efficiency *PT* of *SPNL* scales when varying the number of concurrent threads. We clearly see that *PT* first decreases due to parallel acceleration, and then increases due to warm-up costs caused by scheduling and essential synchronizing operations. A "sweet spot" can achieve the best speedup but it varies with specific settings. Generally, a large graph requires a big concurrent granularity. Taking the small graph *uk2002* and the large graph *sk2005* as examples, the "sweet spot" changes from 4 to 8.

E. Performance with edge-based balance

We finally investigate the performance of all partitioners when focusing on the edge-based balance constraint. Since



(a) uk2002 (b) sk2005

Fig. 13. Evaluating the optimal parallel granularity of *SPNL*

METIS does not provide such a setting choice, now we only report others' results. For *XtraPuLP*, here we set $\delta_e = 1.0$ and $\delta_v = 50$ to enforce edge-based balance. Table VI shows the testing reports. *SPNL* generally has better quality than *LDG* and *FENNEL*, with slight efficiency degradation. *XtraPuLP* achieves the best *ECR* optimization but the edge-based balance constraint is not strictly guaranteed. In contrast, the three streaming partitioners make a comprise between reducing *ECR* as much as possible and strictly following the constraint $\delta_e = 1.0$. The *ECR* metric thereby heavily degrades, especially for very skewed graphs like *stanford*, *eu2015*, and *indo2004*.

VII. Conclusion

This paper investigates the shortcomings of existing streaming and offline partitioners, and then proposes a new streaming replacement by fully utilizing knowledge. Optimizations about memory consumption and parallel acceleration are also proposed to further enhance its performance. Experiments validate that the proposed solution works well with prominent partitioning quality and efficiency.

TABLE VI
PERFORMANCE ANALYSIS WHEN FOCUSING ON THE EDGE-BASED BALANCE CONSTRAINT ($K = 32$, PT: SECONDS).

Dataset	LDG				FENNEL				XtraPuLP				SPNL			
	ECR	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*	*ECR*	δ_v	δ_e	*PT*
stanford	0.45	1.5	1.1	1.9	0.50	1.3	1.1	2.0	0.11	2.2	3.0	6.3	0.32	1.5	1.0	2.0
uk2005	0.55	1.3	1.1	0.6	0.72	1.2	1.1	0.6	0.24	3.7	2.7	1.3	0.39	2.4	1.1	0.8
eu2015	0.83	1.6	1.0	27.7	0.86	2.5	1.1	26.9	0.41	14.0	9.1	138.3	0.74	2.6	1.0	31.1
indo2004	0.54	1.3	1.0	29.2	0.55	1.4	1.1	30.8	0.09	9.4	7.9	95.3	0.29	1.6	1.0	31.0
uk2002	0.41	1.1	1.0	54.8	0.42	1.1	1.0	62.8	0.09	5.3	1.0	203.5	0.07	1.2	1.0	66.2
web2001	0.44	1.0	1.0	279.3	0.44	1.0	1.0	292.7	0.16	5.2	1.6	2087.7	0.09	1.2	1.0	352.2
sk2005	0.44	1.2	1.0	321.3	0.50	1.1	1.1	335.2	0.14	4.3	1.0	833.3	0.13	1.3	1.0	356.4
uk2007	0.40	1.2	1.0	681.2	0.42	1.2	1.1	714.0	F	F	F	F	0.08	1.3	1.0	867.2

REFERENCES

[1] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of SIGMOD*. ACM, 2010, pp. 135–146.

[2] Z. Wang, Y. Gu, Y. Bao, G. Yu, J. X. Yu, and Z. Wei, "Hgraph: I/o-efficient distributed and iterative graph computing by hybrid pushing/pulling," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 5, pp. 1973–1987, 2021.

[3] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu, "Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing," in *SIGMOD Conference*. ACM, 2016, pp. 479–494.

[4] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is np-hard," *Inf. Process. Lett.*, vol. 42, no. 3, pp. 153–159, 1992.

[5] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*, ser. Lecture Notes in Computer Science, 2016, vol. 9220, pp. 117–158.

[6] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distributed Comput.*, vol. 48, no. 1, pp. 96–129, 1998.

[7] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *Proc. of ICDE*. IEEE Computer Society, 2014, pp. 568–579.

[8] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. of SIGKDD*, 2012, pp. 1222–1230.

[9] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: streaming graph partitioning for massive scale graphs," in *Proc. of WSDM*. ACM, 2014, pp. 333–342.

[10] M. F. Faraj and C. Schulz, "Buffered streaming graph partitioning," *ACM J. Exp. Algorithmics*, vol. 27, pp. 1.10:1–1.10:26, 2022.

[11] S. Gong, Y. Zhang, and G. Yu, "Accelerating large-scale prioritized graph computations by hotness balanced partition," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 4, pp. 746–759, 2021.

[12] G. M. Slota, C. Root, K. D. Devine, K. Madduri, and S. Rajamanickam, "Scalable, multi-constraint, complex-objective graph partitioning," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 12, pp. 2789–2801, 2020.

[13] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 25:1–25:39, 2015.

[14] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.

[15] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. of International Conference on High-Performance Computing and Networking*. Springer, 1996, pp. 493–498.

[16] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. of WSDM*. ACM, 2013, pp. 507–516.

[17] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "JA-BE-JA: A distributed algorithm for balanced graph partitioning," in *SASO*. IEEE Computer Society, 2013, pp. 51–60.

[18] G. M. Slota, S. Rajamanickam, K. D. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *Proc. of IPDPS*. IEEE Computer Society, 2017, pp. 646–655.

[19] D. Deng, F. Bai, Y. Tang, S. Zhou, C. Shahabi, and L. Zhu, "Label propagation on k-partite graphs with heterophily," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 3, pp. 1064–1077, 2021.

[20] J. Nishimura and J. Ugander, "Restreaming graph partitioning: simple versatile algorithms for advanced balancing," in *Proc. of SIGKDD*, 2013, pp. 1106–1114.

[21] G. Echbarthi and H. Kheddouci, "Fractional greedy and partial restreaming partitioning: New methods for massive graph partitioning," in *Proc. of IEEE BigData*. IEEE Computer Society, 2014, pp. 25–32.

[22] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: stream-based partitioning for power-law graphs," in *Proc. of CIKM*. ACM, 2015, pp. 243–252.

[23] D. Kong, X. Xie, and Z. Zhang, "Clustering-based partitioning for large web graphs," in *Proc. of ICDE*. IEEE, 2022, pp. 593–606.

[24] R. Mayer, K. Orujzade, and H. Jacobsen, "Out-of-core edge partitioning at linear run-time," in *Proc. of ICDE*. IEEE, 2022, pp. 2629–2642.

[25] Y. Li, C. Li, A. Orgerie, and P. R. Parvédy, "WSGP: A window-based streaming graph partitioning approach," in *Proc. of 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021*. IEEE, 2021, pp. 586–595.

[26] R. Mayer and H. Jacobsen, "Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints," in *Proc. of SIGMOD*. ACM, 2021, pp. 1289–1302.

[27] T. Ayall, H. Duan, C. Liu, F. Gereme, M. Abegaz, and M. Deleli, "Taking heuristic based graph edge partitioning one step ahead via offstream partitioning approach," in *Proc. of ICDE*. IEEE, 2021, pp. 2081–2086.

[28] S. Ji, C. Bu, L. Li, and X. Wu, "Local graph edge partitioning," *ACM Trans. Intell. Syst. Technol.*, vol. 12, no. 5, pp. 61:1–61:25, 2021.

[29] A. C. Zhou, B. Shen, Y. Xiao, S. Ibrahim, and B. He, "Cost-aware partitioning for efficient large graph processing in geo-distributed datacenters," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 7, pp. 1707–1723, 2020.

[30] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *J. Parallel Distributed Comput.*, vol. 48, no. 1, pp. 71–95, 1998.

[31] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, no. 6-8, pp. 318–331, 2008.

[32] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," in *Proc. of IPDPS*. IEEE, 2010, pp. 1–12.

[33] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning," *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 11, pp. 2710–2722, 2020.

[34] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *Proc. of ICDE*. IEEE Computer Society, 2017, pp. 1083–1094.

[35] Z. Shi, J. Li, P. Guo, S. Li, D. Feng, and Y. Su, "Partitioning dynamic graph asynchronously with distributed FENNEL," *Future Gener. Comput. Syst.*, vol. 71, pp. 32–42, 2017.

[36] Q. Hua, Y. Li, D. Yu, and H. Jin, "Quasi-streaming graph partitioning: A game theoretical approach," *IEEE Trans. Parallel Distributed Syst.*, vol. 30, no. 7, pp. 1643–1656, 2019.

[37] M. F. Faraj and C. Schulz, "Recursive multi-section on the fly: Shared-memory streaming algorithms for hierarchical graph partitioning and process mapping," in *Proc. of CLUSTER*. IEEE, 2022, pp. 473–483.

[38] N. Wang, Z. Wang, Y. Gu, Y. Bao, and G. Yu, "TSH: easy-to-be distributed partitioning for large-scale graphs," *Future Gener. Comput. Syst.*, vol. 101, pp. 804–818, 2019.

[39] M. Hanai, T. Suzumura, W. J. Tan, E. S. Liu, G. Theodoropoulos, and W. Cai, "Distributed edge partitioning for trillion-edge graphs," *VLDB Endow.*, vol. 12, no. 13, pp. 2379–2392, 2019.